ModES: Embedded Systems Design Methodology and Tools based on MDE

Francisco Assis M. do Nascimento, Márcio F. S. Oliveira, Flávio Rech Wagner
UFRGS, Instituto de Informática, Porto Alegre, Brazil
{fanascimento,mfsoliveira,flavio}@inf.ufrgs.br

Abstract

MDE-based approaches have been proposed as a solution to cope with the inefficiency of current design methods. In this context, our work defines meta-models for application, capturing functionality by means of processes communicating by ports and channels; platform, indicating available hardware/software resources; mapping from application into platform; and implementation, oriented to code generation and hardware synthesis. Differently from other approaches, our mapping meta-model does not specify only the allocation of application processes into fixed hardware components. Instead, it delimits a design space, determining all possible associations between processes and platform components. Transformations between meta-models implement the mapping, allowing the evaluation of possible implementations during a design space exploration process. This exploration is based on accurate estimates of the effects of each transformation, without requiring costly hardware synthesis and code generation steps. These are performed only for the final implementation, after all design decisions have been taken. We implemented our approach using the Eclipse Modeling Framework (EMF) features, where our meta-models are captured by Ecore models and the transformation engine uses the MDDi-QVT plug-in. The application on a real case study validates the approach.

1. Introduction

To cope with the inefficiency of the current design methods, many research efforts in the academy and industry are concentrated on the development of new methodologies, which should be able to deal with the growing complexity of the embedded systems design process. Approaches based on MDE (Model Driven Engineering) have been proposed as an efficient methodology for embedded systems design [1][2][12][19].

One of the approaches to MDE is MDA (Model Driven Architecture) [9], which is a framework proposed by OMG for software development, driven by models at different abstraction levels. In an MDA approach, a system is modeled using a platform-independent model (PIM), which is transformed into a platform-specific model (PSM), given a platform model (PM). The languages used to express these models are defined by means of meta-models that are able to define the abstract and concrete syntax as well as the operational semantics of the modeling languages.

Our work proposes an MDE-based approach to embedded system design and defines meta-models to represent applications, capturing functionality by means of processes communicating by ports and channels; platforms, indicating available hardware/software resources; mappings from application into platform; and implementations, oriented to code generation and hardware synthesis. Meta-models and an API to handle meta-model instances compose the MDE infrastructure, which allows the integration of several model-based design tools. Currently, the ModES (Model-driven Design of Embedded Systems) framework contains the proposed MDE infrastructure and integrates tools to perform model-based estimation [12], design space exploration [17], code generation [5], and hardware synthesis [7].

Differently from other approaches, our mapping meta-model does not specify only the allocation of application processes into fixed hardware components. Instead, it delimits a design space, which corresponds to all possible implementations that can be obtained through the choice of sequences of transformations between models.

Therefore, the set of transformations between models implements the possible mappings from application into platform. It allows the evaluation of possible implementations during a design space exploration process that is based on accurate estimates of each model transformation, without requiring costly hardware and software synthesis/simulation steps. These are performed only for the final implementation, after all design decisions have been taken.
The remaining of this paper is organized as follows. Section 2 presents our design methodology, describing in Section 2.1 the proposed meta-modeling infrastructure, consisting of the application, platform, mapping, and implementation meta-models, and in Section 2.2 the model-based design tools. Section 3 describes a real case study, which illustrates and validates our embedded systems design approach. A comparison of our methodology with other MDE-based approaches is given in Section 4. Section 5 draws main conclusions and proposes future research directions.

2. ModES: Model-driven design approach

ModES (Model-driven Design of Embedded Systems) is a methodology and a corresponding set of tools that implement our MDE approach. The ModES methodology includes a meta-modeling infrastructure and tools to support model-based design tasks such as design space exploration, estimation, code generation, and hardware synthesis.

2.1. Meta-modeling infrastructure

As shown in Figure 1, in our design flow the application is specified independently from the platform. A mapping defines how application functions are partitioned among architectural components. Accordingly, four internal meta-models have been proposed in order to allow the independent application and platform modeling.

2.1.1 Internal Application Meta-model (IAMM). In order to represent an application in a standard way, a model that is captured according to some application meta-model (e.g., Simulink or SystemC meta-models) is translated into a common application model defined by the Internal Application Meta-model (IAMM) (shown in Figure 2). Thus, from different application modeling formalisms, a uniform model with a well-defined syntax and operational semantics is obtained.

2.1.2. Internal Platform Meta-model (IPMM). Usually, different Platform Meta-models (PMM) are used in the current version of ModES, we use the Eclipse Modeling Framework (EMF) [5] features to implement the meta-modeling infrastructure. We have Ecore representations for our meta-models, and a designer can use the UML2 editor plug-in to create applications, platforms, and mappings models, which are also stored in an Ecore-based repository. This Ecore repository also stores the implementation model that is generated by the transformation engine during the design process.

In the following, we describe the meta-models shown in Figure 1.

Figure 1. Meta-modeling infrastructure

The MDE approach is used to generate the internal representations, by performing model-based transformations using QVT (Query, Views, and Transformations) resources, standardized by OMG [13].

In the current version of ModES, we use the Eclipse Modeling Framework (EMF) [5] features to implement the meta-modeling infrastructure. We have Ecore representations for our meta-models, and a designer can use the UML2 editor plug-in to create applications, platforms, and mappings models, which are also stored in an Ecore-based repository. This Ecore repository also stores the implementation model that is generated by the transformation engine during the design process.

2.1.1 Internal Application Meta-model (IAMM). In order to represent an application in a standard way, a model that is captured according to some application meta-model (e.g., Simulink or SystemC meta-models) is translated into a common application model defined by the Internal Application Meta-model (IAMM) (shown in Figure 2). Thus, from different application modeling formalisms, a uniform model with a well-defined syntax and operational semantics is obtained. This translation corresponds to the traditional steps of a compiler, but it is implemented in ModES by means of transformations between models.

Using the IAMM, a system specification captures the functionality of an application in terms of a set of modules that are composed of concurrent communicating processes and/or sub-modules. The module behavior is captured in terms of actions represented by a Control Data Flow Graph (CDFG) that captures the data and control flow between the actions. The CDFG corresponds to the UML actions of the scenarios (sequence diagrams) that are related to the process, according to its active object. Therefore, our Internal Application Meta-model captures structural aspects of the Application Model by using a hierarchy of modules and processes, as well as behavioral aspects by means of the CDFGs. These concepts of modules, intercommunicating processes, and CDFGs, adopted by the IAMM, correspond to CSP-like languages, which are able to express any kind of concurrent, distributed system.

2.1.2. Internal Platform Meta-model (IPMM). Usually, different Platform Meta-models (PMM) are used in the current version of ModES, we use the Eclipse Modeling Framework (EMF) [5] features to implement the meta-modeling infrastructure. We have Ecore representations for our meta-models, and a designer can use the UML2 editor plug-in to create applications, platforms, and mappings models, which are also stored in an Ecore-based repository. This Ecore repository also stores the implementation model that is generated by the transformation engine during the design process.

In the following, we describe the meta-models shown in Figure 1.

Figure 2. Internal Application Meta-model

Using the IAMM, a system specification captures the functionality of an application in terms of a set of modules that are composed of concurrent communicating processes and/or sub-modules. The module behavior is captured in terms of actions represented by a Control Data Flow Graph (CDFG) that captures the data and control flow between the actions. The CDFG corresponds to the UML actions of the scenarios (sequence diagrams) that are related to the process, according to its active object. Therefore, our Internal Application Meta-model captures structural aspects of the Application Model by using a hierarchy of modules and processes, as well as behavioral aspects by means of the CDFGs. These concepts of modules, intercommunicating processes, and CDFGs, adopted by the IAMM, correspond to CSP-like languages, which are able to express any kind of concurrent, distributed system.

2.1.2. Internal Platform Meta-model (IPMM). Usually, different Platform Meta-models (PMM) are used in the current version of ModES, we use the Eclipse Modeling Framework (EMF) [5] features to implement the meta-modeling infrastructure. We have Ecore representations for our meta-models, and a designer can use the UML2 editor plug-in to create applications, platforms, and mappings models, which are also stored in an Ecore-based repository. This Ecore repository also stores the implementation model that is generated by the transformation engine during the design process.

In the following, we describe the meta-models shown in Figure 1.

Figure 1. Meta-modeling infrastructure

The MDE approach is used to generate the internal representations, by performing model-based transformations using QVT (Query, Views, and Transformations) resources, standardized by OMG [13].

In the current version of ModES, we use the Eclipse Modeling Framework (EMF) [5] features to implement the meta-modeling infrastructure. We have Ecore representations for our meta-models, and a designer can use the UML2 editor plug-in to create applications, platforms, and mappings models, which are also stored in an Ecore-based repository. This Ecore repository also stores the implementation model that is generated by the transformation engine during the design process.

In the following, we describe the meta-models shown in Figure 1.

2.1.1 Internal Application Meta-model (IAMM). In order to represent an application in a standard way, a model that is captured according to some application meta-model (e.g., Simulink or SystemC meta-models) is translated into a common application model defined by the Internal Application Meta-model (IAMM) (shown in Figure 2). Thus, from different application modeling formalisms, a uniform model with a well-defined syntax and operational semantics is obtained. This translation corresponds to the traditional steps of a compiler, but it is implemented in ModES by means of transformations between models.
required for the specification of platform models, corresponding to the platform modeling languages that are commonly employed in the description of system architectures, such as SystemC and VHDL. As for application models, in order to have a standard representation, the PMM is translated into an Internal Platform Meta-Model (IPMM), shown in Figure 3.

In our methodology, the available hardware/software components and the meta-information about them are stored in a platform repository, whose meta-model is based on the General Resource Model from UML-SPT [13].

![Figure 3. Internal Platform Meta-model](image)

This meta-information can correspond to physical properties such as performance or energy consumption, logical dependencies between components, or versioning information. It will be used to support design tasks such as estimation, simulation, and design space exploration.

Therefore, the Internal Platform Meta-model corresponds to a taxonomy of hardware and software elements with characterization properties. Any extension to the meta-model can be easily included by extending the meta-model classes.

### 2.1.3 Mapping Meta-model (MMM)

The Mapping Meta-model describes the rules used to transform instances of IAMM and IPMM into an instance of the Implementation Meta-model (IMM), as shown in Figure 4.

In the proposed approach, a Mapping defines a set of transformations, which transforms: Application Models into the Internal Application Model; Platform Models into the Internal Platform Model; and, finally, the combination of the Internal Application Model and Platform Model into the Implementation Model. A transformation contains rules, which define the possible mappings between models involved in this design methodology. Furthermore, the Mapping Model guides the exploration tool or a designer to build a candidate Implementation Model. The main function of Mapping Models is to allow a design space exploration tool to automatically optimize the mapping of a PIM into a PSM.

![Figure 4. Mapping Meta-model](image)

### 2.1.4 Implementation Meta-model (IMM)

The Implementation Meta-model allows the representation of the models that can implement the system specification without violating the system requirements. Figure 5 shows the class diagram of the Implementation Meta-model.

![Figure 5. Implementation Meta-model](image)

The IMM can be described as a list of selected platform components, application components, and associations between them. Furthermore, it also represents the design decisions about component configuration, task mapping, and resource allocation.

The information contained in an instance of IMM and in the instances of the other proposed meta-models is used by a design tool to generate: partial embedded software source code; scripts for hardware synthesis; and system deployment scripts. As explained in the next section, other tools, such as an estimation tool, use an instance of the Implementation Meta-model in order to estimate the properties of a candidate implementation model, before the subsequent steps of the development process are performed.

### 2.1.5 Transformation engine

The allocation, partitioning, and binding tasks of the design space exploration process must adopt some deterministic or heuristic approach, such as dynamic programming or genetic algorithms, to investigate the space of possible implementations. For each possible implementation to
be considered, the exploration tool or designer takes some design decisions and incorporates the corresponding information into the conditions of the transformation rules of the Mapping Model.

At this point, the transformation engine can be invoked to traverse the set of transformation rules, so as to select and apply the ones that are enabled at that moment, thus generating a hardware/software mapped architecture in the form of an Implementation Model.

Our transformation engine is implemented using MDDi-QVT (Model Driven Development integration - Query, View, Transformation) [11], which is an open-source implementation of the QVT standard. MDDi-QVT provides a QVT parser, which reads a textual QVT specification and builds a model conforming to the QVT meta-model [13], and a QVT compiler that generates an API in Java from a QVT model, which can be used to perform transformations between models.

In order to use the current version of MDDi-QVT, we generate a textual QVT specification file from the application, platform, and mapping models conforming to our meta-models. Then we apply the QVT parser and compiler to obtain a Java API. This API is used to implement the transformation engine, which is called by other design tools.

2.2. Design flow and tools

The meta-modeling infrastructure can be used by several tasks during the embedded system design. From specification until software/hardware generation and synthesis, the design tools can access the meta-modeling infrastructure to generate specific design artifacts, which will be used as inputs for design tasks. The resulting artifacts can be stored in the meta-data repository to continue the design flow. This section shows how the meta-modeling infrastructure is used by our design methodology in different design tasks.

2.2.1 Specification. In our design methodology, the designer may specify the embedded application using different modeling languages, such as UML, Simulink [9], or others. For each modeling language, a corresponding Application Meta-model is defined, which includes all concepts of the corresponding specification language. Then, the application model is translated into the Internal Application Meta-model (IAMM) of the meta-modeling infrastructure.

Currently, the proposed approach supports only UML application models, but a Simulink meta-model is under development. Other application meta-models can be provided to improve the support for additional languages, such as the SystemC meta-model proposed in [3].

Usually, different Platform Meta-models (PMM) are required to specify platform models, corresponding to the platform modeling languages that are commonly employed in the description of system architectures, such as Java, SystemC, and VHDL. As for application models, in order to have a standard representation for platforms, the PMM is translated into an Internal Platform Meta-Model (IPMM). Besides the mapping from PMM to IPMM, designers can also rely on the Platform Loader, a tool that allows the direct generation of an instance of IPMM [12], according to the resources selected by the designer for the target implementation platform.

Task mapping, processor allocation, and software binding can also be expressed by UML using deployment diagrams. Furthermore, the QVT-based specification language and the repository API, both available in MDDi-QVT, can be used to specify transformation constraints and pre-design decisions to handle resources in the Mapping Model.

2.2.2. Design Space Exploration. A DSE tool [17] is used to select an alternative solution that better fulfills the system requirements. This DSE tool looks for a solution that minimizes the communication, energy, power, performance, memory footprint, or any combination of these properties, under hard real-time constraints, meeting design restrictions such as the processing capacity of each processor (which depends on its frequency/voltage setting). At each moment, the DSE tool takes design decisions and incorporates the corresponding information into the conditions of the transformation rules of the Mapping Model. Then, the DSE tool invokes the transformation engine provided by the infrastructure, so that it produces candidate Implementation Models, which will be evaluated by the estimation tool in order to select the final solution. Figure 6 shows the interaction between the DSE tool and the infrastructure.
Then, the DSE tool automatically selects resources on the Internal Platform Model and takes mapping decisions on the Mapping Model. Exploration could also be performed on the Internal Application Model. However, currently a designer can perform the exploration on the Internal Application Model only manually, as described in [12]. When all design decisions for a candidate solution were taken, the transformation engine is invoked to traverse the set of transformation rules of the Mapping Model. Then, the transformation engine selects and applies the enabled rules in order to produce a PSM in the form of an Implementation Model. At this point, the estimation tool is used to extract the candidate Implementation Model properties and to compute the solution cost, which should be optimized by a heuristic algorithm implemented by the DSE tool.

The DSE tool currently uses a simulated annealing approach as heuristic for the optimization. A more general ant colony optimization approach is being developed.

2.2.3. Model-based Estimation. The model evaluation for early and fast DSE is supported by SPEU [12], a UML model-based estimation tool, which provides analytical estimates about physical system properties (e.g., execution cycles, energy/power consumption, volume of communication data, and memory footprint). These properties are directly obtained from an Implementation Meta-model instance, with reasonable accuracy when the reuse of repository components is largely employed.

The estimation is based on the costs of the components, as specified in the Platform Model, on the structure/behavior captured by Internal Application Models, and on an ILP (Integer Linear Programming) formulation for the identification of best-case and worst-case execution paths. This allows the designer or the DSE tool to evaluate each candidate solution during the DSE process, without depending on costly synthesis-and-simulation evaluation cycles.

2.2.4. Generation and Synthesis. The Generator Tool is based on templates. It uses the meta-data repository API to obtain information from the Implementation Model and to complete these templates, which are specified using a Java-based template language [6].

When communicating tasks are allocated to different processors, the generation of specific communication directives and/or interconnection components, such as a communication API [20], is required. Likewise, the allocation of various active tasks to the same processor implies the generation of scheduler services [21] as well as of real-time directives on each active task to specify its activation pattern, through a real-time API [21].

The Implementation Model corresponds not only to the software source code, but also to a set of scripts, which configure and execute compilers, synthesis tools, and simulators for the generated and assembled components.

To perform the entire design flow, a designer can thus execute a script, such that all design process phases, including automated exploration, compilation, synthesis, simulation, and deployment, will be performed. Using these scripts, a designer can easily make any modifications to the system models (application, platform, mapping, or implementation), generate the source code and update the scripts, and then execute some of the design flow steps again.

3. Case Study

This section presents two DSE scenarios where the proposed MDE-based approach is used to design a real-time embedded system dedicated to the automation and control of an intelligent wheelchair that helps people with special needs. This wheelchair has several functions, such as movement control, collision avoidance, navigation, and automatic movement.

In the first scenario, a designer manually performs the exploration, where two alternative Application Models are obtained for the same application. This experiment focuses only on the wheelchair movement control use case, which is essential to the system and incorporates critical hard real-time constraints. In the second scenario, an automated DSE scenario is presented for the same application using more functions.

3.1. First scenario

In this scenario, two alternative Application Models are specified for the same application. After that, a Mapping Model is built to obtain the Implementation Model. Then, the SPEU tool is used to evaluate both alternatives, extracting estimates from the models.

To implement the movement control, several components provided by the platform can be reused, such as a mathematical library to solve the control equations, a real-time Java API, and RTOS components [21]. Consequently, the platform aspects considered in the estimation methodology also take into account costs that are related to the real-time scheduler. Table 1 shows some of the available components of the platform model. The component Math:API has many services, as for example sin:Service, cos:Service, atan:Service, and sqrt:Service. All these services are used for both implementation solutions. The RestoreContext:Service
is a service of the scheduler EDFScheduler:OS, which restores the processor status and gives control back to a previous thread. The waitForNextPeriod:Service transfers the control from a real time thread to the OS at the end of its execution. The FemtoJavaMCRTC (Multi-cycle, RealTime Clock) is one of the versions of a Java microcontroller available in the platform [8]. It has special instructions that are needed to support the RealtimeThread:API.

Table 1. Some components of the platform

<table>
<thead>
<tr>
<th>Component</th>
<th>Service</th>
<th>Prog. mem. (bytes)</th>
<th>Data mem. (bytes)</th>
<th>Perf. (cycles)</th>
<th>Energy (switching activity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math:API</td>
<td>Sin:Service</td>
<td>14</td>
<td>4</td>
<td>53</td>
<td>73,815</td>
</tr>
<tr>
<td>EDF Scheduler:OS</td>
<td>RestoreContext:Service</td>
<td>4</td>
<td>0</td>
<td>883</td>
<td>8,117.3</td>
</tr>
<tr>
<td>Realtime Thread:API</td>
<td>waitForNextPeriod:Service</td>
<td>243</td>
<td>13</td>
<td>5721</td>
<td>1,254.3</td>
</tr>
<tr>
<td>FemtoJava MCRTC:Processor</td>
<td>int_t0:Service</td>
<td>2</td>
<td>32</td>
<td>124</td>
<td>187.45</td>
</tr>
</tbody>
</table>

The class diagram for the Application Model of the first solution that describes the ‘movement controller’ use case, which has six classes distributed in four packages, is shown in Figure 7.

Figure 7. Class diagram of Application Model

The Application Model specifies the structure and behavior of classes needed to implement the ‘movement controller’ use case, which has two responsibilities: sensing the movement, thus getting information about the speed and angle through two sensors installed in the wheels, and actuating in the wheelchair motors, by writing values obtained from a joystick device into the actuator interface.

Table 2 lists some of the transformation rules of the Mapping Model. These rules specify the possible mappings from elements of the wheelchair application model into elements of the FemtoJava platform, including the actions that will generate elements in the Implementation Model.

As specified by the transformation rules, the modules can be implemented either by a simple FemtoJava [8] or by a multi-cycle real-time FemtoJava [21]. In the Implementation Model, a memory component will be instantiated for program code and data, while a bus will be instantiated to connect the processor to the memory. Other transformation rules state that: a) processes in the application can make use of the Math and RTThread APIs; b) actions in the thread of the processes may use services to suspend the execution (waitForNextPeriod and waitForCycles services); c) actions can have access to static and dynamic object attributes (get/set Static/Dynamic ObjectField services); d) mathematical operations can be implemented by the Math API; and e) scheduling of processes and threads can be implemented by different types of schedulers from a given operating system.

Table 2. Part of the Mapping Model

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAMM</td>
<td>IPMM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod:</td>
<td>p:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module</td>
<td>Processor,</td>
<td>p.name=FemtoJava32 or p.name=FemtoJavaMCRTC</td>
<td>new Processor(), new Memory(), new Bus()</td>
</tr>
<tr>
<td>proc:</td>
<td>a:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>API</td>
<td></td>
<td></td>
</tr>
<tr>
<td>threadctrl:</td>
<td>s:Service</td>
<td>s.name = waitForNext Period or s.name = waitForCycles</td>
<td>new Service()</td>
</tr>
<tr>
<td>interaction:</td>
<td>s:Service</td>
<td>s.name=interactionStatic or s.name=interactionDynamic</td>
<td>new Service()</td>
</tr>
<tr>
<td>mathoper:</td>
<td>s:Service</td>
<td>s.name=sqrt or s.name=sin or s.name=cos or s.name=s.atan</td>
<td>new Service()</td>
</tr>
<tr>
<td>sched:</td>
<td>s:OS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Given the Platform, Application, and Mapping Models, the DSE process takes design decisions about the partitioning, allocation, and binding of application elements to platform elements and incorporates the information on the Mapping Model of Table 2.

To illustrate the DSE process, two alternative design points of the space of possible implementations...
are reported here. In a first implementation, an EDF scheduler is used to manage the threads (s:OS.name = EDFScheduler), and application objects are dynamically allocated (s.name = getDynamicField or s.name = setDynamicField, i.e., only the dynamic getField actions can be allocated for these objects). In a second implementation, there is no scheduler (the last rule in the table is not used) and no dynamic objects (s.name = getStaticField or s.name = setStaticField), and a simple FemtoJava processor can be used (just p.name = FemtoJava32 in the first rule).

The transformation engine was executed for each candidate implementation in order to generate the two alternative Implementation Models, and the SPEU tool was used to calculate the estimates for each alternative. The obtained results for the first and second candidate Implementation Models are listed in Table 3. The last column shows the difference in percentage of the second solution when compared to the first one.

**Table 3. Results for first and second candidate Implementation Models**

<table>
<thead>
<tr>
<th>Property</th>
<th>First</th>
<th>Second</th>
<th>Diff (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program memory</td>
<td>5,460</td>
<td>1,647</td>
<td>-69.84</td>
</tr>
<tr>
<td>Best-case Data mem.</td>
<td>510</td>
<td>324</td>
<td>-36.47</td>
</tr>
<tr>
<td>Worst-case Data mem.</td>
<td>510</td>
<td>324</td>
<td>-36.47</td>
</tr>
<tr>
<td>Best-case Performance</td>
<td>27,383</td>
<td>1,253</td>
<td>-95.42</td>
</tr>
<tr>
<td>Worst-case Performance</td>
<td>40,106</td>
<td>13,706</td>
<td>-65.83</td>
</tr>
<tr>
<td>Best-case Energy</td>
<td>38,723,290</td>
<td>1,485,002</td>
<td>-96.17</td>
</tr>
<tr>
<td>Worst-Case Energy</td>
<td>56,477,333</td>
<td>188,749,973</td>
<td>-66.58</td>
</tr>
</tbody>
</table>

In the design of the first Implementation Model, the system responsibilities were distributed among a larger number of threads and objects. This solution uses four threads, and each one was modeled as a sequence diagram. Two threads implement the movement actuating function, while the other ones implement the movement sensing function. Although this modeling approach is flexible and allows high software reuse, it is expensive in terms of memory and performance. The second Implementation Model was developed as a lightweight one. It uses only static objects and does not use threads to implement the movement control. Instead, it directly uses the interrupt control system. Moreover, this model has a lower encapsulation of functions than the first one. As result, the second solution is better in all six cost metrics, by large differences. Although the second solution is faster, saves more energy, and presents lower memory footprint, it is very limited in terms of flexibility, due to its static design and its low degree of encapsulation.

### 3.2. Second scenario

In the second scenario, the first candidate solution was selected from the first scenario and more functions were added to the application model, which now contains 17 tasks that have communication dependencies between them. The Platform Model is also based on the same Java microcontroller, upon which an API and operating system components for real-time behavior and communication support are available. Figure 8 illustrates the task graph extracted from the Application Model, which has been specified in UML.

In this scenario, the automated DSE has to select the number of processors to be used in the system, map tasks to them, and allocate processors into the communication structure (two busses communicating through a bridge). In order to reduce the energy consumption, the DSE tool also selects the minimum voltage for each processor, but avoids task deadline violations. Realistic energy, voltage, performance, and memory costs for HW and SW platform components and services (such as task scheduling), extracted by using the SPEU estimation tool, have been used to guide the exploration.

![Figure 8. Application Model task graph](image-url)

The number of design alternatives in this DSE scenario is huge, so that the utilization of an automated DSE approach at a very high abstraction level is fully justified.

Figure 9 shows a chart with the final solution found out by the DSE tool for different combinations of objectives. The “x” axis is organized by optimization objective – only energy, only power, only memory, combined energy-power-cycles-memory (epcm), combined energy-power-cycles (epc), and combined energy-power-memory (epm). The values in the “y” axis represent the relative difference between system properties (energy, cycles, memory footprint, and power), when the system is optimized for different objectives. These values are normalized to the range between 0% (best solution) and 100% (worst solution).
The DSE result for energy optimization presents a good load distribution between the six processors. This parallel execution reduces the number of execution cycles, directly contributing to energy savings. Because of data and code replication, the values for memory and power were not optimized. To optimize the power, the DSE tool mapped the simple tasks to the same processor with low voltage settings. Processors with higher occupation present higher voltage settings.

In order to reduce the memory footprint, the DSE tool mapped tasks that exchange large amounts of data to the same processor, reducing in this way the data replication and the communication structures (packages and messages).

In the last three solutions, the chart presents the DSE results using multi-objective functions. Due to the hard real-time constraints, the optimization is more flexible with memory and power, which dominate the optimization process. Better optimizations could be achieved if the Application Domain could also be explored by the DSE tool, thus obtaining reduced values for execution cycles.

Figures 10-13 show the property values estimated for 19 candidate solutions found out by the DSE tool, optimizing the objectives energy, power, and memory. In each chart, the Y-axis shows the property values for the entire system. The X-axis presents the candidate solutions. The presented results were obtained after the evaluation of 1,000 candidate solutions, using a simulated annealing optimization approach. In our experiments, the DSE tool evaluates 1000 solutions in around 1 hour, using an AMD Athlon 1.8 GHz with 512 Mbytes of RAM.

After selecting the candidate solution that better fulfills the system requirements, the design flow can be continued by using the Generator Tool. This tool will generate the software source code and the design tool scripts for compiler, simulator, and synthesis tools. Then, the Implementation model can be refined if necessary for further analyses or simulations, until the final product.
This case study illustrates two DSE scenarios where the ModES tools and meta-modeling infrastructure were used in two different ways. In the first one, a manual exploration was performed at the Application Model level, and alternative modeling approaches for the application were explored. Nevertheless, the framework obviously also allows explorations on the other two models of the Y-design flow. This kind of DSE was performed in the second scenario, where an automated DSE process explores different task mappings, allocations of processors, how hardware components are interconnected through the communication channels, and which are the individual voltage settings for processors.

In both scenarios, the DSE process of a real application presenting a large design space was presented. Through the MDE approach presented in this work, the designer could specify the system using UML models at a high abstraction level, look for the best alternative to be implemented without relying on costly synthesis-and-simulation evaluation cycles, and generate source code and design scripts to automate and facilitate the transition from initial design phases to the implementation phases.

4. Related work

There are many recent research efforts on embedded systems design based on MDE. The adoption of platform-independent design and executable UML has been vastly investigated. For example, xtUML [12] defines an executable and translatable UML subset for embedded real-time systems, allowing the simulation of UML models and the code generation for C oriented to different microcontroller platforms. The Model Execution Platform (MEP) [19] is another approach based on MDA, oriented to code generation and model execution. Our approach, instead, is more related to the design space exploration at a high abstraction level, using the notion of transformations between models, than the validation or code generation, which are the main concerns of these approaches.

The Generic Modeling Environment [4] (GME) is a modeling tool oriented to the development of domain-specific visual modeling languages by the specification of a meta-model. In GME, meta-models are described as UML class diagrams, and, once created the meta-model for a specific domain, a visual editor is automatically generated that supports the edition of diagrams for the domain as well as the generation of design artifacts. GME has been used to implement some model-based tools for embedded system design. MILAN [1] (Model-based Integrated Simulation) is one such tool, where the focus is on the simulation of embedded systems.

The DaRT [2] (Data Parallelism to Real Time) project also proposes an MDA-based approach for SoC (System-On-a-Chip) design that has many similarities with our approach in terms of the use of meta-modeling concepts. The DaRT work defines MOF-based meta-models to specify application, architecture, and software/hardware associations and uses transformations between models as code transformations to optimize an association model. In doing so, it allows the re-factoring of an application model in order to better match it with a given architecture model. In DaRT, no design space exploration strategy based on these transformations is implemented, and the focus is mainly the code generation for simulation at TLM (Transaction Level Model) and RT (Register Transfer) levels.

Compared to our approach, no related work takes advantage of the MDA notion of transformation between models to represent, delimitate, and explore the design space of possible implementations for a given application on a specified platform at a high abstraction level.

5. Conclusions and future work

In this paper, the MDE fundamental notion of transformation between models is used to represent and delimitate the design space of possible implementations of the specified application on the specified platform. A set of transformation rules defines the possible mappings from application into platform, and by executing these rules possible implementations can be quickly generated, under control of a design space exploration tool. For each implementation, a design tool can obtain estimates without any costly hardware and software synthesis/simulation steps.

One of the future directions to be considered is the meta-modeling of design constraints in the Internal Application Meta-model. These constraints shall be implicitly incorporated into the conditions of the transformation rules of the Mapping Meta-model, in such a way that the design exploration process considers them.

References