MDE Approach to the Co-Synthesis of Embedded Systems Using a MOF-based Internal Design Representation

Francisco Assis M. do Nascimento, Marcio Ferreira S. Oliveira, Flávio Rech Wagner
Institute of Informatics - Federal University of Rio Grande do Sul - UFRGS
Av. Bento Gonçalves, 9500, Porto Alegre/RS, Brazil
fanascimento, mfsoliveira, flavio@inf.ufrgs.br

Abstract

This work presents a Model Driven Engineering (MDE) approach to the generation of a MOF-based internal design representation for an embedded application described using UML class and sequence diagrams. Differently from other similar MDE-based approaches for embedded system design, which translate UML models to some specific internal format, our internal design representation, as a MOF-based metamodel, can take advantage of the concept of transformation between models to implement co-synthesis tools. Our internal design representation captures the specified system as a set of hierarchical modules, interconnected by ports, and processes, which communicate through send/receive operations using channels or through shared variables. The execution flow of the application is also captured by means of a control/data flow graph, which is used as input to a co-synthesis framework for embedded systems design. The paper describes the defined transformations between models and illustrates them with a case study to show the effectiveness of the approach.

1. Introduction

Approaches based on MDE (Model Driven Engineering) have been proposed as a solution to cope with the complexity of embedded systems design [21]. One variant of MDE is the Model Driven Architecture (MDA) [16], which is a framework proposed by OMG (Object Management Group) for the software development, driven by models at different abstraction levels.

In an MDA approach, a system is modeled using a platform-independent model, which is transformed into a platform-specific model, given a platform model. The languages used to express these models are defined by means of meta-models that are able to define the abstract syntax of the modeling languages, for which concrete syntaxes as well as an operational semantics are usually given.

A very popular modeling language is UML (Unified Modeling Language) [19], which has been used for the specification of embedded systems at a high level of abstraction [21]. UML adopts the object oriented paradigm and includes different diagrams for the modeling of structure and behavior. For instance, UML has Class and Component diagrams to capture structure and Interaction (Sequence and Collaboration), Activity and State Machine diagrams to represent behavior. These diagrams are adequate for the modeling at a high level of abstraction.

In order to be used as input representation for co-synthesis tools, a UML model must be translated into some formalism that can expose the control and data flow of the specified application in a concise and efficient way, since this information is essential to the algorithms used in the existing synthesis tools. Most of these tools adopt control data flow graph based representations, which allows the traversal of the execution paths of a given application. [8].

In a UML model one can use Activity diagrams to specify such kind of information, but the internal representation defined in conformance to the OMG’s UML metamodel [18][17] is not adequate to implement co-synthesis algorithms, since the information is dispersed in different parts of the UML/MOF metamodel. This makes very difficult to perform some basic operations on this representation, which are necessary for the co-synthesis algorithms [8].

Differently from all other approaches oriented to MDA for embedded system design, which translate UML models to some specific internal representation, we use only MOF concepts to define our internal design representation. As a MOF-based metamodel, our internal design representation can take advantage of the concept of transformation between models to implement the co-synthesis tasks.

The remaining of the paper is organized as follows. A comparison of our methodology with other related MDE-based approaches is given in Section 2. Section 3 introduces our MDE-based approach for co-synthesis, Section 4 describes the Internal Application Meta-Model that can cap-
ture in a standard way the specified application, and Section 5 presents the transformations between models that generate our Internal Application Model from the UML model of the application. Section 6 describes a case study, which illustrates our approach. Section 7 draws main conclusions and proposes future research directions.

2. 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 [14] 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 DaRT (Data Parallelism to Real Time) project [4] 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, however, no co-synthesis strategy based on these transformations is implemented, and the main focus is mainly the code generation for simulation.

The Internal Format (IF) from the OMEGA project [11] associates to each class a process and captures the behavior as state machines that represent the interactions between these processes. There is no concept of module to group processes and so to take into account the different forms of communications according to the partitioning of the processes. This missing information would be essential for the co-synthesis tasks.

In the approach presented in [22], UML Sequence Diagrams are translated into a communication dependency graph in order to implement a specific performance analysis technique. Thus, this approach does not consider the structure and hierarchy of a UML model, as in our approach.

The co-synthesis tool POLIS [3] has an internal design representation, called CFSM (Co-Design Finite State Machine), which allows the implementation of efficient co-synthesis strategies. However, it is not possible to use UML as input modeling language for POLIS, neither to implement the co-synthesis tasks using MDE concepts.

Another important task in a co-synthesis framework is hardware/software co-simulation. For this task, Ptolemy [13] is a very popular tool, but it cannot use UML as modeling language either.

3. ModSyn: Model Driven co-Synthesis

Our MDE-based approach to the co-synthesis of embedded systems [6] adopts 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. In this paper, we present one part of this approach, which, from a UML model, performs the generation of an internal design representation that is adequate for the co-synthesis tasks.

Differently from IF [11], which does not consider the influence of the structural features of the UML model in the communication behavior of a specified application, and from the approach in [22], which takes into account only UML Sequence Diagrams in order to identify the communications between the objects in the UML model, our internal design representation also captures the hierarchy and communication structure of the UML model in the form of a graph, which can represent the control and data flow dependencies in a convenient way for the co-synthesis algorithms.

Figure 1 shows our MDE-based design flow, called ModSyn (Model-driven co-Synthesis for embedded systems design). In our approach, the application is specified independently from the platform, using UML [19] as modeling language, but any other DSL (Domain Specific Language) [21] could also be used. A mapping defines how application functionality is partitioned among architectural components in order to produce an implementation for the specified system.

![Figure 1. ModSyn Design Flow](image-url)

Accordingly, four internal meta-models allow the independent application and platform modeling: Internal Application Meta-model (IAMM), Internal Platform Meta-Model (IPMM), Mapping Meta-model (MMM), and Implementa-
Interconnection Meta-Model (IMM). Each meta-model provides the abstract syntax for the adopted design concepts in ModSyn. For each of these ModSyn meta-models we also have defined a concrete syntax. Thus, one can see these metamodels as defining DSLs, which are adequate to represent each kind of design information in ModSyn. These metamodels are described in [6] and will not be detailed here. This paper will present the generation of an internal design representation model conforming to IAMM from UML class and sequence diagrams. This task is performed by the Application Manager, which is shown in Figure 1.

The Application and Platform Managers in Figure 1 adopt an MDE approach to generate the internal design representations (Internal Application Model - IAM, Mapping Model - MM, and Internal Platform Model - IPM), by performing model-based transformations. In the current version, we use the open source Eclipse Modeling Framework (EMF) [7] and oAW (openArchitectureWare) [20] features to implement ModSyn. We have Ecore representations for our meta-models, and a designer can use the UML2 editor plug-in in Eclipse to create applications, platforms, and mappings models, which are also stored in an Ecore-based repository. This Ecore repository also stores the Implementation Model, which is generated by the transformation engine during the design process.

The System Designer performs the transformations between models that generate inputs for different co-synthesis tools as well as incorporate the outputs of these tools into the Ecore repository. From the repository, the Implementation Manager can generate the final implementation for the specified system by means of transformations from models to code [5].

The ModSyn framework provides transformations between models that can generate a Timed Labeled Transition System (LTS) [1] from IAM, which can be used in the Uppaal model checker [12] to perform formal verification of specified properties for the system. This feature is very useful for the designer, since the LTS is automatically generated and can help the designer to debug and validate the specification. The framework also provides the generation of Co-Design Finite State Machines to be used by the Polis framework [3] in the Hardware/Software Partitioning, and an actors-based model for functional simulation using Ptolemy [13].

4. Internal Application Meta-model

In ModSyn, a system under design is specified by means of structural and behavioral UML constructs. For the system structure, UML class diagrams indicate the components of the system under design. The system behavior can be specified using UML Sequence diagrams that indicate the allowed execution scenarios.

In order to represent an application in a standard way, a model that is captured using UML is translated into a common application model defined by the Internal Application Meta-model (illustrated by Figure 2 and Figure 3). This translation is implemented in ModSyn by means of transformations between models.

As shown in Figure 2, in an application model conforming to IAMM, a system specification captures the functionality of the application in terms of a set of modules (Module class). Each module has module declarations (ModuleDeclaration class) and a module body (ModuleBody class).

As Figure 2 also illustrates, a module declaration can be used to specify typed channels, ports, signals, and variables (AppChannel, AppPort, AppSignal, and AppVariable classes, respectively). Here the concepts of ports, signals, and variables come from hardware description languages, as VHDL (VHSIC Hardware Description Language) [2]. The channels are used by processes to send or receive messages. The ports interconnect the modules. Signals can be used to specify shared memories for processes. Variables correspond to the local memories for processes.

As also presented in Figure 2, a module body consists of interconnections (Interconnection class) with other modules and a module behavior, as well as sub-modules (as indicated by the association from the ModuleBody to the Module class).

In order to interconnect the module ports we have the concept of Net, which can connect two ports. The module behavior is captured in terms of a set of processes (AppProcess class), and each process has a set of actions, which will represent the occurrence of UML events in the scenarios (UML Sequence diagrams), as will be shown in
the following.

The control and data flow of an application model is represented by an InteractionGraph, presented in Figure 3. In the definition of our InteractionGraph, we adopt an approach similar to the proposed in [9], which takes concepts related to UML Activity diagrams from the UML/MOF meta-model.

As illustrated in Figure 3, an InteractionGraph consists of a set of nodes (IGNode class) and edges (IGEdge class). Each node can represent different kinds of control flow (IGInitialNode, IFFinalNode, IGForkNode, IGJoinNode, IGMergeNode, and IGDecisionNode classes) and two kinds of executable nodes (IGCallNode and IGReplyNode, sub-classes of the IGMNode class), which represent the possible actions of sending and replying messages in the UML Sequence diagram.

For each UML Sequence diagram $SD_m$ there is an InteractionGraph $IGSD_m(V,E,K,L)$, which is a control/data flow graph (CDFG) [8], where $V$ is the set of nodes, $E$ is the set of edges, $K : V \rightarrow \{Initial, Final, Fork, Join, Merge, Decision, Call, Reply\}$ is a function that indicates the type of each node, and $L : V \rightarrow \{IGSD\}$, is a relation that associates a IGMNode to another InteractionGraph and allows the capture of the behavioral hierarchy of the application. For the application model, we have an InteractionGraph $IG_{app}(V,E,K,L)$, which captures the control/data flow of the entire application.

Therefore, our Internal Application Meta-model captures structural aspects of an application model by using a hierarchy of modules and processes, as well as behavioral aspects by means of the actions of sending and replying messages, where a message may execute some method in the corresponding object. The presentation of a formal semantics for our Internal Application Meta-Model is out of the scope of this paper. However it is important to emphasize that these concepts of modules, intercommunicating processes, and actions, adopted by the IAMM, correspond to similar concepts from CSP [10] and CCS [15], which are able to express any kind of concurrent, distributed system.

In order to represent the functional behavior of a UML model, ModSyn adopts a network of timed automata model conforming to the Labeled Timed Automata Meta-model (LTAMM) (illustrated by Figure 4), which is part of our IAM and captures all concepts introduced by the UPPAAL model checking tool [12]. This translation is also implemented in ModSyn by means of transformations between models.

As Figure 4 shows, conforming to the LTA Meta-Model, a system consists of ltaDeclarations, which can be used to declare variables, functions, and channels, and ltaProcesses, which are instances of ltaTemplates. Each ltaTemplate corresponds to a timed automaton, which can also have ltaDeclarations of local variables and functions. Each timed automaton is represented by a set of ltaLocations and ltaTransitions, which have source and target locations. Each transition may have attributes: ltaSelections (non-deterministically bind a given identifier to a value in a given range when transition is taken), ltaGuards (transition is enabled in a state if and only if the guard evaluates to true), ltaSyncronizations (transitions labelled with complementary synchronization actions - send and receive - over a common channel synchronise), and ltaUpdates (when transition is taken, its update expression is evaluated and the side effect of this expression changes the state of the system).
The transformation from a UML model into our Internal Application Model consists of a set of transformations between models, which are implemented using the Xtend language from the openArchitectureWare framework [20]. Figure 5 shows part of the model transformations in Xtend, where lines 1-2 import the UML and ModSyn meta-models in order to make them visible by the transformations.

```java
import uml;
import modes;

@Object(top | Model, currentModel | ) transformUMLpackage(umlPackage | ) {
    @Package top | Package (p | ) | handlePackage(p | , this | ) | if p.ownedElement | typeSelect | (Package | ) | mapSubPackagesByClass(p | , this | ) | then new InteractionGraph | this initIG(IGFragment | i | , app | ) | if i.ownedElement | typeSelect | (InteractionFragment | ) | mapInteractionToIG(app | ) | then create List[modes::Process] this mapProcessesInInteraction(app | ) | if Collaboration c | in app.IGs | then mapIGsInCollaboration(c | , this | ) | else if ModuleBody mb in app | then mapModuleBody(mb | , this | ) | else null | this | ;
    @Package top | Package (p | ) | handlePackage(p | , this | ) | if p.ownedElement | typeSelect | (Package | ) | mapSubPackagesByClass(p | , this | ) | then new InteractionGraph | this initIG(IGFragment | i | , app | ) | if i.ownedElement | typeSelect | (InteractionFragment | ) | mapInteractionToIG(app | ) | then create List[modes::Process] this mapProcessesInInteraction(app | ) | if Collaboration c | in app.IGs | then mapIGsInCollaboration(c | , this | ) | else if ModuleBody mb in app | then mapModuleBody(mb | , this | ) | else null | this | ;
    @Module(top | Module (m | ) | addAll(m | ) | )
}
```

Figure 5. UML to IAM transformation in Xtend (part 1)

The main transformation is given in lines 6-10, where each `Package` in the UML model is traversed (line 7), then the sub-modules are identified (line 8), the processes are built from the Sequence diagrams (line 9), and, finally, the `InteractionGraphs` are also built from the Sequence diagrams (line 10).

As can be seen in lines 12-14 in Figure 5, showing the function handlePackage(), each `Package` in the UML model is traversed recursively (see line 14 in Figure 5), and each existing UML Class in a package is transformed into a `Module` class by calling the function mapModule() (line 13 in Figure 5).

Each UML Attribute of each UML Class is transformed into a `ModuleDeclaration` class, as shown in lines 17-18 in Figure 5. The associations between the UML classes will determine the sub-modules of each module: Each UML Class, which is part of an aggregation or composition of another UML Class, will be transformed into a `Submodule`, by calling the function putSubModule() (see lines 23-25 in Figure 5). Derived classes are transformed into modules, and all the inherited attributes are replicated inside each such module.

In the UML MOF, each Lifeline in each UML Sequence diagram is transformed into a process, which has its actions determined by the Message classes covered by the corresponding Lifeline class. As can be seen in Figure 6, each Lifeline class is referenced by a Property class (see line 37) of an Attribute class in each `InteractionClass` (see lines 32-33) inside a Collaboration class of the UML model.

Thus, for each Property class, corresponding to a Lifeline, the function mapProcessByAttribute() inserts a new Process class in the `ModuleBehavior` class of the `ModuleBody` class in the corresponding `ModuleClass` (see lines 39-45 in Figure 6).

As presented in Figure 6, for each `InteractionFragment` class, corresponding to a Sequence Diagram (see lines 49-50), in the `Collaboration` classes of the application model an `InteractionGraph` is built (lines 55-57) and inserted into the list of IGS in the `ApplicationClass` (line 57).

The function initIG() initializes and creates an `IGInitNode` and an `IGFinalNode` for each `InteractionGraph` (lines 59-65 in Figure 6). After that, for each synchronous message call or signal call in the Sequence diagram a `IGCallNode` is created, as shown in lines 69-70 in Figure 7, and for each reply message a `IGReplyNode` is created (line 71).

`IGCallNodes` and `IGReplyNodes` are labeled with
The UML class diagram in Figure 8 shows the MoveCtrl class, which represents the wheelchair movement controller with sensor and actuator drivers (represented by the Driver class), and a navigation mechanism (represented by the Navigator class with a Joystick component). There are two types of movement controllers (represented by MoveS and MoveC classes) that have different functions to determine each move for the wheelchair.

In Figures 10 (a), (b), and (c) we have UML Sequence diagrams that specify possible execution scenarios for the application. By applying the transformation between models presented in the previous section, we obtain the Internal Application Model partially shown in Figure 9.

6. Case Study

The case study consists of a real-time embedded system dedicated to the automation and control of an intelligent wheelchair, which has several functions, such as movement control, collision avoidance, navigation, target pursuit, battery control, system supervision, task scheduling, and automatic movement. In order to illustrate the generation of an internal application model from a UML model, we focus only on the wheelchair movement control, whose simplified
Figure 10. UML Sequence Diagrams

The InteractionGraph for the entire application is shown in Figure 12(b), where we have three IGExecutableNodes cn-ig1, cn-ig2, and cn-ig3, which are associated by the relation $L$ to the corresponding InteractionGraphs of the sequence diagrams $SD1$, $SD2$, and $SD3$, respectively. The IGForkNode $fk$-M1 and IGJoinNode $jn$-M1 indicate that the three InteractionGraphs are composed in parallel.

The simple example illustrates well how our Internal Application Model is able to capture the structural and behavioral aspects of a given application modeled using UML. In the current version of ModSyn some of the constructs of the CDFG corresponding to the InteractionGraph of the Figure 11. The IGExecutableNodes are shown as circles, the IGControlFlow edges as arrows, and the IGControlNodes as rounded boxes.
UML2 are not yet implemented, as for example, the combined fragments in the sequence diagrams.

7. Conclusions and Future Work

In this paper, the MDE fundamental notion of transformation between models is used to generate an internal representation model to be used by co-synthesis tools, from a UML model of an application consisting of Class and Sequence diagrams. The obtained model captures structural aspects of an application model by using a hierarchy of modules and processes, as well as behavioral aspects by means of a control/data flow graph model.

We are currently implementing co-synthesis algorithms based on this internal representation model conforming to the Internal Application Meta-Model and using the concept of transformations between models from MDE to perform the co-synthesis tasks, as, for example, the task of hardware/software partitioning applied on the processes represented by the InteractionGraphs of an application. Some types of message calls and the combined fragments in the sequence diagrams of UML 2.0 are not yet handled by our current implementation and will be one of our topics for future work.

References