Abstract—Nowadays, the increasing complexity of the embedded software development process, demands design techniques capable of addressing such complexity efficiently. In this article, a methodology that follows MDA guidelines for the design of real-time embedded software is presented. This methodology comprises two fundamental activities: application modeling and code generation. Our modeling strategy uses UML and MARTE extensions to elaborate models where the specification of the application functionality is decoupled from the platform execution support. This modeling approach is complemented by a code generation strategy that transforms the application model into efficient C code for execution on embedded systems. A sender-receiver application is used as a case study to illustrate the complete methodology workflow.

Keywords: UML, MARTE, SRM, code generation, embedded systems, FreeRTOS.

1. Introduction

Due to the constantly increasing complexity of microprocessing ICs, the code-centric approach to design embedded, real-time, software is no longer an effective task. Given that about 80% of embedded systems development cost is attributed to software aspects of design [1], the demand for methods and design techniques that make it possible to address the complexity of the software design process is growing every day.

MDA (Model Driven Architecture) [2] is a design vision, proposed by the Object Management Group (OMG), that promises to reduce time and effort to develop portable and high quality software for execution platforms such as real time embedded systems. Modeling, aids in understanding software application functionality through the mechanism of abstraction, while the code that implements such functionality, in a specific platform, is automatically obtained “generated” from the models. Nonetheless, an effective approach to the practice of MDA is only possible with the support of tools that automatically transform the model into the application code. In MDA, models are constructed using the Unified Modeling Language (UML). Since UML was conceived as a general purpose modeling language, it lacks the expressive power to address the Real Time Embedded Systems (RTES) domain. This shortcoming was alleviated with the standardization of the profile for Modeling and Analysis of Real Time and Embedded systems (MARTE) [3]. SRM (Software Resource Modeling) is a profile contained within MARTE, that enables the description of real time operating system and framework APIs via UML model libraries. The purpose of the article is to illustrate a methodology for embedded software application modeling in sensor monitoring and control applications, using MDA and SRM guidelines, as well as our proposed code generation strategy by which C language code for a real time framework is obtained.

This paper is organized as follows: in section 2 previous work in this area of study is overviewed; section 3 introduces the SRM sub profile of MARTE; in section 4 the modeling and code generation strategies are described; section 5 presents the case study example and its code generation results and, finally, the article is concluded in section 6.

2. Related Work

Adopting the MDA design vision promises some advantages. One, is minimizing errors and coding effort to implement the application thanks to automatic code generation [4]. Nevertheless, research is needed to improve the results of the automatic generation process given that “source code generation is an immature technique that is either very restrained or very unoptimized” [5].

Two main approaches are identified in the literature for code generation in a model based design context: visitor based and template based code generation [6]. Templates are code fragments composed of static text (code) and parameterized text. The parameterized text is later replaced by expressions in the target language resulting in a source code file. It is remarked in [6] that the template based technique has advantages over the visitor based generation, since templates resemble closely the code to be generated and, besides, they are understandable and easy to design.

A generalized technique that uses template based code generation for embedded application design, is described in [7], [5] and [8]: since UML models are stored in a textual, standard, format, denominated XMI (XML Metadata Interchange) in order to be interchangeable between modeling tools, this technique uses an XMI to XML mapping through XSLT transformations in order to extract key information

1. XSLT (Extensible Style sheet Language Transformations): It is an XML based language that enables transformations of this kind of documents to other formats such as different XML schema, HTML and others.
from the UML models (e.g., variable names, class names, MARTE stereotypes and others). Then, the resulting XML file is used to build a tree structure, more precisely a Document Object Model (DOM), which is later used by a template engine (e.g., Freemaker) to generate the code according to pre-established templates for a given language (e.g., C++/SystemC, Java, etc). “Unfortunately, using XMI and XSLT has scalability limitations. Manual implementation of model transformations in XSLT quickly leads to non maintainable implementations because of the verbosity and poor readability of XMI and XSLT” [6].

Code generation frameworks that do not use XSLT are available, most notably Acceleo [9], JET [10] and Xpand [11] which are all part of the eclipse modeling tools ecosystem. Particularly, Acceleo is an implementation of the MOF Model to Text transformation language standard issued by OMG [12] and UML 2.x models built with various modeling tools are compatible with it. Code generators have been developed with Acceleo for the generation of Modelica code [13] from ModelicaML (UML profile for Modelica) and C++/SystemC from UML/MARTE models [14], however MARTE::SRM utilization in the models and C code generation for embedded prototypes using this framework is scarcely explored.

Commercial tools that provide support for MARTE and code generation for embedded systems are Rational Rhapsody Developer [15] and Artisan Studio [16] but these are closed source and are not available for most of researchers due to its high cost.

Compared to prior efforts and tools available, our work specifically aims to enable the UML modeling of RTES applications and its automatic transformation into C language source code according to MARTE::SRM and MDA guidelines. Moreover, since standard UML 2.x features and MARTE stereotypes are used, any tools that support those standards can be used to construct the models. Full code generation, for deployment on real embedded prototypes is achieved using eclipse integrated tools such as Papyrus [17] for modeling and Acceleo which implements the MOF Model to text transformation (code generation) standard.

3. SRM Overview

SRM is a sub profile of MARTE that provides facilities for building UML model libraries that describe software execution platforms (e.g., Real Time Operating Systems, RTOS, or real-time frameworks) APIs in a standard way. It provides standard stereotypes for addressing concerns such as concurrent execution contexts (e.g., tasks, interrupts), interaction between concurrent application components for communication or synchronization (e.g., mutexes, queues, semaphores, etc.) and hardware/software resource intermedation (e.g., driver or memory management) [3].

The SRM profile can be used in processes where platform modeling is important (e.g., the MDA Y-Chart) and it covers RTOS concerns with low level of details to enable generative approaches (code generation) from the models [18]. A complete review of the SRM profile is beyond the scope of this article. A thorough description is presented in [18] and [3].

4. Methodology

This embedded software application design workflow is based on the MDA Y-Chart process in which “a platform independent model of the software (PIM) is transformed into a platform specific model (PSM); given a platform description model (PDM)” [18]. In this scheme, the SRM sub profile of MARTE is used in the construction of the execution software platform model (PDM) in order to describe its resources and services. The automatically generated code, obtained from an application model is going to be executed in hardware platforms with restricted computational resources, i.e., with a few kilobytes of RAM and FLASH and limited processing power. For this reason, the target language chosen for the code generation was C, and the selected software execution platform was the FreeRTOS [19], which is a real-time and lightweight micro-kernel also written in C.

4.1 Modeling Strategy

In the rest of the article the PIM will be referred as the Analysis Model, the PDM as the Platform Model and the PSM as the Specific Application Model. The following subsections illustrate our approach for the construction of the aforementioned models (Analysis, Platform and Application models). Papyrus [17] is used as the tool for model construction, given its support for the MARTE stereotypes and integration with the Eclipse modeling tools ecosystem.

4.1.1 The Analysis Model

This model contains the functional specification of the embedded software application. It is designed to be independent of the platform2, i.e., it can be ported between platforms without change [18]. In our case, the language chosen to specify the actions in the model was the C language (ANSI C), so this model remains independent among the platforms that support such language [4].

This model uses different but complementary UML views or diagrams. The structural view provides information of the elements conforming an application and specifies the relationships between them. The dynamic view defines the behavior and interaction between structural components. In our approach, the class diagram is used to specify attributes and operations for the structural entities of the application, while the dynamic aspect of the model is specified via the activity diagram which is used to define the body of class

2A platform is a set of technological resources that provide a specific functionality. Any application supported by the platform can make use of such functionality regardless of how it is implemented [2].
operations (methods) and the state machine diagram is used to model the behavior of reactive classes.

![State Machine Diagram](image)

**Fig. 1: UML diagrams used in the Analysis Model**

Figure 1a depicts an example class diagram. Empty classes mean that its code is available (legacy code), otherwise their attributes and operations should be specified as well. Usage relationships handle class dependencies with external (existing) libraries or data types. Alternatively, common association relationships between classes can be used. Class operation behavior is defined by means of an associated activity diagram to the operation (Figure 1b shows an example of an activity diagram). If the class has a reactive behavior it should be specified in an associated state machine diagram to the class. Figure 1c shows an example of a state machine diagram. In state machines, transitions between states are triggered by event occurrences. Entry or exit actions can be defined for a state, i.e., actions that execute at the moment of entering or exiting the state.

### 4.1.2 The Platform Model

This is the model library where all platform resources and APIs necessary to construct the application are defined [20]. In our proposed approach a decision was made to design a framework that better matches the application domain (sensor monitoring applications). Lee [21] defines a framework as a set of constraints on components of the execution platform such that a set of benefits result from those constraints. The framework designed is based on the active object design pattern which combines the benefits of preemptive multitasking operating systems and the event driven programming paradigm [22]. “Active objects are nothing more than individual tasks with their own event queues” [22]. The designed framework has the following concurrent components: reactive tasks, interrupts, and algorithmic tasks. The reactive task (a.k.a active object, i.e., it owns an event queue) is where state machine, reactive, behavior executes. Events, if present, are extracted from the queue in order of arrival (FIFO) and dispatched to the state machine, otherwise the reactive task enters a blocking state. There is also an algorithmic task resource in which periodic, real-time, behavior can be scheduled for execution. Algorithmic tasks and hardware interrupts can send asynchronous messages (events) to reactive tasks, and reactive tasks can communicate between them in a similar fashion, but not with algorithmic tasks or interrupts given that they do not own a queue for the reception of event messages. The benefits of constraining the software platform with the framework is improving application concurrency and simplifying synchronization complexity among tasks by using asynchronous message passing (events) instead of semaphores and mutexes, besides it also simplifies the process of code generation. A selection of SRM stereotypes was made in order to properly characterize the framework’s resources. Table 1 shows the selected stereotypes and their semantic.

### Table 1: Selected SRM stereotypes

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>swSchedulableResource</td>
<td>encapsulated sequences of actions which execute concurrently.</td>
</tr>
<tr>
<td>messageComResource</td>
<td>communication resource used to exchange messages.</td>
</tr>
<tr>
<td>interruptResource</td>
<td>computing context to execute user delivered routines.</td>
</tr>
<tr>
<td>EntryPoint</td>
<td>supplies the routine executed in the context of a concurrent resource.</td>
</tr>
</tbody>
</table>

Figure 2, depicts the UML model of the framework created using the SRM profile and the FreeRTOS API. The SRM stereotypes «swSchedulableResource», «messageComResource» and «interruptResource», are used to denote, respectively, concurrent, communication and interrupt resources of the software, FreeRTOS based, platform.

### 4.1.3 The Specific Application Model

In this model, the mapping of the functionality onto the platform takes place. First, the Platform Model needs to be imported, then the Specific Application model is constructed by instantiating and initializing the resources defined in the Platform Model. Also, the binding of the Application with the Analysis Model is realized by connecting instances (objects) of the Platform Model with instances of the Analysis Model by means of «EntryPoint» stereotyped dependency relationships [20]. The later step, specifies which function behavior (routine) from an object defined in the Analysis Model is going to be executed in a concurrent resource of the framework, i.e., reactive or algorithmic task. Figure 3 illustrates the process of linking Analysis and Application models using the «EntryPoint» stereotype.
Fig. 2: Platform Model (Tagged Values not shown due to space limitations).

Fig. 3: Binding the Specific Application Model (bottom) with the Analysis Model.

4.2 Code Generation Strategy

The code generation strategy follows a template based approach using Acceleo. All models are transformed into code by executing a chain of Acceleo template scripts.

Algorithm 1 shows the pseudocode for the transformation of the Analysis Model into C code. The GenerateStructure template receives the Analysis Model as input parameter. Then, there is an iteration over all classes that are present inside the model and if attributes or associations are defined for that class, the GenerateHeader template is called with that class as an argument. Also, if class operations are defined the GenerateImplementation template is invoked in order to create the corresponding C file implementation of that class.

The GenImplementation template generates the inclusion of the corresponding header files and the implementation for class owned operations (methods). The methods of a class, are generated by transforming a subset of the associated activity diagram into a sequence of statements in the C language. For the classes that have state machine defined behavior, a script is used to transform the associated state machine diagram into C code conforming to the finite UML state machine implementation proposed by Samek [23].

The Specific Application Model transformation, where the application main file is generated is illustrated by Algorithm 2. First, a file named “main.c” is opened for writing, then all application dependencies are included by calling the GenIncludes template, which selects the instantiated objects belonging to the Analysis Model and includes their header files. Then, an iteration takes place over all instances defined in the Application Model in order to ask if its classi-
5. Case Study Application

The application used to illustrate the modeling and code generation strategies consists of two counter and two receiver concurrent objects. The counting objects keep an internal count with a different time resolution (10 ms and 100 ms). Both of them send messages (sender ID and count value) to receiver objects.

Figure 5 shows the Analysis Model class diagram for the case study application. Attributes and operations are defined for the Counter and Receiver classes. task_intercomm and serial_usb classes are empty which means that its code is available. Note, that both classes are connected through a usage relationship with the CountEvt data type. This means that this type definition is known by both classes. CountEvt represents the event message that a Counter object will send to a Receiver object at specific count values, this behavior is specified in the Counter_Algorithm routine. The Counter class uses macros defined in the task_intercomm class to send the events. The Receiver, that has a reactive behavior specified by means of the associated state machine diagram illustrated by Figure 4, changes its state every time the received count value matches either 100 (the EARLY_COUNT_SIG signal is emitted), 150 (MID_COUNT_SIG) or 250 (LATE_COUNT_SIG). An entry action defined for every state sends debug information (Sender_ID and Receiver_ID) through the USB port using the serial_usb driver.

The Platform Model (see Figure 2) is imported by the application model in order to instantiate the resources of the framework. Then, as illustrated by Figure 6, two Counter and two Receiver objects are instantiated in the Analysis Model and four tasks are instantiated in the Application Model; two reactive tasks and two algorithmic tasks. The Receiver_Dispatch routine (this routine dispatches received events to state machines) is linked for execution inside reactive tasks and the Receiver_Algorithm routine, that generates and sends count events to reactive tasks, is linked with both algorithmic tasks.

Note, that inside the main function, the creation of tasks and queues takes place, this is done by GenQueueCreation and GenTaskCreation, this templates extract the initialization values from the application model instances and generate the C statements that realize this task with the FreeRTOS APIs. Reactive and algorithmic task bodies are generated by GenAlgTaskBody and GenReactiveTaskBody respectively. Finally, any interrupts instantiated in the Application Model are generated by GenInterrupts.

Since the C language is not object oriented by design, the resulting code from the generation process was designed in an object oriented fashion, in the sense that every method of a class receives as input parameter a pointer to a structure containing all the attributes defined, for that specific class, in the class diagram, which results in a natural mapping from the UML diagrams to the code.
5.1 Results

The generated code was compiled with the Code Composer Studio v4.2 compiler for the MSP430F5438A microcontroller from Texas Instruments, with no optimizations. The correct execution of the generated application was tested on the TI MSP-EXP430F5438 evaluation board. Figure 7, illustrates portions of the generated code, particularly, the main function, the implementation of algorithmic and reactive tasks and the reception of messages in a PC terminal.

The FLASH memory consumption obtained from analyzing the “.map” file, generated by the linker, reveals that the complete application takes 8414 bytes of which 1338 bytes correspond to the generated code and libraries (drivers) that provide application support while the remaining 7076 bytes correspond to the FreeRTOS execution support and msp430 support libraries. RAM consumption is 1550 bytes total of which 59 bytes are used by the main application file, 1026 bytes represent the user configured amount of heap and the rest is used by the FreeRTOS execution support.

6. Conclusions and Future Work

In this paper a methodology workflow for UML modeling of embedded systems applications, using SRM and MDA guidelines, was illustrated. Also, the subsequent process of automatic model transformation (code generation) into compilable C code was demonstrated. Both processes, application modeling and code generation, were carried out with open source tools, Papyrus and Acceleo, which are part of the eclipse modeling tools.

The main advantage of including the SRM profile in the design methodology is that execution platforms can be easily described independently of application functionality and a stereotype guided strategy for code generation can be implemented.

A framework was designed in order to restrict the platform model to the active object model of computation, this decision simplifies synchronization complexity of concurrent components of the framework but the application domain is limited to event oriented, reactive embedded applications such as sensor monitoring and control or sensor network systems. The memory footprint of the generated code is appropriate for execution in memory constrained microcontrollers generally used in sensor monitoring applications or wireless sensor nodes.

Future research involves the study of debugging strategies that can be incorporated into the methodology workflow,
using the sequence diagram, in order to verify, in real-time the correctness of the generated application. One possible alternative is to generate instrumented code that can reproduce a sequence diagram by sending information of the application execution status to a PC application.

References