Seamless Integration of Hardware Interfaces in UML-based MDSE Tools

Lars Huning[1], Timo Osterkamp[1], Marco Schaarschmidt[2] and Elke Pulvermüller[1]
[1]*Institute of Computer Science, University of Osnabrück, Wachsbleiche 27, 49090 Osnabrück, Germany*
[2]*Faculty of Engineering and Computer Science, University of Applied Sciences Osnabrück, Germany*

Keywords: Automatic Code Generation, Embedded Systems, Hardware Interfaces, Model-Driven Software Engineering.

Abstract: Model-Driven Software Engineering (MDSE) promotes the use of models for software development. One application of MDSE is the development of embedded systems, whose size and complexity are growing steadily. Usage of MDSE for embedded systems often consists of creating high-level architectures, e.g., with the Unified Modeling Language (UML), while the actual implementation of the system is done manually. One reason for this is the semantic gap between high-level UML models and the low-level programming associated with microcontrollers, i.e., imperative programming at the register level. This paper proposes an approach for the seamless integration of hardware interfaces, e.g., GPIOs or UARTs, in UML-based MDSE tools. This enables developers to create their application continously in the MDSE tool, instead of resorting to manual programming outside the environment of the MDSE tool. For this, we present an approach that describes how object-oriented hardware abstraction layers may be seamlessly integrated in MDSE tools. Furthermore, we provide a GUI tool for hardware interfaces that enables the initial configuration of these interfaces. An automatic code generation approach may subsequently be used to generate the initialization code for the hardware interfaces of a microcontroller. We present a use case for our approach in which the software application of an embedded system is ported to several other microcontrollers from different manufacturers.

1 INTRODUCTION

The size and complexity of embedded systems has been increasing rapidly in the last years (Trindade et al., 2014). For example, a modern car may contain more than seventy electronic control units and run more than 100 million lines of code (Charette, 2009). In the past, a growth in size and complexity of desktop applications has led to the adoption of object-oriented paradigms and programming languages, e.g., Java and C++. Subsequently, object-oriented modeling techniques, e.g., the Unified Modeling Language (UML) (OMG UML, 2017), as well as *Model-Driven Software Engineering (MDSE)* (Brambilla et al., 2012) have been proposed to further deal with the growing size and complexity of systems. MDSE and object-oriented techniques are also increasingly adopted in the embedded domain, e.g., for automatic code generation (Huning et al., 2020) or safety-critical applications, such as medicinal devices (Kim et al., 2013). However, at the time this paper is written, model-driven approaches in the embedded domain are mainly used to design high-level architectures. The actual implementation is still done manually (Kim et al., 2013). One of the reasons for these manual implementations is the semantic gap between high-level models and the implementation platforms (Kim et al., 2013). This is also true for low-level hardware interactions with the platform, e.g., hardware drivers or board support packages. These low-level elements rarely conform to object-oriented paradigms, which makes a seamless integration with the high-level architecture created with MDSE tools difficult, as the interaction with hardware drivers may only be represented in UML in a very limited fashion, i.e., via non-object-oriented, static references. While it is possible to automatically generate code skeletons of the high-level architecture with MDSE tools, and subsequently implement the hardware interactions manually, this type of development may quickly lead to a divergence between the model and the source code (Selic, 2008). This reduces the benefits obtained by modeling. This paper, in contrast, advocates for the full code generation of application with the help of MDSE tools, by bridging the gap between object-oriented modeling of the high-level architecture and low-level interaction with the hardware. For this, we provide an approach for the seamless inte-

Huning, L., Osterkamp, T., Schaarschmidt, M. and Pulvermüller, E.
Seamless Integration of Hardware Interfaces in UML-based MDSE Tools.
DOI: 10.5220/0010575802330244
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 233-244
ISBN: 978-989-758-523-4
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

gration of an object-oriented *Hardware Abstraction Layer (HAL)* in MDSE. Furthermore, this paper introduces a tool for automatically generating the hardware initialization of hardware interfaces, e.g., *General Purpose Input/Output (GPIO)* or *Analog Digital Converter (ADC)*. The generated hardware initialization may be seamlessly integrated with MDSE tools, thereby allowing developers to interact with hardware in an object-oriented fashion in MDSE tools. Thus, developers for embedded systems may fully profit from the advantages promised by MDSE, e.g., an increase in developer productivity and a decrease in the number of bugs within the system (Bunse et al., 2007; Kashif et al., 2009).

HALs provide a uniform interface for interacting with the hardware of microcontrollers, e.g., hardware interfaces such as GPIO, ADC or *Universal Asynchronous Receiver Transmitter (UART)*. This enables developers to use the same interface for interacting with the hardware of different microcontrollers. This reduces the level of detail to which a developer has to understand the inner workings of a specific microcontroller. This is not only important when developers start a project with a microcontroller that is unfamiliar to them, but also for legacy applications. In some industries, long-lasting delivery commitments exist, e.g., providing spare parts for as long as a decade. During this time frame, the type of microcontroller used for a specific product may no longer be sold by its manufacturer. Therefore, the application has to be ported to another microcontroller. In case the application was initially developed with a HAL, it may be sufficient to exchange the underlying implementation of the hardware interfaces in order to port the application to the new microcontroller. On the other hand, if no HAL was used for development, e.g., the hardware is accessed directly via registers by code snippets that are scattered across the whole application, the entire application may need to be modified. Thus, a HAL may not only ease the training period for developers facing an unfamiliar microcontroller, but also improve the portability of embedded systems. While there exist several HALs, e.g., (ARM Limited, 2021a; ARM Limited, 2021b), they often do not adhere to an object-oriented programming paradigm, which makes integration with an UML-based MDSE tool difficult. Even those few HALs that are object-oriented, e.g., (modm., 2021), do not consider integration with MDSE tools. Due to their non-trivial size and complexity, which includes a multitude of templates and generator files that ultimately generate the HAL, their integration with MDSE tools remains difficult. This paper introduces an object-oriented HAL and a development workflow that enables a seamless integration of this HAL into MDSE tools.

While a HAL provides a uniform interface for accessing the hardware interfaces of a microcontroller, these hardware interfaces also often have to be initialized prior to their usage. For example, the exact pin which a UART uses for data transmission needs to be specified. For more complex hardware interfaces, this initialization is a non-trivial process in which the order of initialization statements is of importance. While there exist *Graphical User Interface (GUI)* tools that aim to simplify and partially automate this task, e.g., (ST Microelectronics, 2021a; Infineon, 2021b), these tools are often limited to microcontrollers from a specific manufacturer. Moreover, the code generated automatically by these tools, is not object-oriented and thus once again hard to integrate with UML-based MDSE tools. This paper presents a novel GUI tool for specifying the configuration of how hardware interfaces should be initialized. For this, we introduce an *Extensible Markup Language (XML)*-based format for describing microcontrollers and their configurations that is independent of a specific manufacturer. Furthermore, we provide an automated workflow that describes how the automatically generated code from these configurations may be seamlessly integrated into the development with MDSE tools.

In summary, this paper provides the following novelties:

- An approach for the seamless integration of an object-oriented HAL with MDSE tools.
- A GUI tool for automatically generating hardware initializations for microcontrollers, whose output may be seamlessly integrated with MDSE tools.
- A practical demonstration of our approach by developing an application with the previously mentioned concepts for microcontrollers from NXP, Infineon and ST Microelectronics.

The remainder of this paper is organized as follows: Section 2 provides an overview of the development workflow to be used with our approach. Section 3 introduces the GUI tool for specifying the initialization configuration of hardware interfaces, while Section 4 introduces the object-oriented HAL. Section 5 describes the integration of these contributions with MDSE tools. A practical demonstration of our approach is described in Section 6, where the approach is used to create an embedded system for microcontrollers from different manufacturers. We describe related work in Section 7. Section 8 concludes our paper.

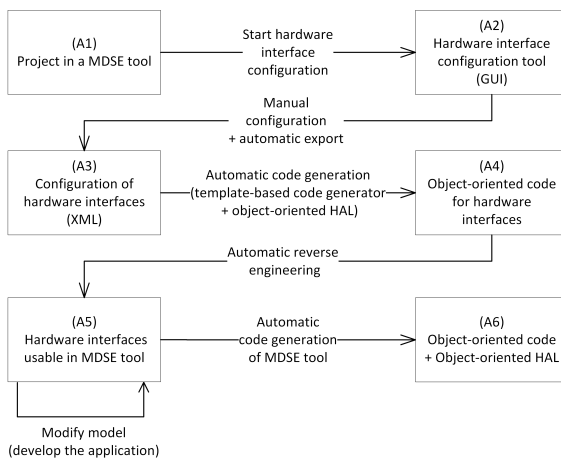

Figure 1: Overview of the development workflow used in this paper.

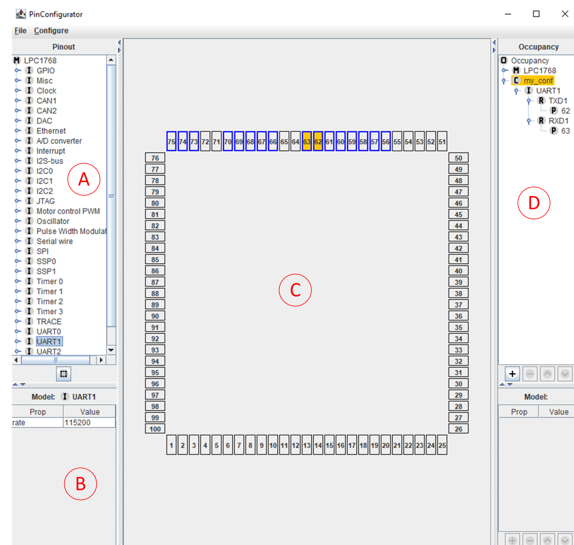

Figure 2: Screenshot of the GUI tool developed in this paper for the purpose of configuring the hardware initialization of microcontrollers.

2 OVERVIEW OF THE APPROACH

This section presents an overview of the development workflow used for the approach presented in this paper. The workflow is shown in Figure 1. Only one direction of the workflow is shown, with one phase smoothly transitioning into the next. In practice, iterations and returning to a prior phase for modifications is often necessary. In order to improve the legibility of the figure, this is not shown in Figure 1.

At the beginning of the workflow, a project within an MDSE tool exists, i.e., a set of UML diagrams that describe the structure and behavior of the application (cf. (A1) in Figure 1). The diagrams may already describe the whole application minus the hardware interactions. Alternatively, the diagrams may be empty at this point in time and modified later in step (A5) of the workflow shown in Figure 1. From this MDSE tool, a separate GUI tool may be started (cf. (A2) in Figure 1). This tool enables developers to specify the hardware interfaces their application uses, as well as configure the initialization for these hardware elements. The tool is described in detail in Section 3. Once the developer has finished his configuration,it may be exported automatically as an XML file (cf. (A3) in Figure 1). The XML file serves as the input to a code generation engine, that generates source code for the exported configurations of the hardware interfaces (cf. (A4) in Figure 1). The generated source code utilizes an object-oriented HAL, which is introduced in Section 4 of this paper. The employed HAL interfaces may subsequently be imported into the MDSE tool via automatic reverse engineering. This way, the HAL interfaces, and therefore hardware interactions, are accessible to developers within the MDSE tool in an object-oriented fashion (cf. (A5) in Figure 1). As the hardware is now accessible within the model, a developer may now develop the whole application within the model in an object-oriented manner. With our approach, the developer is no longer required to include manual references to low-level hardware interactions at the register-level. Once this process is finished, the MDSE tool is capable of generating the source code for this application automatically. For compilation, the implementation of the HAL interfaces for the actual microcontroller used in the project has to be linked with the generated source code (cf. (A6) in Figure 1).

3 PinConfig TOOL

This section describes a novel GUI tool designed to configure the hardware interfaces of a microcontroller. It is used in steps (A2) and (A3) of the workflow described in Section 1. The tool is referred to as *PinConfig tool* for the remainder of this paper. Section 3.1 presents the actual GUI of the tool, while Section 3.2 describes how the characteristics of microcontrollers are stored internally. Section 3.3 presents the XML format used to store the configurations for a specific microcontroller in a specific project.

3.1 Graphical User Interface

Figure 2 shows the GUI of the PinConfig tool. Panel A contains a list view, where the current microcontroller that is configured is displayed on top (LPC1768 (NXP, 2021a) in Figure 2). In the remainder of the list view, the different hardware interfaces of the microcontroller are displayed. In Figure 2, the UART1 interface of the LPC1768 is selected. Panel B shows additional configuration values for the interface currently selected in panel A. In Figure 2, this is the baudrate of the UART selected in panel A.

Panel C shows the board layout of the microcontroller and enables users to select the pins to use for the hardware interface highlighted in panel A. The pins that may be chosen are highlighted with a blue border, while the pins that have actually been chosen are marked yellow. In Figure 2, the pins 62 and 63 are configured as a transmitting and receiving pin for UART1 respectively. Figure 2 shows a Quad Flat Package (QFP) structure of the pins of the microcontroller, i.e., the pins are located at the sides of the microcontroller. The tool also supports the Ball Grid Array (BGA) structure, where the pins are located under the bottom of the chip.

Panel D shows the currently configured hardware interfaces of the microcontroller. In Figure 2, UART1 is configured to use the pin 62 for transmission (TXD1) and pin 63 for reception (RXD1). The purpose, for which a pin is used, e.g., marking pin 62 as TXD1, may be selected in a separate submenu.

3.2 Representing Microcontrollers

The GUI presented in Section 3.1 contains a variety of information about the microcontroller that is configured, e.g., the board structure, the available hardware interfaces and the roles these interfaces may perform. This section describes a novel XML file that is used to describe the structure of the microcontroller internally. Listing 1 shows the general structure of the XML file.

At the start of the XML file (lines 1-6) in Listing 1, the basic information about the microcontroller, e.g., its name, the number of its pins and the board layout (e.g., LQFP or BGA) is stated. The information about the hardware interfaces is divided among the following tags, *<interfaces>*, *<roles>* and *<pins>*. The <interfaces> tag represents an actual interface, e.g., a UART, while the <pin> tag represents the pins that are available on the microcontroller. The <role> tag serves as a foreign key between these elements, thereby enabling the PinConfig tool to highlight which pins are available for a specific interface on a given microcontroller.

```
1  <microcontroller id="lpc1768">
2    <info>
3      <name>LPC1768</name>
4        <pincount>100</pincount>
5        <package>LQFP</package>
6    </info>
7    <interfaces>
8      <!--cf. Listing 2-->
9    </interfaces>
10   <roles>
11     <!--cf. Listing 3-->
12   </roles>
13   <pins>
14     <!--cf. Listing 4-->
15   </pins>
16 </microcontroller>
```

Listing 1: General XML structure for representing a microcontroller.

```
1  <interface id="uart0">
2    <props>
3      <prop name="rate">115200</prop>
4      <!-- ... -->
5    </props>
6    <name>UART0</name>
7    <roles>
8      <role id="txd0"></role>
9        <!--...-->
10   </roles>
11 </interface>
```

Listing 2: The structure of the <interface> XML element.

Listing 2 shows the <interface> tag. Besides the name of the hardware interface it belongs to (cf. *UART0* in line 6 in Listing 2), the tag also contains the properties of this hardware interface. For example, in line 3 of Listing 2 the property *rate* is defined, indicating that the interface *UART0* contains a property that represents the baudrate of the UART. The following value (115200) represents a default value that may be changed in the GUI described in Section 3.1. Lines 7-10 of Listing 2 indicate the possible roles the UART may perform.

While roles are referenced via their id in the interface elements, they are defined in their own XML elements. This is shown in Listing 3. Besides introducing the id that is also referenced by the interface elements (cf. line 1 in Listing 3), the roles may contain a set of properties (cf. line 2-5 in Listing 3), a note that contains the information from the data sheet of the microcontroller (cf. line 6 in Listing 3) and a symbol for this role that is shown in the GUI of the tool (cf. line 7 in Listing 3).

Listing 4 shows the pin element, which gives each pin a name (cf. line 3 in Listing 4) and a position on the board, which may be obtained from the data sheet

```
1  <role id="txd0">
2    <props>
3      <prop name="type">O</prop>
4      <!--...-->
5    </props>
6    <note>Transmitter for UART0.</note>
7    <symbol>TXD0</symbol>
8  </role>
```

Listing 3: The structure of the <role> XML element.

of the microcontroller (cf. lines 4-7 of Listing 4). Furthermore, each pin may reference a set of roles, e.g., *txd0* in line 9 of Listing 4.

```
1  <pin id="98">
2    <props> </props>
3    <name>98</name>
4    <position>
5      <x>0</x>
6      <y>23</y>
7    </position>
8    <roles>
9      <role id="txd0"/>
10     <!--...-->
11   </roles>
12 </pin>
```

Listing 4: The structure of the <pin> XML element.

3.3 Representing Hardware Configurations

The GUI described in Section 3.1 is capable of creating an XML file in which the current configurations regarding the pins and properties are exported. The structure of the XML file is shown in Listing 5. The XML tags used in Listing 5 are similar to those introduced in Section 3.2. However, where the tags in Section 3.2 describe the possible configurations for a microcontroller, the tags in Listing 5 refer to a specific configuration for a specific project in which the microcontroller is used. Therefore, some tags have the prefix *configured_* prepended. The tag <configured_interface> describes the specific configuration for an interface (a UART in line 3-19 of Listing 5). The tag <configured_role> describes the configuration of a specific role that the interface is configured for (in contrast to a possible role, which the interface *may* be configured for, as in the <role> tag introduced in Section 3.2). This includes the pin on which the role is actually performed (<configured_pin>), e.g., pin 98 in line 15 of Listing 5). The <prop> tag is still employed to provide further configurations, e.g, the baudrate for the configured UART (cf. line 6 in Listing 5).

```
1  <occupancy>
2    <configured_interfaces>
3      <configured_interface>
4        <name>UART0</name>
5        <props>
6          <prop name="rate">115200</prop>
7        </props>
8        <configured_roles>
9          <configured_role>
10           <props>
11             <prop name="type">o</prop>
12           </props>
13           <symbol>TXD0</symbol>
14           <configured_pin>
15             <id>98</id>
16           </configured_pin>
17         </configured_role>
18       </configured_roles>
19     </configured_interface>
20     <!--...-->
21   </configured_interfaces>
22 </occupancy>
```

Listing 5: XML structure of the export format describing the hardware configurations selected by the developer.

4 GENERATION OF INITIALIZATION CODE FOR HARDWARE INTERFACES

The goal of this paper is a seamless integration of hardware accesses in UML-based MDSE environments. For this, the hardware has to be represented in an object-oriented fashion (cf. Section 4.2), while low-level code statements should be generated automatically wherever possible (cf. Section 4.3 and 4.4). For this purpose, this paper refers to a specific project structure and certain key files, which are introduced in Section 4.1. The aspects described in this section belong to the steps (A4) and (A5) of the development workflow presented in Section 2.

4.1 Project Structure

The generation process described in Section 4 makes use of several file types. This section presents an

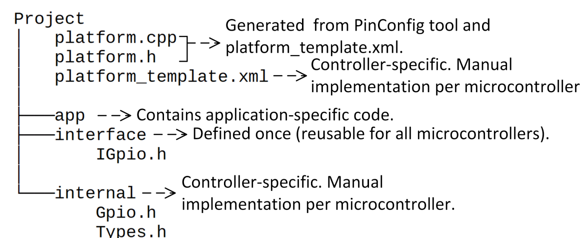

Figure 3: Overview of the different files used in Section 4.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

overview of these files (cf. Figure 3). The application itself is developed in the directory *app*. For this, an object-oriented HAL may be used to access the hardware. The interfaces of this HAL are stored in the directory *interface*. These interfaces only have to be defined once and may be reused for all microcontrollers that have been considered during the creation of the HAL. The implementation of these interfaces depends on the specific microcontroller used in the project and may be found in the directory *internal*. This implementation has to be created manually once per microcontroller. It is reusable for all projects that utilize the same microcontroller. The initialization code for the hardware interfaces is contained in *platform.h* and *platform.cpp* (cf. Listing 8 and 9 introduced in Section 4.3). These two files are generated automatically from the file *platform_template.xml* (cf. Listing 10 introduced in Section 4.4). The structure of *platform_template.xml* is similar for every microcontroller, but requires controller-specific additions that specify the exact commands available for the hardware initialization of the specific controller.

In case an existing application should be ported to another microcontroller, only two manual steps are necessary. The content of the directory *internal* has to be changed to the implementation for the new microcontroller. Furthermore, *platform_template.xml* has to be replaced by the version specific to the new microcontroller. In case the original microcontroller used to develop the application and the new microcontroller to which the application is ported provide similar capabilities, the application itself (inside the *app* directory) does not require any changes for the porting process.

4.2 Object-oriented HAL

The PinConfig tool described in Section 3 may be used to configure and automatically generate the initialization code for hardware interfaces (cf. Section 4.4 for the automatic generation). Without this, developers would have to deal with low-level code constructs in MDSE tools. However, besides the initialization code for the hardware interfaces, developers also have to interact with low-level code in order to access the hardware interfaces at runtime, e.g., to query the state of a GPIO. This section presents an object-oriented HAL that may be integrated with MDSE tools (cf. Section 5), thereby eliminating this type of low-level code interactions at runtime as well. As a complete HAL that encompasses virtually every available hardware interface is a task associated with a tremendous amount of work and resources, we choose to limit ourselves to some of the most com-

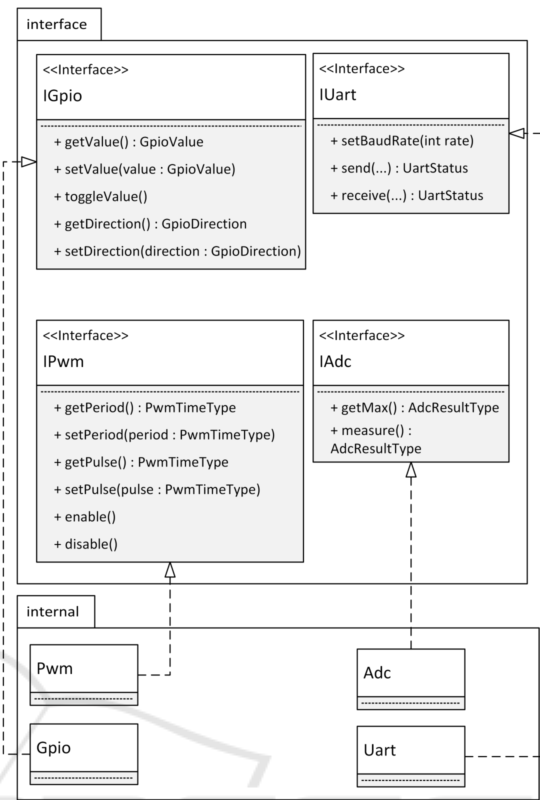

Figure 4: UML 2.5 class diagram of the structure of the HAL. Three dots (...) are used as method parameters for method signatures that are too long for the Figure. The Figure omits some template parameters (cf. Listing 7 and its description).

mon hardware interfaces in this paper, namely GPIO, ADC, UART and *Pulse Width Modulation (PWM)*. Other hardware interfaces may be integrated into the HAL following the same principles.

In order to create the HAL, we studied microcontrollers from different manufacturers concerning the previously mentioned hardware interfaces. We studied microcontrollers from the manufacturers Microchip (Microchip, 2021), Infineon (Infineon, 2021c), NXP (NXP, 2021a), ST Microelectronics (ST Microelectronics, 2021b) and Espressif (Espressif, 2021). While the usage possibilities of the hardware interfaces for the studied microcontrollers at runtime is relatively consistent (e.g., a UART may send or receive data), the configuration of these hardware interfaces often differs more strongly between the different microcontrollers. Therefore, the HAL presented in this paper only abstracts the runtime usage of the aforementioned hardware interfaces. Characteristics that are related to the initial configuration and usually remain constant during the execution of the application, still require some controller-specific code. This code is integrated via an automatic code generation

process and is explained in Section 4.4.

Figure 4 shows a UML class diagram of the hardware interfaces and the abstraction of their functionality that is used during runtime. For legibility purposes, the number of methods per hardware interface in Figure 4 is reduced to the most common functions for each hardware interface. The idea of Figure 4 is that developers implement their application with the respective interfaces from the `interface` package, thereby gaining independence from a specific hardware platform. The implementations of these interfaces exist in the package `internal`. These implementations differ for different hardware platforms. Therefore, in case an application written with the `interface` package should be ported to another platform, the `internal` package has to be replaced by another package that implements the HAL functionality for the new hardware platform. While Figure 4 shows four hardware interfaces that have been realized in the HAL presented in this paper (GPIO, ADC, PWM, UART), the remainder of Section 4 only refers to the GPIO interface for legibility purposes. Moreover, the examples refer to the Infineon Aurix TC297 (Infineon, 2021a) microcontroller. The presented concepts may be applied analogously to the other hardware interfaces shown in Figure 4.

Figure 4 shows data types specific to each hardware interface, e.g., `GpioValue` and `GpioDirection` for the interface `IGpio`. In this example, `GpioValue` indicates the current digital value of the GPIO, while `GpioDirection` indicates whether the specific GPIO is configured as an input or output. Often, microcontrollers provide their own type definitions for these values. The HAL shown in Figure 4 abstracts these type definitions for the respective microcontrollers, in order to provide a consistent API for developers regardless of the specific hardware platform on which the application should be executed. For this, a mapping between the abstracted data types and the data types specific to microcontrollers is required. In the context of this paper, this is achieved by the file *Types.h*, which contains such mappings. Listing 6 shows an excerpt of this file for the GPIO interface.

In line 5 of Listing 6, the abstract type `GpioValue` is defined as the specific type `IfxPort_State` in case the HAL is executed on an Aurix TC297 microcontroller. Line 6 and 7 of Listing 6 show another important concept contained in `Types.h`, i.e., the abstraction of constants. Constant values, e.g., for setting a GPIO to a high (`GpioHigh` in Listing 6) or low (`GpioLow` in Listing 6) voltage, may be used by developers to configure a GPIO in a designated way. Similar to the abstraction of types, as explained for `GpioValue` above, constants may be abstracted by the

```
1  #ifndef HOLMES_TYPES_H
2  #define HOLMES_TYPES_H
3  namespace holmes {
4    // Gpio
5    typedef IfxPort_State GpioValue;
6    static const GpioValue GpioLow =
         IfxPort_State_low;
7    static const GpioValue GpioHigh =
         IfxPort_State_high;
8    // [...]
9  }
10 #endif // #ifndef HOLMES_TYPES_H
```

Listing 6: Excerpt of the type definitions used in the HAL (Types.h).

HAL to increase the independence of the application code from a specific hardware platform.

Listing 7 shows an excerpt of an example implementation of the GPIO interface realization. It uses the types defined in *Types.h*. Moreover, template parameters are used to specify compile time constants that refer to hardware aspects used by the hardware interface. In Listing 7, this is the pin and the port which the GPIO should use (line 4). In Figure 4, these template parameters have been omitted to improve legibility. Besides the template parameters that specify hardware aspects, the methods provided by the `IGpio` interface have to be overridden in the interface realization. Listing 7 shows this exemplary in lines 12-14 for the method `setValue()`.

```
1  #ifndef HOLMES_INTERNAL_GPIO_H
2  #define HOLMES_INTERNAL_GPIO_H
3  namespace holmes {
4    template<uint8_t port, uint8_t pin>
5    class Gpio : public holmes::IGpio {
6    private:
7      volatile _Ifx_P* _port;
8    public:
9      Gpio() {
10       _port = (Ifx_P*) (0xF003A000u + 0
             x100u * port);
11     }
12     void setValue(GpioValue value) {
13       IfxPort_setPinState(_port, pin,
             value);
14     }
15     // [...]
16   }
17 }
18 #endif // #ifndef HOLMES_INTERNAL_GPIO_H
```

Listing 7: Excerpt of the implementation of the GPIO HAL interface for the Aurix TC297 microcontroller.

4.3 Hardware Initialization

Section 4.2 introduced an object-oriented HAL that requires hardware-specific values for the usage of an

object of the HAL. For example, the port and pin of the respective GPIO have to be passed as template parameters in Listing 7. This type of information is already configured in the PinConfig tool described in Section 3 and may therefore be generated automatically from the corresponding export format of the tool (cf. Section 3.3). Then, a set of type definitions may be used to enable developers to access a respective GPIO via an identifier configured in the PinConfig tool, thereby precluding that a developer has to know the port and pin on which the GPIO operates. These type definitions are stored in a file called *platform.h*, which may be automatically generated (cf. Section 4.4 for the generation process). An excerpt of *platform.h* is shown in Listing 8. The $ signs in Listing 8 indicate a placeholder variable that should be replaced with an actual value, i.e., the actual port on which the GPIO should operate.

```
1  #ifndef HOLMES_PLATFORM_H
2  #define HOLMES_PLATFORM_H
3  namespace holmes {
4    ${gpio_hal}
5    // [...] more type definitions
6    void init();
7  }
8  #endif // #ifndef HOLMES_PLATFORM_H
```

Listing 8: Excerpt of the type definitions that allow developers to refer to hardware interfaces with custom names (platform.h). The $ symbol indicates a placeholder variable.

In line 4 of Listing 8 a placeholder, *${gpio_hal}*, is used to represent type definitions which provide an alias for a GPIO on a specific port and pin. The number of type definitions equals the number of GPIO interfaces that have been configured in the PinConfig tool presented in Section 3. These placeholders may themselves contain more placeholders, e.g., for specifying the specific port and pin of the GPIO. Thus, *${gpio_hal}* consists of a number of statements of the following form: `typedef internal::Gpio<$port, $pin> $name;`. The values of the placeholders may be extracted automatically from the output format of the PinConfig tool described in Section 3.3. Thus, developers may specify a name for a specific GPIO they wish to use in the PinConfig tool, which may subsequently be used by the developers to instantiate an instance of the specific GPIO within their program. Besides the type definitions, *platform.h* declares the `init()` method (cf. line 6 in Listing 8), which configures the hardware interfaces according to the output format of PinConfig tool presented in Section 3.3. The definition of this method may be found in the file *platform.cpp*, which may also be automatically generated (cf. Section 4.4).

The structure of *platform.cpp* is shown in Listing 9.

```
1  ${gpio_pre}
2  static void initGpio() {
3    ${gpio_header}
4    ${gpio_body}
5    ${gpio_footer}
6  }
7  //[...] Templates other hardware interfaces
8  void holmes::init() {
9    initGpio();
10   //[...] Initialize other interfaces
11 }
```

Listing 9: Structure of the hardware initialization (platform.cpp). The $ symbol indicates a placeholder variable.

In line 8-11 of Listing 9, the aforementioned `init()` method is defined. In essence, this simply defers the initialization process to the initialization methods for each specific hardware interface. In Listing 9, such a specific initialization method is shown in lines 2-6. The placeholder ${gpio_header} is replaced by code that should only be executed once before the initialization of every GPIO. Conversely, ${gpio_footer} is replaced by code that should only be executed once after the initialization of every GPIO. The ${gpio_body} placeholder is replaced by the specific configurations for each individual GPIO. An example for this is the line `IfxPort_setPinMode((Ifx_P *)&MODULE_P${port}, ${pin}, ${mode})` for the Aurix TC297 microcontroller, which sets the port and pin of the GPIO, as well as whether it is initially configured as an input or an output. The placeholders of this line may once again be automatically extracted from the output format of the PinConfig tool presented in Section 3.3. Besides the aforementioned placeholders, line 1 of Listing 9 shows the ${gpio_pre} placeholder, which may provide additional initializations that may be accessed from within `initGPIO()`.

4.4 Code Generation for Hardware Initialization

Section 4.3 introduced the files *platform.h* and *platform.cpp*, which contain the initialization code for the hardware interfaces as configured with the PinConfig tool presented in Section 3.1. This section shows how those files are generated automatically. Listing 10 shows the structure of *platform_template.xml*.

At the start of Listing 10 the template structure of the files *platform.h* and *platform.cpp* is stored (cf. the previously introduced Listing 8 and 9 for the respective structures). The remainder of Listing 10 (cf. line 10-25) contains information about the controller-specific commands to initialize a hardware interface.

```
1  <code_generator>
2    <header_file>
3      <!-- Template holmes_platform.h
4         (cf. Listing 8)-->
5    </header_file>
6    <source_file>
7      <!-- Template holmes_platform.cpp
8         (cf. Listing 9)-->
9    </source_file>
10   <interfaces>
11     <gpio>
12       <gpio_hal>
13         typedef internal::Gpio&lt;${port},
14           ${pin}&gt; ${name};
15       </gpio_hal>
16       <gpio_pre></gpio_pre>
17       <gpio_header></gpio_header>
18       <gpio_body>
19         IfxPort_setPinMode((Ifx_P *)&amp;
               MODULE_P${port}, ${pin}, ${
               mode});
20       </gpio_body>
21       <gpio_footer></gpio_footer>
22     </gpio>
23     <!-- ... -->
24   </interfaces>
25 </code_generator>
```

Listing 10: XML file used as a template to generate the hardware initialization (platform_template.xml). The $ symbol indicates a placeholder variable. Note that some special characters have to be used because of the XML syntax, e.g., *<* to represent the symbol $<$.

For example, in Listing 9 (*platform.cpp*), the template variable *gpio_body* is used to specify the initialization process for each GPIO. In lines 18-20 of Listing 10, the controller-specific commands for this initialization process are stored. Similarly, lines 12-15 define the type definition required for the *gpio_hal* template variable that is used in Listing 8 (*platform.h*).

Generating the initialization files *platform.h* and *platform.cpp* automatically may be achieved by iterating through every hardware interface configured in the output format of the PinConfig tool (cf. Section 3.3). For each hardware interface of the same type X, e.g., GPIO, and for each sub-tag in the <gpio> tag of Listing 10 (cf. lines 12-21, e.g., <gpio_hal>), a string builder X_i is created. For example, a separate string builder X_i is created for the <gpio_hal>, <gpio_pre>, <gpio_header>, <gpio_body> and <gpio_footer> tags. The string builders X_i are appended with the content from the respective sub-tags of Listing 10. During this appending, the remaining placeholder values in Listing 10, e.g., ${pin}, are filled with the configured values from the output format of the GUI tool. Thus, once every hardware interface that is configured in the output format of the PinConfig tool has been iterated through, the string builders X_i contain the initialization information of every configured hardware interface. For example, the string builder for ${gpio_hal} contains the type definitions for all configured GPIO interfaces in the output format. The string builders X_i may then subsequently be used to replace the other placeholders for *platform.h* and *platform.cpp*, which are stored in lines 2-9 of Listing 10. Thus, the templates for *platform.h* and *platform.cpp* no longer contain any placeholders and the actual files *platform.h* and *platform.cpp* may be generated by copy-pasting the content from the templates in a newly created file.

5 INTEGRATION WITH MDSE TOOLS

The results from Section 4 allow developers to interact with hardware in an object-oriented abstract manner, as well as the automatic code generation of the initialization code for hardware interfaces. However, these elements are not yet integrated in an MDSE development process, as the results from Section 4 only exist as code. This section describes how this generated code, as well as the PinConfig tool described in Section 3.1, may be integrated with MDSE tools.

Before the actual integration process is described, we remark upon the heterogeneity of current MDSE tools. As these tools are developed by different providers, their customization options for developers differ. Thus, integration of our approach with these tools may differ slightly for each tool. However, the principle is similar for each tool. In the following, the provided examples are based on the MDSE tool IBM Rational Rhapsody (Rhapsody, 2020).

1) Starting the Hardware Configuration Tool from the GUI of the MDSE Tool. While the GUI tool presented in Section 3 may be started as a standalone application, it may also be started from an MDSE tool. Furthermore, such an integration in the MDSE tool allows for an integration of the subsequent code generation process (cf. Section 4.4) in the MDSE tool as well. In IBM Rational Rhapsody, this may be achieved by the use of so called helper files that may add entries to Rhapsody's menus and execute arbitrary Java code once such an entry is clicked.

2) Integrating the HAL Into the MDSE Tool Via Reverse Engineering. The HAL described in Section 4 exists as source code. In order to use this code in MDSE tools and enable an object-oriented access

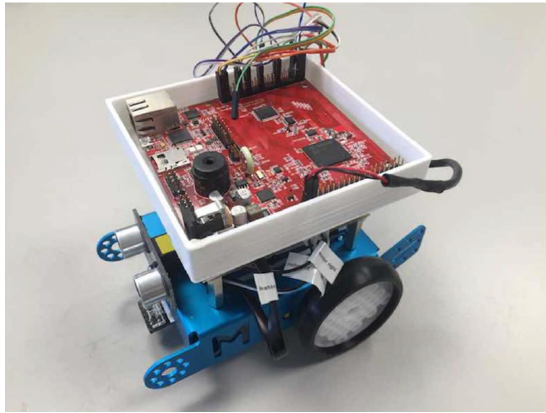

Figure 5: Picture of the modified mBot platform as used in this paper. The microcontroller on top of the mBot is an Aurix TC297.

of hardware interactions, corresponding classes either have to be created manually by the developer in the MDSE tool or created automatically via the reverse engineering functionality of the MDSE tool. In IBM Rational Rhapsody, the reverse engineering process may be executed automatically via a dedicated Java API. Thus, in our prototype implementation, this process is automatically performed after configuration of the hardware interfaces via the PinConfig tool has ended.

3) Include HAL and Platform Files During Compilation. Once the code for the application has been created from the model with the MDSE tool's code generation, the HAL and the automatically generated *platform.h* and *platform.cpp* files have to be linked during compilation.

6 USE CASE

We evaluated our approach for a small embedded application example, i.e., a toy car that is capable of autonomous driving. For this, we modify the Makeblock mBot platform (Makeblock, 2021) with a custom converter board. This converter board enables the use of 3.3V and 5V microcontrollers with the mBot, as well as the use of the GPIO, UART, ADC und PWM hardware interfaces. The GPIOs are used to control LEDs and read digital values from sensors, i.e., an ultrasonic sensor in this application. The ultrasonic sensor is used to detect obstacles in front of the mBot. The ADC is used to digitalize the luminosity levels measured by sensors below the mBot. This enables the mBot to autonomously follow a set trajectory, provided the path of the trajectory contains a different brightness level than the rest of the track. For example, this may be achieved by printing a black line that represents the path of the trajectory on white sheets of paper. The PWM is used to control the motors of the mBot, while the UART enables interaction with a bluetooth chip. The bluetooth chip, in turn, is capable of communicating with a smartphone application that may be used to 1) control the mBot remotely, 2) start its autonomous trajectory following driving mode and 3) enable the automatic obstacle avoidance enabled through the ultrasonic sensors. Figure 5 shows an image of the modified hardware platform with the Aurix TC297 microcontroller connected.

For the purpose of this use case a software application was created with an MDSE tool (IBM Rational Rhapsody (Rhapsody, 2020)) that provides the application logic for the functionality described above. For hardware interactions, the HAL described in Section 4 is used within the application model. In order to show the effect of our approach on the portability of applications, we selected five microcontrollers for which the application should control the mBot. These are a LPC1768 (NXP, 2021a), XMC4500 (Infineon, 2021c), STM32F4 (ST Microelectronics, 2021b) and an Aurix TC279 (Infineon, 2021a). With the HAL implementations for these microcontrollers, as described in Section 4, the only step required for running the application on these different microcontrollers is the configuration of the hardware via the PinConfig tool described in Section 3. By employing the code generation described in Section 4.4, the remaining controller-specific code is generated automatically. Thus, porting the application comes down to specifying the hardware initialization for the new microcontroller in the PinConfig tool.

7 RELATED WORK

This section describes related work to the approach presented in this paper. This encompasses approaches that describe a HAL, as well as approaches that generate some form of initialization code for the hardware of microcontrollers.

Industrial tools that are capable of generating initialization code for the hardware of microcontrollers include, e.g., (Infineon, 2021b; ST Microelectronics, 2021a; NXP, 2021b) . Most often, these tool are proprietary and are developed by a specific manufacturer of microcontrollers. In consequence, the tools only support the configuration of microcontrollers for a specific family of microcontroller, i.e., those offered by the specific manufacturer. For example, (ST Microelectronics, 2021a) is limited to the configuration

of microcontrollers from ST Microelectronics. Our approach, in contrast, provides a generic description format for microcontrollers and their hardware interfaces (cf. Section 3) and may thus include microcontrollers from different manufacturers (cf. Section 6). Furthermore, these tools do not consider the integration of the generated source code into MDSE tools at all. This is exacerbated by the fact that most of these tools do not create object-oriented source code. Non-object-oriented source code is hard to integrate with UML-based MDSE tools, as UML is an object-oriented modeling language and supports non-object-oriented concepts only in a limited fashion. Our approach, in contrast, couples the hardware initialization to an object-oriented HAL (cf. Section 4) and may therefore be easily integrated into MDSE tools (cf. Section 5).

Besides these industrial approaches, there are several academic approaches for generating the initialization code for the hardware interfaces of microcontrollers. In (Yunfei Bai et al., 2007; Bhanu et al., 2009), an XML-based description methodology for microcontrollers is described. It is intended for the code generation of initialization code. However, their description assumes direct hardware access for each microcontroller, whereas our approach assumes the usage of a HAL. Furthermore, their approach does not generate object-oriented code and thus is hard to integrate in MDSE tools. Nevertheless, their description format served as an inspiration for our XML-based description formats described in Section 3.

An approach that is limited to the generation of source code for the MPC5644A microcontroller has been presented in (Geng et al., 2012). Our approach, in contrast, is extensible by design and may generate code for any microcontroller once the required XML-files (cf. Section 3 and 4) are created. Furthermore, their approach generates code for Matlab/Simulink (MathWorks, 2021), while our approach focuses on integration with UML-based MDSE tools.

Another approach for platform-independent source code generation has been described in (Kim et al., 2013). The platform-dependent characteristics of the target platform are described with an Architectural Analysis Description Language (AADL), which is supplemented by a code snippet repository. Our approach, in comparison, uses an XML-based format to describe the target microcontrollers and relies on a HAL for the implementation of hardware accesses instead of a code snippet repository. Due to the use of an object-oriented HAL and UML, our approach enables developers to work with UML-based MDSE tools to develop their application. The approach presented in (Kim et al., 2013), in contrast, does not describe an integration with UML-based MDSE tools and instead relies on other modeling languages, e.g., Stateflow (MathWorks, 2021) or UPPAAL (Behrmann et al., 2004).

Section 4 introduces an approach for the seamless integration of object-oriented HALs into MDSE. There exist other, non object-oriented HALs, e.g., (ARM Limited, 2021a; ARM Limited, 2021b). Due to their non-object-oriented nature, they are harder to integrate with MDSE tools. The number of object-oriented HALs is relatively small, e.g., (modm., 2021). Currently, these approaches do not describe how they may be seamlessly integrated with MDSE tools.

8 CONCLUSION

MDSE tools are often based on UML and its high-level object-oriented concepts. The interaction with hardware, which usually occurs through low-level program code at the register level, often requires a change in abstraction levels for the developer. Moreover, due to the hardware interactions not being object-oriented, MDSE tool support for such hardware interactions is lacking. This paper proposes an approach how object-oriented HALs may be seamlessly integrated into MDSE tools to enable developers to develop embedded systems in a holistic, object-oriented fashion. For this, we introduce a GUI tool to configure hardware interfaces and provide an automatic code generation approach that uses XML-based template files. The resulting code may be automatically imported into MDSE tools. We evaluated our approach by developing an application example for microcontrollers from three different manufacturers (NXP, Infineon and ST Microelectronics). Due to the use of our HAL, the application code could be ported to the other microcontrollers with a minimum of modifications.

Future work includes an automated generation of the XML files describing the structure of microcontrollers for the GUI tool in which the hardware is configured. Furthermore, we envision a SysML (Object Management Group, 2019)-like frontend for the GUI tool that enables developers to specify the hardware configuration without leaving their MDSE tool.

ACKNOWLEDGEMENTS

This work was partially funded by the German Federal Ministry of Economics and Technology (Bun-

desministeriums fuer Wirtschaft und Technologie-BMWi) within the project "Holistic model-driven development for embedded systems in consideration of diverse hardware architectures" (HolMES). The authors would like to thank Adrian Richter for software development assistance, as well as Johannes Trageser and Lars Donner for helpful comments on the HAL and its integration in MDSE tools.

REFERENCES

ARM Limited (2021a). Cortex Microcontroller Software Interface Standard (CMSIS) https://developer.arm.com/tools-and-software/embedded/cmsis (accessed on 25th March 2021).

ARM Limited (2021b). Mbed OS. https://os.mbed.com/mbed-os/ (accessed on 11th March 2021).

Behrmann, G., David, A., and Larsen, K. G. (2004). *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bhanu, G. P., Bai, Y., Tan, S. L., and Chng, E. S. (2009). A Generic MCU Description Methodology with Dependency Evaluation. In *2009 International Conference on Signal Processing Systems*, pages 565–569.

Brambilla, M., Cabot, J., and Wimmer, M. (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers.

Bunse, C., Gross, H.-G., and Peper, C. (2007). Applying a model-based approach for embedded system development. pages 121–128.

Charette, R. N. (2009). This car runs on code. *IEEE spectrum*, 46(3):3.

Espressif (2021). ESP32. http://esp32.net/ (accessed on 18th March 2021).

Geng, P., Ouyang, M., Li, J., and Xu, L. (2012). Embedded C Code Generation Platform for Electric Vehicle Controller. *Advanced Materials Research*, 546-547:778–783.

Huning, L., Iyenghar, P., and Pulvermueller, E. (2020). A workflow for automatically generating application-level safety mechanisms from UML stereotype model representations. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE*, pages 216–228. INSTICC, SciTePress.

Infineon (2021a). 32-bit AURIX™ Microcontroller based on TriCore™. https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/(accessed: 13th January 2021).

Infineon (2021b). Dave. https://www.infineon.com/dgdl/infineon-dave_introduction-dt-v01_00-en.pdf?fileid=5546d462636cc8fb01645f681d4713ed (accessed on 11th March 2021).

Infineon (2021c). XMC4500. https://www.infineon.com/cms/de/product/microcontroller/32-bit-industrial-microcontroller-based-on-arm-cortex-m/32-bit-xmc4000-industrial-microcontroller-arm-cortex-m4/xmc4500/ (accessed on 19th March 2021).

Kashif, H., Mostafa, M., Shokry, H., and Hammad, S. (2009). Model-based embedded software development flow. In *2009 4th International Design and Test Workshop (IDT)*, pages 1–4.

Kim, B., Phan, L. T. X., Sokolsky, O., and Lee, L. (2013). Platform-dependent code generation for embedded real-time software. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10.

Makeblock (2021). mBot. https://www.makeblock.com/mbot (accessed on 9th March 2021).

MathWorks (2021). Matlab. https://www.mathworks.com/products/matlab.html (accessed on 18th March 2021).

Microchip (2021). ATMegaA328-PU. https://www.microchip.com/wwwproducts/en/ATmega328P (accessed on 19th March 2021).

modm. (2021). modm: a barebone embedded library generator. https://modm.io/ (accessed on 16th March 2021).

NXP (2021a). https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1700-cortex-m3/512kb-flash-64kb-sram-ethernet-usb-lqfp100-package:lpc1768fbd100 (accessed on 18th March 2021).

NXP (2021b). MCUXpresso. https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:mcuxpresso-ide (accessed on 11th March 2021).

Object Management Group (2019). OMG Systems Modeling Language Version 1.6. Technical report, Object Management Group.

OMG UML (2017). OMG Unified Modeling Language Version 2.5.1. Technical report, Object Management Group.

Rhapsody (2020). IBM. Rational Rhapsody Developer. https://www.ibm.com/us-en/marketplace/uml-tools (accessed 20th August 2020).

Selic, B. (2008). Personal reflections on automation, programming culture, and model-based software engineering. *Automated Software Engineering*, 15(3-4):379–391.

ST Microelectronics (2021a). STM32CubeMX. https://www.st.com/en/development-tools/stm32cubemx.html (accessed on 11th March 2021).

ST Microelectronics (2021b). STM32F4XX. https://www.st.com/en/microcontrollers-microprocessors/stm32f4-series.html (accessed on 19th March 2021).

Trindade, R., Bulwahn, L., and Ainhauser, C. (2014). Automatically generated safety mechanisms from semi-formal software safety requirements. In Bondavalli, A. and Di Giandomenico, F., editors, *Computer Safety, Reliability, and Security*, pages 278–293, Cham. Springer International Publishing.

Yunfei Bai, Eng Siong Chng, and Gorthi Prashant Bhanu (2007). An mcu description methodology for initialization code generation software. In *2007 International Conference on Parallel and Distributed Systems*, pages 1–7.