Proceedings of the 17th IEEE International Conference on Emerging Technology and Factory Automation (ETFA 2012)

A Code Generation Framework for Distributed Real-Time Embedded Systems

Mario Bambagini, Marco Di Natale
{*mario.bambagini, marco.dinatale*}*@sssup.it*
Scuola Superiore Sant'Anna Pisa, Italy

***Abstract*—Modeling languages and tools, including Simulink, Scicos, SysML and the Eclipse Modeling Framework (EMF), bring the promise of an improved quality and productivity in the development of embedded systems and software. Unfortunately, none of these modeling languages, taken individually, is capable of fulfilling all the needs in the development of complex distributed embedded applications, from the modeling, analysis and validation stages to the automatic generation of the implementation. Overall, their strengths and weaknesses are somewhat complementary and an integrated approach could be the most promising solution. In this paper, we present a framework for integrated code generation in complex real-time distributed systems, where MBD approaches are used for the analysis and the generation of the functional (or behavioral) part, and MDA approaches (SysML/EMF) are used for modeling the execution platform, the task model and the deployment of functions onto the platform resources. This paper presents a meta-model for the description of execution platforms and an open-source code generation framework, based on the selected mapping of the functional components on the chosen platform.**

I. Introduction

The functional complexity of modern embedded systems is rapidly growing and distributed systems with time constraints are today typical of automotive, avionics and control systems. In these cases, the planning of the functional deployment and the efficient use of platform resources is crucial.

The use of models for the analysis of the system properties, the documentation of the design decisions and possibly the automatic generation of the software implementation is an industrial reality, backed by several commercial products. Two approaches are today emerging. The Model-Based Design approach (MBD) [1] prescribes models based on a formal Model of Computation, or at least with a formally defined executable semantics. This allows the simulation of the functional model and possibly also the verification of the model properties by model checking. A synchronous (reactive) execution model is assumed by these tools. Automatic generation of an implementation is also typically provided. Examples of such tools are Simulink [2], Scicos [3] and SCADE [4].

The Model-Driven Architecture (MDA) [5] initiative from OMG [6], in contrast, originated from the object-oriented software community and, rather than focusing on a formal semantic backing and analysis capabilities, places emphasis on generic but extensible models, managing structural complexity and supporting the automatic generation of implementations for the structural part (declaration and creation of systems and subsystems, communication and synchronization objects). MDA puts emphasis on meta-meta-modeling capabilities, languages for the transformation of models into other models (M2M) and models into text (M2T). In particular, MDA recommends a development flow in which a Platform Independent Model (PIM) is developed first, and then a Platform-Specific Model (PSM) is created from it by model-to-model transformations when the execution platform and the functional deployment are defined. The PSM allows the generation of an implementation that is executable on the given platform. The separation of the (functional) PIM model from the platform model and the PSM allows to improve software portability and reuse.

The MBD methodology lacks in the capability of modeling architectures, as well as computation and communication delays that depend on the platform. Available commercial code generators provide implementations only for code to be deployed on a single CPU or, to a limited degree, for time-triggered distributed systems. Often, the designer needs to provide an implementation for an execution platform which is not time-triggered, adding custom-developed communication code and performing the application partitioning by hand.

On the other hand, MDA offers high flexibility in the definition of domain-specific models, including tasks, resources and execution platforms. Moreover, MDA allows to define functional deployment models and include model-to-model and model-to-code transformations that may ease the generation of communication and tasks code. The behavioral semantics of the languages recommended by the OMG for the MDA (including UML [7] and SysML [8]) is incompletely specified, so simulations and verifications on those models are tool-specific and often characterized by a limited scope.

The framework presented in this paper aims at bridging the gap between the two approaches. The starting point is the synchronous model of the application functions, developed in Simulink (in the MBD fashion). A meta-model based on the Eclipse EMF [9] has been designed as a superset of the Simulink meta-model to allow the import of the functional model, and also provides modeling concepts to define the execution platforms and to deploy the function components onto the platform resources.

Our framework assumes that the functional model consists of a set of functions that can only be activated at events belonging to a periodic stream. While this may seem a limitation, code can only be generated by Simulink models when a fixed-step solver is used to integrate the continuous parts. In this case, all functional blocks (i.e.: function-activated blocks) are activated at multiples of the system base period.

The framework includes a deployment verification engine that can verify if the deployment solution, provided by the user, preserves the synchronous semantics of the MBD model. In case the deployment results in additional computation or communication latency (beyond the synchronous model assumptions), it provides methods for the worst-case evaluation of the delays that should be added to the synchronous model. In addition, the framework includes a code generation facility, based on the Acceleo plug-in [10], that, based on the platform and deployment models, generates the communication code and the code of the tasks. This infrastructure code is then linked with the behavioral code generated starting from the functional blocks of the Synchronous Reactive (SR) model to construct the application.

The development steps, which are covered by our framework and outlined in Figure 1, are summarized as follows.

1) *Functional model*, defined in Simulink and then exported in an XML format to the Eclipse/EMF framework;
2) *Platform selection*, allows the designer to select the hardware and software resources and the resource management policies in the system architecture. The architecture is completed by the software platform stack, including the Real-time Operating System(s) or RTOS, device drivers and communication stacks with an optional middleware. Platform services are accessed using a standardized API;
3) *Mapping*, provides the definition of the mapping of the functional components onto the computation and communication resources of the platform. As a result of this step, the following stages are enabled;
4) *Performance analysis* for estimating the quality of the designed solution with respect to time (and checking schedulability);
5) *Code generation* for the automatic generation of the code implementation from the models. The (platform-independent) behavioral code is generated using the Simulink Coder tool [11] by Mathworks, while the (platform-dependent) communication, synchronization and tasking code is generated from the EMF framework using OMG standard languages and open-source tools. Ensuring a consistent code generation requires tool synchronization and exchange of information;
6) *Back-annotation*. The definition of the platform and the mapping of the functions onto it allows the evaluation of the communication and computation delays. These delays (evaluated in the worst-case or with stochastic analysis methods) can be backannotated on the functional model creating another model that can be analyzed by simulation to evaluate the performance impact of the platform selection and function-to-platform mapping.

In addition to the generation of the communication and synchronization code, the Eclipse-based framework is also in charge of generating the code which provides the configuration of the operating system(s) and the drivers.

Organization of the paper: Section II presents the tool framework. Section III describes the plug-ins, focusing on the code generation process. Section IV shows an example of code generation. Finally, Section V ends the paper with the concluding remarks and a discussion of the future work.

A. Related work

In agreement with modern development methodologies [12] [5] [1], Model-Driver Architecture (MDA) recognizes system development as a multi-stage effort, in which a set of required functions, defined in an abstract or Platform Independent Model (PIM) are deployed, possibly automatically, onto an execution architecture. The result of this step is a Platform Specific Model (PSM). The term Model-Based Design (MBD) indicates a slightly different approach and especially a different set of models and tools (considered as competing or alternative with respect to those used in an MDA process). While MDA originated from the move of a fundamentally software-oriented community (object-oriented design) towards system-level modeling and embedded systems development, MBD is very popular in the development of control-oriented functions and originated from the domain of control engineering and systems engineering. As such, MBD languages are usually based on a restricted but formal syntax and semantics, with an underlying Model of Computation (MoC) based on mathematical rules. A Synchronous Reactive semantics (SR) [13] is the foundation of the most popular tools such as Simulink and SCADE. It has been proposed [14] [15] [16] that future trends in model engineering will encompass the definition of integrated design flows exploiting complementarities between UML or SysML and Matlab [17]. Matlab/Simulink provide a stronger semantics characterization of the modeling language and the possibility of modeling the controlled system as a continuous-time system of differential equations. The MBD languages allow for the simulation of the controller-plant interactions and verification capabilities. The OMG is currently enriching UML and SysML with a set of more formal action languages, including fUML and Alf. However, these languages are still not fully established in the practice and supported by tools (the Alf 1.0 specification is of February 2011) and do not allow for the modeling of physical systems and continuos-time systems (they are still oriented to the software components).

The combination of the two models (SysML and Simulink) requires the capability of model-to-model transformations and integration of heterogeneous models. These operations are today often performed by hand, motivated by the fact that proprietary modeling languages, such as Simulink, lack a publicly accessible meta-model [14]. The matching between functional and execution architectures is advocated by many in the academic community (examples are the Y-cycle [18] and the Platform-Based Design PBD [19]) and in the industrial domain (the AUTOSAR automotive standard [20] is probably the most relevant recent example) as a way of *obtaining modularity and separation of concerns between functional specifications and their implementation on a target platform. In this way, true portability of functionality and full reuse of*

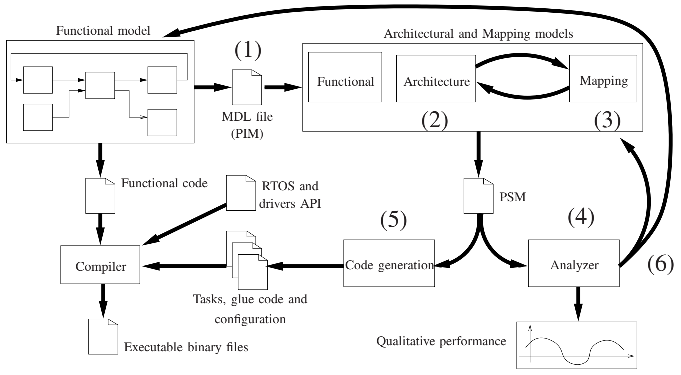


Fig. 1. Design and code generation flow.

both the functional components and the execution platform systems can be achieved. The OMG similarly proposed the MDA development, in which a PIM is transformed into a PSM, although without an explicit separation between the execution architecture modeling and the mapping of functions to architecture. Examples of methods tools and case studies in the transformation of a PIM into a PSM are in [21]. In reality, very few examples exist for the application of the proposed methodology to the mapping of complex functionality to a distributed embedded system, in which a messaging and task structure arises from the mapping. Approaches that are tailored to large-scale embedded systems include the development of domain-specific languages for the platform entities either by the creation of specialized profiles of a standard language, such as the MARTE profile [22] built on top of UML or the EAST-ADL language [23]. Code generation and integration of heterogeneous models are not among the goals of these projects. Similarly, the generation of code from a PSM is mostly limited to customization of a specific operating system or communication (middleware) layers. As for the model-to-model transformations and heterogeneous models integration, several approaches, methods, tools and case studies have been proposed. Several approaches (examples are GME [24] and Metropolis [25]) propose the use of a general meta-model as an intermediate target for the model integration. Also, the Eclipse modeling framework [26] provides support for meta-model specifications through its ECore meta-meta-modeling language [9]. Model-to-model transformation engines are available for the Eclipse environment including ATL [27] and QVT [26].

Raghav et al. [28] and Hugues et al. [29] proposed two similar MDA methods for describing the functional behavior according to a reference architecture and then comparing the deployed system with respect to the reference to check whether the performance (delay) target is guaranteed. Although the performance is checked against scheduling delays, there is no clear separation of the functional and platform designs.

Contributions of the paper:

The proposed framework bridges the gap between the MBD and MDA approaches. A meta-model is defined to represent the functional model, the deployed architecture and the mapping information. The functional and architectural models evolve independent from each other. Only the mapping model establishes a connection between them, indicating where the functional components are allocated on and how data is exchanged. A code generation facility is provided for the automatic generation of the task, communication and synchronization code. The performance is evaluated by estimating communication delays and task scheduling delays and then simulating the behavior of the system for the selected platform option.

Among the cited commercial and research frameworks, those that provide a clear separation of the functional and platform models (such as AUTOSAR, GME, Metropolis or the ATESST/EAST-ADL framework) typically do not provide a code generation path for distributed implementation based on open source tools and open standards (such as the Eclipse EMF). Other proposed frameworks do not allow the merger of synchronous models into an MDA architecture model (with enough information to support flow preservation) but strictly follow the MDA approach. Examples are the open source Papyrus and Topcased frameworks.

Other projects focuses on the modeling infrastructure and the capability of modeling timed events (TIMMO/TIMMO2 [30]) with (at least until now) no support for code generation. Compared with other projects that put emphasis on the code generation infrastructure (such as GeneAuto [31] or ProjectP [32]), we aimed at the use of Ecore meta-models and standard model-to-text transformations; we provide support for modeling distributed execution platforms (missing in GeneAuto) and the corresponding code generation stages for network communication.

II. Modeling Framework

The proposed framework covers aspects related to the creation, analysis, and deployment of the models. Except for the commercial tools for functional modeling (Simulink), all of the project meta-models, editors and code generators have been developed as a set of plug-ins of the freely-available Eclipse EMF. EMF provides not only the capability to define meta-models, but also tools for the automatic generation of Java classes, model serialization (in XMI) and factories, as well as simple tree-based model editors. The following section focuses on the definition of the meta-models that have been proposed to represent the platforms, the function-to-platforms and mapping models. The code generation capability, built on the model-to-text Acceleo generation engine, is described in Section III. The tools for the timing analysis and the model back-annotation are under development.

The meta-model is divided in three main packages: Functional, Platform, consisting of a Library and System Architectural sub packages, and Mapping. For the sake of simplicity, links among the packages are not shown in the figures, and will be explained only when needed.

The functional meta-model, depicted in Figure 2, represents the PIM and is designed to be as general as possible to accommodate imports from different modeling tools. It consists of a net list of functional subsystems (or components), and the constraints on their execution (derived from the semantics of the functional model or annotated after their import), including their activation events, the execution order and the timing constraints. Because of the need to connect the code generated in Eclipse/EMF starting from the platform mapping model with the code generated by the Simulink RTW/EC tool-chain, a set of rules has been adopted which defines the smallest unit of execution in the functional model. In our case, the smallest unit corresponds to a Simulink (non virtual) subsystem, executing at a single rate. The Mathworks code generator is instructed to generate a (C-language) *Step* function, implementing the state update and the output update part of the model blocks inside the subsystem.

When imported in our functional meta-model, the net list is composed by the following elements:

- *Proc*, representing an active object offering a set of functions in a provided interface with a mandatory *Step* function plus the initialization and termination operations, and several required interfaces consisting of the Read and Write methods from/to the input/output ports;
- *Var*, a passive object matching the concept of a logical connection between *Procs* or the interface object between a *Proc* and the controlled system (an I/O interface). It is equivalent to a Simulink signal connection between two functional subsystems of the controller or between a controller subsystem and the controlled plant;

- *Trigger*, events commanding the execution of *Proc Steps*.

The constraints include *TimeConstraint* specifications, which may be either activation events assumptions (periods or minimum inter-arrival times) or deadline constraints, as applied to method invocations paths (represented by *EventPath*, *Path* and *Event* concepts).

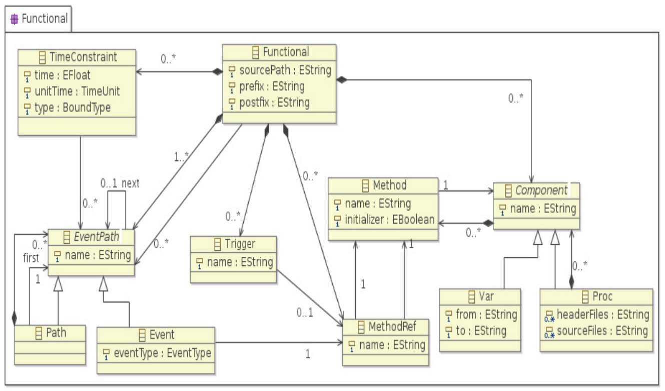


Fig. 2. The meta-model for the description of the functional subsystems.

The execution platform meta-model, shown in Figure 3, defines the hardware and software resources available in the system. The execution platform model needs to be modular and to support the concept of component libraries.

The main architectural elements that are used by the execution architecture designer are the followings:

- *Embedded Control Unit* (ECU), which is a set of electronic boards, connected through communication links;
- *Board*, which may host several *Controllers* and *Devices*;
- *Controller*, which may include several *Cores* and *Peripherals*;
- *Core*, representing a computational unit;
- *Peripheral*, representing an electronic component which extends the Controller's functionalities;
- *Devices*, which represent I/O devices using a set of peripherals. So far, the meta-model supports buttons, touch screens, leds, lcd displays and servo motors. Communication units belong to a special class of devices, handled separately;
- *Real-Time Operating System* (RTOS), which may run on *Cores*.

The platform meta-model is completed by the project-specific definitions, as shown in Figure 4, showing the elements of the hardware and software architecture deployed for supporting the execution of the functions in one specific project instance. The main entities at this level are ECU instances (*ECUDeployment*), with the RTOSs executing on them (*RTOSDeployment*) and the communication buses (*Bus*). Each *Bus* object references the connected ECUs and the associated communication devices.

Figure 5 shows a screenshot of the developed Eclipse plug-in which defines the platform execution model out of the library components.

Once the system execution architecture is defined, a mapping model associates functional elements to tasks and then tasks to the (HW) processing elements, following the schema of Figure 6. The communication signals of the functional view

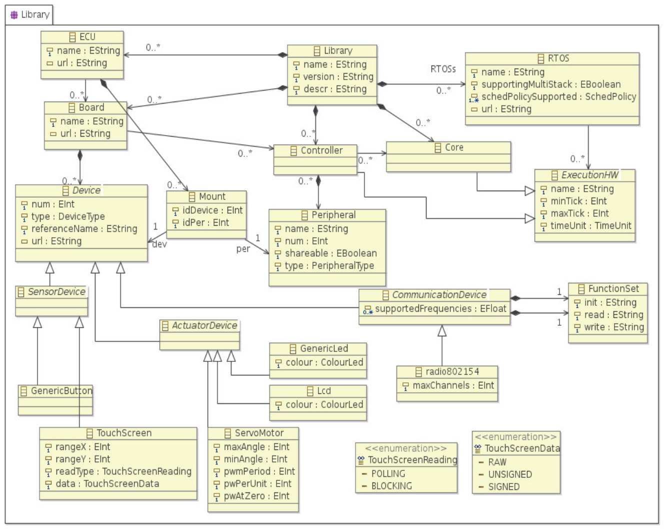


Fig. 3. The meta-model for the library components that are used to construct the execution platform.

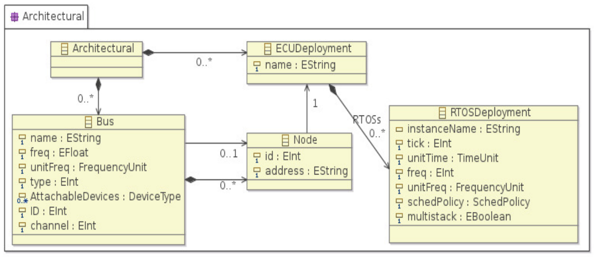
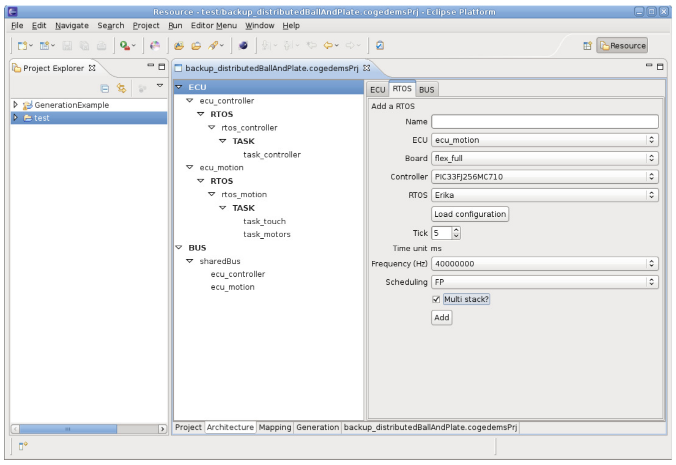


Fig. 4. Meta-model of the objects that are used to build the project-specific execution architecture.

are mapped (when needed) to messages and in turn, messages onto the physical links of the execution platform.

The task is the unit of concurrent execution that can run on one of the system cores, under the control of an operating system. The information about which RTOS hosts a specific task is handled by an association between the *Task* objects


Resource - test/backup_distributedBallAndPlate.cogedemsPrj - Eclipse Platform
File Edit Navigate Search Project Run Editor Menu Window Help
Resource
Project Explorer
GenerationExample
test
backup_distributedBallAndPlate.cogedemsPrj
ECU RTOS BUS
Project Architecture Mapping Generation backup_distributedBallAndPlate.cogedemsPrj
Load configuration
Add

```
ECU
  ecu_controller
    RTOS
      rtos_controller
        TASK
          task_controller
  ecu_motion
    RTOS
      rtos_motion
        TASK
          task_touch
          task_motors
BUS
  sharedBus
    ecu_controller
    ecu_motion
```

Add a RTOS

Name:
ECU: ecu_motion
Board: flex_full
Controller: PIC33FJ256MC710
RTOS: Erika
Tick: 5
Time unit: ms
Frequency (Hz): 40000000
Scheduling: FP
[✓] Multi stack?

Fig. 5. The editor used to construct the model of the execution platform.

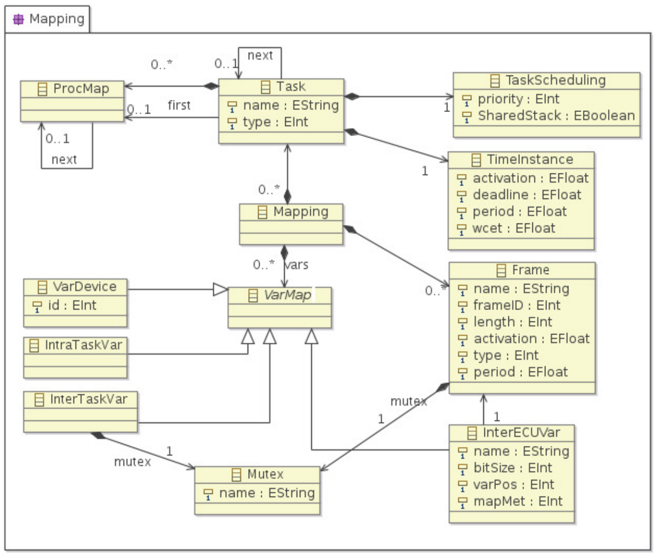


Fig. 6. Meta-model for the definition of the mapping of functional components into tasks and connections among them.

(Figure 6) and the *RTOSDeployment* (Figure 4). The Step methods of the functional subsystems are executed in the context of one of the task defined in the mapping model. More precisely, a list of *ProcMaps*, sorted according to an execution order, belongs to each task. Each *ProcMap* refers to the appointed *Proc*'s method (the connection between *ProcMap* in Figure 2 and *Method* in Figure 6 is not shown). Moreover, designers must specify all the information concerning task times (such as period and deadline) and scheduling (priority and reserved stack). The mapping of the functional subsystems (*Procs*) into tasks is subject to constraints and validation.

VarMap, the second important class of the Mapping package, concerns the mapping of *Vars* (representing communication or I/O links, depending on the mapping). As shown in Figure 6, *VarMap* is a generic interface and four possible derived classes are instantiable: *VarDevice* is used to map custom devices to real devices and the other three to model all the possible communication scenarios according to the placement of the methods of *Procs* into tasks. *IntraTaskVar* communication takes place when two communicating procs are mapped into the same task (implemented using variables local to the task). *InterTaskVar* communication occurs between two tasks on the same *Core*. In this case, a suitable protection mechanisms for shared resources, provided by the operating system (part of the platform model) is used. The most complex case happens when the communication needs to be established between two remote tasks executing on different cores, or processors, connected by either an intra-chip network, a field bus or another network. An *InterECUVar* mapping object is automatically generated to represent this kind of communication, linking a *Var* to a specified portion of a *Frame*. *Frames* are periodic communication messages which are exchanged between two nodes through a shared bus.

III. Code Generation process

Clearly, given the variety of the available hardware and basic software standards, it is impractical for us to define library objects for all the possible execution platforms. Currently, the generation of the code is focused on the OSEK/VDX (now AUTOSAR) operating system standard [33] (the library is of course extensible, as long as the definitions comply with the presented meta-model). The generator uses the standard OSEK objects (e.g. task, alarm, counter and mutex) and supports the OSEK/AUTOSAR API.

In this section, we provide information on the main Acceleo code generation templates for the system configuration, tasks, communications, and I/O code.

A. System configuration

In OSEK, the OIL (OSEK Implementation Language) [34] declarations are used to configure the operating system structures and objects according to the tasks and the shared resources. OSEK is a standard for single-processor systems. Therefore, the code generation process will need to generate one set of OIL declarations for each core. The OIL description of an OSEK application consists of a set of OIL objects, characterized by attributes and references which can be divided in three macro areas: operating system, task and resource definitions. Figure 7 shows the code generation script that is used to generate the first part, concerning the operating system configuration, which specifies:

- Locations of the header files;
- Source files, for creating the binary executable file. More precisely:
  - Functional files, containing the functional subsystem methods (previously generated by RTW/EC from the Simulink model);
  - Driver files, containing the custom device drivers;
  - Task files, containing the task settings and their bodies with the invocations of the functional methods and the middleware read/write operations;
  - Buses files, containing the configuration of the communication stacks of all the buses. Moreover, this file configures the middleware tasks for the implementation of the remote communication;
- Information about boards, controllers and cores, and the configuration of the scheduler for each core.

In the task configuration part of the OIL file, shown in Figure 8, periodic tasks are declared. A unique shared counter is defined as a base for all the alarms, which in turn provide the activation of the periodic tasks. For each task, a dedicated alarm object is defined. The task declaration specifies its (static) priority, preemptability, stack memory requirements and the resources it uses. For each frame, a middleware task and its relative alarm are added to handle the communication.

Another set of declarations is used to define the required shared resources for each node as shown in Figure 9. The OSEK resource in question corresponds to an immediate priority ceiling (pseudo) semaphore protecting the critical sections (identified by a pair of matching calls: *GetResource* and *ReleaseResource*). Shared resources are generated for all the communications that occur between functional subsystems mapped onto different tasks belonging to the same operating system. Additional Resources are needed to protect communication buffers as they are the interface between the application and the communication middleware for the objects mapped into network messages.

```
OS myOs {
 /* Library includes */
[for (lib: LibraryLinker | cg.libraries)]
 CFLAGS = "-I[lib.sourcePath/]/inc";
[/for]
 /* Functional code includes */
[for (task: Task | cg.mapping.tasks)]
 [if (task.location=rtos)]
  [for (procMap: ProcMap | task.procs)]
   [let proc : Proc = procMap.proc.owner]
 CFLAGS = "-I[cg.functional.sourcePath/]/"
          "[cg.functional.prefix/]"
          "[proc.name/][cg.functional.postfix/]";
   [/let]
  [/for]
 [/if]
[/for]
 /* Boards */
[for (board : Board | ecu.ecu.boards)]
 [if (board.name='flex_motionboard')]
 EE_OPT = "__USE_MOTIONBOARD__";
  BOARD_DATA = EE_FLEX {
   USELEDS = TRUE;
  };
 [/if]
[/for]
 /* Controllers and cores */
[for (board : Board | ecu.ecu.boards)]
 [for (controller : Controller | board.controllers)]
  [if (rtos.location.name.strcmp(controller.name)=0)]
   [for (core: Core | controller.cores)]
 MCU_DATA =  [core.name/] {
  MODEL = [controller.name/];
 };
 CPU_DATA = [core.name/] {
    [for (var : VarMap | cg.mapping.vars)]
     [if (var.eClass().name.strcmp('VarDevice')=0)]
      [let varDevice : VarDevice = var]
       [if (varDevice.ecu=ecu)]
        [let lnk : LibraryLinker = varDevice.device.
              eContainer().eContainer().eContainer()]
  APP_SRC = "[lnk.sourcePath/]/src/"
            "[varDevice.device.referenceName/].c";
    [/let][/if][/let][/if][/for]
    [for (bus : Bus | cg.architecture.busses)]
     [for (node: Node | bus.node)]
      [if (node.ecu=ecu)]
       [let lnk : LibraryLinker = node.device.
              eContainer().eContainer().eContainer()]
  APP_SRC = "[lnk.sourcePath/]/src/"
            "[node.device.referenceName/].c";
       [/let]
      [/if]
     [/for]
    [/for]
  APP_SRC = "[rtos.instanceName/]_tasks.c";
  APP_SRC = "[rtos.instanceName/]_main.c";
  APP_SRC = "[rtos.instanceName/]_vars.c";
  APP_SRC = "[rtos.instanceName/]_bus.c";
    [for (task: Task | cg.mapping.tasks)]
     [if (task.location=rtos)]
      [for (procMap: ProcMap | task.procs)]
       [let proc : Proc = procMap.proc.owner]
        [for (filename: String | proc.sourceFiles)]
  APP_SRC = "[cg.functional.sourcePath/]/[filename/]";
        [/for]
       [/let]
      [/for]
     [/if]
    [/for]
    [if (rtos.multistack)]
  MULTI_STACK = TRUE;
    [else]
  MULTI_STACK = FALSE;
    [/if]
 };
   [/for]
  [/if]
 [/for]
[/for]
 KERNEL_TYPE = [rtos.schedPolicy.toString()/];
};
```

Fig. 7. Part of the Acceleo script for generating the kernel configuration of the OSEK/OIL file.

```
 /* COUNTERS */
 COUNTER SHARED_TASK_COUNTER;
 /* TASKS and ALARMS */
[for (task : Task | cg.mapping.tasks)]
 [if (task.location.toString().strcmp(rtos.toString())=0)]
 TASK [task.name/] {
  PRIORITY = [task.scheduling.priority/];
  [if (task.scheduling.SharedStack=true)]
  STACK = SHARED;
  [/if]
  SCHEDULE = FULL;
  [for (varMap : VarMap | cg.mapping.vars)]
   [if (varMap.eClass().name='InterTaskVar')]
    [let var : InterTaskVar = varMap]
  RESOURCE = "[var.mutex.name/]";
    [/let]
   [/if]
  [/for]
  [for (frame : Frame | cg.mapping.frames)]
   [if ((frame.from.ecu=ecu) or (frame.to.ecu=ecu))]
  RESOURCE = "[frame.mutex.name/]";
   [/if]
  [/for]
 };
 ALARM ALARM_[task.name/] {
  COUNTER = "SHARED_TASK_COUNTER";
  ACTION = ACTIVATETASK { TASK =  "[task.name/]"; };
 };
[for (frame : Frame | cg.mapping.frames)]
 [if ((frame.from.ecu=ecu) or (frame.to.ecu=ecu))]
 TASK TASK_FRAME_[frame.name/] {
  PRIORITY = 1;
  SCHEDULE = FULL;
  RESOURCE = "[frame.mutex.name/]";
 };
 ALARM ALARM_FRAME_[frame.name/] {
  COUNTER = "SHARED_TASK_COUNTER";
  ACTION = ACTIVATETASK { TASK =
                     "TASK_FRAME_[frame.name/]"; };
 };
 [/if]
[/for]
```

Fig. 8. Part of the Acceleo script for generating the ALARM and TASK objects of the OSEK/OIL file.

```
 /* RESOURCES */
[for (varMap : VarMap | cg.mapping.vars)]
 [if (varMap.eClass().name='InterTaskVar')]
  [let var : InterTaskVar = varMap]
 RESOURCE [var.mutex.name/] {
 RESOURCEPROPERTY = STANDARD;
 };
  [/let]
 [/if]
[/for]
[for (frame : Frame | cg.mapping.frames)]
 [if ((frame.from.ecu=ecu) or (frame.to.ecu=ecu))]
 RESOURCE [frame.mutex.name/] {
  RESOURCEPROPERTY = STANDARD;
 };
 [/if]
[/for]
```

Fig. 9. Part of the Acceleo script for generating the RESOURCE objects of the OSEK/OIL file.

OIL declarations are complemented by the corresponding application configuration files. As shown in Figure 10, this configuration code contains the timer interrupt handler, the hardware configuration function and the program entry point (*main* function) for each core. The system configuration is implemented by the function *system_init*, which invokes the initialization function for all the buses to which the node is connected and for all the ECUs. The *main* function calls the initializing functions, setting the system base rate and configuring each alarm to activate the corresponding periodic task. Finally, it enters an infinite loop.

```
/* System Timer Interrupt */
ISR2(_T1Interrupt)
{
 clean_T1_interrupt_flag();
 CounterTick(SHARED_TASK_COUNTER);
}

/* System Initialization Function */
void system_init(void)
{
 /* ------ Network components initialization ------ */
[for (bus : Bus | cg.architecture.busses)]
 [for (node: Node | bus.node)]
  [if (node.ecu=ecu)]
   [bus.name/]_init();
  [/if]
 [/for]
[/for]
 /* ------------- Procs ------------ */
[for (task : Task | cg.mapping.tasks)]
 [if (task.location.toString().strcmp(rtos.toString())=0)]
  [for (procMap : ProcMap | task.procs)]
   [for (method : Method | procMap.proc.owner.methods)]
    [if (method.initializer=true)]
 [method.name /]();
    [/if]
   [/for]
  [/for]
 [/if]
[/for]
}

/* Main function */
int main(void)
{
 /* Initialize system */
 system_set_timer(SYSTEM_KHZ);
 system_timer_init();
 system_init();
 /* Start periodic tasks */
 tasks_start();
 /* Background activity here */
 for (;;);
 return 0;
}
```

Fig. 10. Script for the timer interrupt, initialization code and task activation.

B. Tasks

The tasks code is generated as a sequence of calls to the step methods of the functional subsystems mapped into it, as shown in Figure 11. Calls are sequentialized according to the mapping order, which must be consistent with the partial order of execution specified by the semantics of the functional model. Each invocation of the *step* method is preceded/followed by the middleware read/write functions implementing the functional data flows.

Tasks activations are performed by the RTOS, as configured through the OSEK system call *SetRealAlarm* by the *tasks_start* function.

```
/* TASKs Definition */
[for (task : Task | cg.mapping.tasks)]
 [if (task.location.toString().strcmp(rtos.toString())=0)]
TASK([task.name/])
{
  [for (procMap : ProcMap | task.procs)]
 //management of the method: [procMap.proc.name/]
   [for (methodRef : MethodRef | cg.functional.methodRefs)]
    [if (methodRef.called=procMap.proc)]
     [for (varMap : VarMap | cg.mapping.vars)]
      [if (varMap.var=methodRef.caller.owner)]
 resolve_READ_CONNECTION_[methodRef.caller.owner.name/](0);
      [/if]
     [/for]
    [/if]
   [/for]
 //method of [procMap.proc.owner.name/];
 [procMap.proc.name/]();
   [for (methodRef : MethodRef | cg.functional.methodRefs)]
    [if (methodRef.caller=procMap.proc)]
     [for (varMap : VarMap | cg.mapping.vars)]
      [if (varMap.var=methodRef.called.owner)]
 resolve_WRITE_CONNECTION_[methodRef.called.owner.name/](0);
      [/if]
     [/for]
    [/if]
   [/for]
  [/for]
}
 [/if]
[/for]

/* Task Initialization Function */
void tasks_start(void)
{
[for (task : Task | cg.mapping.tasks)]
 [if (task.location.toString().strcmp(rtos.toString())=0)]
 SetRelAlarm(ALARM_[task.name/],
  (long int)[task.timeInstance.activation.round()/]
  *SYSTEM_TICK_MS,
  (long int)[task.timeInstance.period.round()/]
  *SYSTEM_TICK_MS);
 [/if]
[/for]
}
```

Fig. 11. Acceleo script for generating tasks' body and task configuration.

C. Buses

The code implementing the distributed communication, for which the generation template is (partly) shown in Figure 12, is divided into two main scopes: communication management and initialization.

Communications among nodes are handled through a middleware layer that hides the details of the physical communication link. Functional communication is mapped into Frames. Each frame makes use of a shared memory buffer (protected by a mutex) and a periodic task which either sends or receives data (the operation depends on the direction of the link).

Communication initialization is achieved generating an initialization method ”*(bus name)*_init” for each bus used by the ECU, which configures all the transceivers accessing the bus. The same function initializes the frame data structures and configures the periodic middleware tasks which forwards frames through the network.

D. Communication and synchronization (Glue code)

The code implementation of the middleware read/write functions, representing the communication links (*Vars*), depends on the task and core mapping of the blocks at the two endpoints of the communication.

```
[for (frame : Frame | cg.mapping.frames)]
 [if ((frame.from.ecu=ecu) or (frame.to.ecu=ecu))]
char CG_BUFFER_[frame.name/]['['/][frame.length/][']'/];

TASK(TASK_FRAME_[frame.name/]) {
 unsigned char buffer['['/]MAX_PAYLOAD_SIZE[']'/];
 memset(buffer, ' ', MAX_PAYLOAD_SIZE);
  [if (frame.from.ecu=ecu)]
 GetResource([frame.mutex.name/]);
 memcpy (buffer,&(CG_BUFFER_[frame.name/]),[frame.length/]);
 ReleaseResource([frame.mutex.name/]);
   [if (frame.from.device=frame.bus.coordinator)]
 [frame.from.device.coordinator.write/]([frame.from.id/],
       [frame.frameID/], [frame.to.address/], buffer, 0);
   [else]
 [frame.from.device.device.write/]([frame.from.id/],
       [frame.frameID/], [frame.to.address/], buffer, 0);
   [/if]
  [else]
 int from;
   [if (frame.from.device=frame.bus.coordinator)]
 [frame.from.device.coordinator.read/]([frame.to.id/],
                       [frame.frameID/], &from, buffer);
   [else]
 [frame.from.device.device.read/]([frame.to.id/],
                       [frame.frameID/], &from, buffer);
   [/if]
 GetResource([frame.mutex.name/]);
 memcpy (&(CG_BUFFER_[frame.name/]), buffer, [frame.length/]);
 ReleaseResource([frame.mutex.name/]);
  [/if]
}
 [/if][/for]
//Bus management
[for (bus : Bus | cg.architecture.busses)]
int8_t [bus.name/]_init (int device) {
 //Devices initialization
 radio_conf_t conf;
 conf.pan_id = [bus.ID/];
 conf.channel = [bus.channel/];
 [for (node: Node | bus.node)]
  [if (node.ecu=ecu)]
 conf.device_addr = [node.address/];
 addr_t addrs['[]'/] = {[node.address/]
   [for (_node : Node | bus.node)]
    [if (_node<>node)], [_node.address/][/if][/for]};
   [if (bus.coordinator=node)]
 if([node.device.coordinator._init/ (
  [node.device.referenceName/], &conf, &addrs['[0]'/])<0)
  return -1;
   [else]
 if([node.device.device._init/](
              [node.device.referenceName/], &conf, 0)<0)
  return -1;
   [/if][/if][/for]
 [for (frame : Frame | cg.mapping.frames)]
  [if ((frame.bus=bus) and
           ((frame.from.ecu=ecu) or (frame.to.ecu=ecu)))]
 memset(CG_BUFFER_[frame.name/], ' ', [frame.length/]);
   [if (frame.to.ecu=ecu)]
 SetRelAlarm(ALARM_FRAME_[frame.name/], SYSTEM_TICK_MS*
    [frame.activation/], ([frame.period/]*SYSTEM_TICK_MS)/2);
   [else]
 SetRelAlarm(ALARM_FRAME_[frame.name/], SYSTEM_TICK_MS*
     [frame.activation/], [frame.period/]*SYSTEM_TICK_MS);
   [/if][/if][/for]
 return 0;
}
[/for]
```

Fig. 12. Acceleo script for generating bus and communication management.

For each communication Var connecting a data signal element mapped on the system platform, the Acceleo script, reported in Figure 13, can generate three different implementations. When the Var mapping is an instance of *IntraTaskVar*, the access functions execute simple read/write operations to a shared variable. If the endpoints are located on different tasks managed by the same operating system (*InterTaskVar*), read and write accesses to the shared variable are protected by instructions to get/release the relative Resource to prevent race conditions. The last case, which occurs when the sender and the receiver are on different processors (*InterECUVar*), is handled by reading/writing the assigned area of the message frame buffer. To avoid race conditions when accessing the middleware frame buffer, memory accesses are protected using a middleware resource.

```
[for (varMap: VarMap | cg.mapping.vars)]
        [if (varMap.eClass().name='IntraTaskVar')]
                [let var: IntraTaskVar = varMap]
[varMap.var.type.name/]
         __communication_buffer_for_[varMap.var.name/]__;
inline void resolve_READ_CONNECTION_[var.var.name/] (
                             [var.var.type.name/]* arg) {
        __communication_buffer_for_[var.var.name/]__ =
                                      [varMap.var.from/];
}
inline void resolve_WRITE_CONNECTION_[var.var.name/] (
                             [var.var.type.name/]* arg) {
        [var.var.to/] =
            __communication_buffer_for_[var.var.name/]__;
}
                [/let]
        [/if]
        [if (varMap.eClass().name='InterTaskVar')]
                [let var: InterTaskVar = varMap]
[varMap.var.type.name/]
         __communication_buffer_for_[varMap.var.name/]__;

inline void resolve_READ_CONNECTION_[var.var.name/](
                             [var.var.type.name/]* arg) {
        GetResource([var.mutex.name/]);
        __communication_buffer_for_[var.var.name/]__ =
                                       [varMap.var.from/];
        ReleaseResource([var.mutex.name/]);
}
inline void resolve_WRITE_CONNECTION_[var.var.name/](
                              [var.var.type.name/]* arg) {
        GetResource([var.mutex.name/]);
        [var.var.to/] =
             __communication_buffer_for_[var.var.name/]__;
        ReleaseResource([var.mutex.name/]);
}
                [/let]
        [/if]
        [if (varMap.eClass().name='InterECUVar')]
                [let var: InterECUVar = varMap]
extern char CG_BUFFER_[var.frames.name/]['[]'/];
                        [if (var.frames.to.ecu=ecu)]
inline void resolve_READ_CONNECTION_[var.var.name/](
                              [var.var.type.name/]* arg) {
        GetResource([var.frames.mutex.name/]);
        memcpy (&([var.var.to/]),
     &(CG_BUFFER_[var.frames.name/]['['/][var.varPos/]
     [']'/]),sizeof([varMap.var.type.name/]));
        ReleaseResource([var.frames.mutex.name/]);
}
                        [else]
inline void resolve_WRITE_CONNECTION_[var.var.name/](
                              [var.var.type.name/]* arg) {
        GetResource([var.frames.mutex.name/]);
        memcpy (&(CG_BUFFER_[var.frames.name/]
                              ['['/][var.varPos/][']'/]),
      &([var.var.from/]), sizeof([varMap.var.type.name/]));
        ReleaseResource([var.frames.mutex.name/]);
}
                        [/if]  [/let]  [/if]
[/for]
```

Fig. 13. Acceleo script for generating middle ware read/write functions.

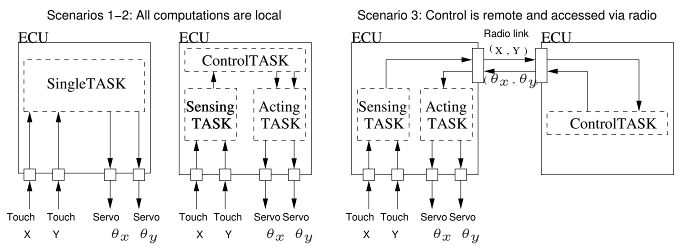

Scenarios 1–2: All computations are local
Scenario 3: Control is remote and accessed via radio
ECU
SingleTASK
ControlTASK
Sensing TASK
Acting TASK
Radio link
(X , Y)
(θ_x , θ_y)
Touch X
Touch Y
Servo θ_x
Servo θ_y

Fig. 15. The three mapping configurations for our case study.

IV. Generation Example

In order to show the capability of remapping a functional model into different execution platforms offered by our framework, an example consisting of a ball-and-plate control application is introduced. A ball is placed on top of the plate and the objective of the controller is to keep the ball as close as possible to the plate center. Overall, the sensor and actuation structure is composed by a touch screen placed on the plate and two servomotors to control the pitch and roll angles.

A. Simulink model

The functional model of the control was first developed without any reference to the execution platform, simulated and validated according to the physical model of the system. (Figure 14). The Simulink model contains the following five subsystems:

- *subTouch*, containing the touch screen sensor block;
- *subController*, implementing the PID controller, tuned to be executed every 50ms;
- *subServoX*, containing the servomotor block in charge of handling the X-axis;
- *subServoY*, containing the servomotor block in charge of handling the Y-axis;
- *Plant*, implementing the dynamics of the real system.

Once the model is validated through simulations in the Simulink environment, the code for the four subsystems (all except the Plant) is generated using the Simulink Coder (Figure 14). Then, the model is parsed and the EMF functional model is constructed, creating four procs: *subTouch*, *SubController*, *subServoX* and *subServoY*. This step is labeled as (1) in Figure 1.

B. Execution platform configurations from library components

Three different execution platforms have been defined for our case study (Figure 15), corresponding to centralized and distributed implementations of the control application.

The first and second configurations run on a single ECU and the operations are coded into one or three tasks, respectively. In the third configuration, the application runs on two ECUs, connected by a radio link. Sensing and actuation are performed on one ECU, while the control is executed remotely.

The platform configurations are composed in the three cases using components taken out of a library. In this case, the required hardware and software resources are composed by the Flex boards and the Erika Enterprise RTOS [35], implementing the OSEK/VDK standard.

The Flex boards included in the library are:

- *Flex light*, mounting a Microchip dsPIC33FJ256MC710 controller which hosts a dsPIC33 core and offers a wide set of peripherals (9 timers, 85 digital I/O, 8 PWM channels, 2 ADCs and 2 SPIs);
- *Motion board*, pluggable on top of the Flex light with drivers for two servo motors, a touch screen, and an 802.15.4 transceiver (a cc2420, connected via SPI).

An ECU, referred as *Motion ECU*, is created by connecting a Flex light and a Motion board together.

The API used to manage the physical devices (touch screen and servo motor), is based on the AUTOSAR standard. For each device, the associated functions allow initialization, reading and writing. The $\mu Wireless$ communication protocol is used for node-to-node communications. The protocol allows concurrent access with possible collisions and message loss. In Figure 1, this procedure is denoted as (2).

C. Mapping models

The first configuration consists of a Motion ECU, running a single periodic task calling the Step methods of subTouch, subController, subServoX and subServoY, in order. The task period is equal to the control period ($50ms$).

The second configuration consists of a Motion ECU providing access to the custom devices using three tasks. The first task runs the Step method of the subController. The Step methods of the actuator blocks (subServoX and subServoY) are executed by the second task and the step method of subTouch is invoked by the remaining task. All tasks are executed with a period equal to 50ms.

The distributed scenario, which contains two Motion ECUs, analyzes the effect of the communication jitter and packet losses. On each ECU, tasks execute with a period of $50ms$. The subsystem implementing the control is mapped on one ECU and the I/O subsystems on the other. The generation process creates four read and write function pairs, implemented as accesses to two frames sent over the wireless communication channel. The ball position (X and Y) is placed in the frame sent to the controller. The desired angles for the plate motors defined by the controller, are mapped into a reply frame. Overall, not a single line of C code has been written in all cases and the Simulink model was unchanged every time it was redeployed into a new architecture. The mapping phase corresponds to stage (3) in Figure 1.

D. Performance evaluation

Figure 15 shows the measured error in the position of the ball with the three execution platform (and mapping) options starting from an initial offset of 12cm from the center, and compares it with the results of a zero-delay Simulink model. Errors are expressed in terms of distance (meters) between the plate center and the actual ball position.

When all subsystems are local and mapped into a single task, the performance is very close to the simulation data. The 3-tasks model introduces additional sampling delays and a slower dynamics. The distributed architecture presents an even worse performance as bringing the ball to the center takes a longer time. Moreover, the system suffers of marked oscillations due to communication jitters and packet losses. Although the fourth step in Figure 1 concerns mostly an offline analysis of the PSM model, also the online evaluation of the performance can be considered as part of step (4).

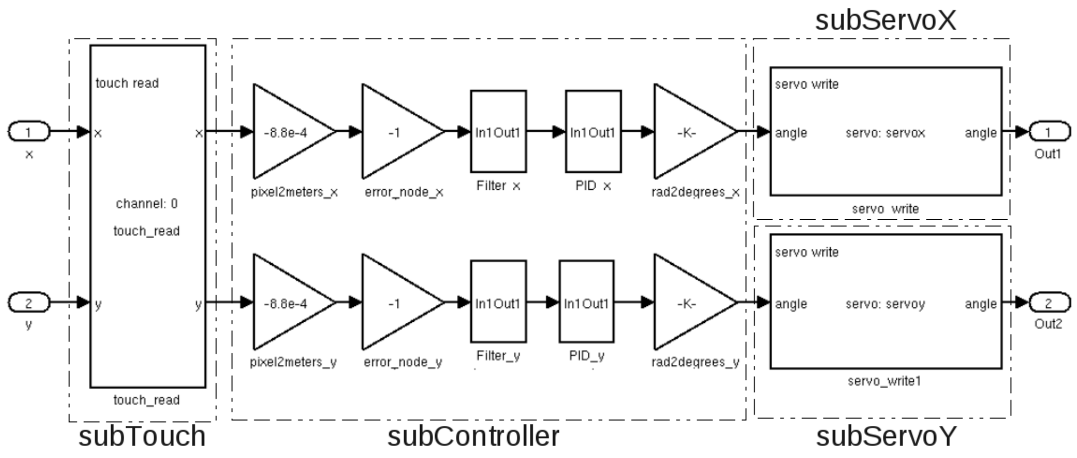


Fig. 14. Simulink subsystems whose behavioral code is generated by Simulink Coder.

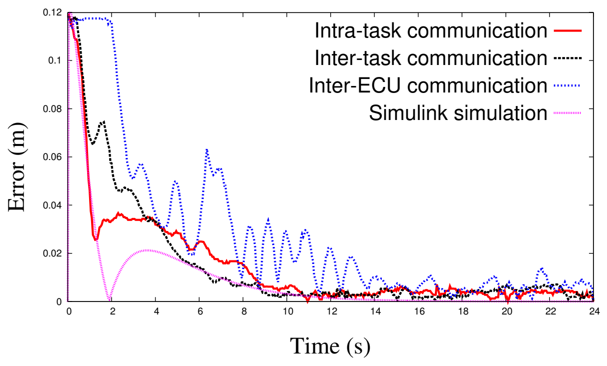


Fig. 16. Ball errors of the proposed solutions with respect to time.

V. Conclusion

This paper presents a framework for the modeling of execution platforms and the mapping of MBD-developed functionalities on them. The framework assists in the automatic generation of tasks and middleware layer that implements the communication and synchronization activities. Future developments will include the capability of estimating communication and scheduling delays and back-annotate the Simulink model with them, in order to enhance the simulation effectiveness and the accuracy of the performance prediction.

References

[1] P. J. M. G. Nicolescu, *Model Based Design for Embedded Systems*, ser. Computational Analysis, Synthesis and Design of Dynamic Systems. CRC Press, 2009.
[2] "Simulink," http://www.mathworks.it/products/simulink/index.html.
[3] "Scicos," http://www.scicos.org/.
[4] "Scade," http://www.esterel-technologies.com/products/scade-suite/.
[5] "Model driven architecture," http://www.omg.org/mda/.
[6] "The object management group (omg)," http://www.omg.org/.
[7] "The unified modeling language (uml)," http://www.uml.org/.
[8] "The systems modeling language (sysml)," http://www.omgsysml.org/.
[9] "Eclipse modeling framework (emf)," http://www.eclipse.org/modeling/emf/.
[10] "Acceleo," http://www.acceleo.org/.
[11] "Real-time workshop/embedded coder," http://www.mathworks.com/products/matlab-coder/index.html.
[12] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, pp. 25–31, 2006.
[13] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, Jan. 2003.
[14] Y. Vanderperren and W. Dehaene, "From uml/sysml to matlab/simulink: current state and future perspectives," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, ser. DATE '06, pp. 93–93.
[15] "Esl now survey," http://www.esl-now.com, 2005.
[16] E. W. . Celoxica, "Survey of system design trends 2005," http://www.celoxica.com, 2005.
[17] "Matlab product web page," http://www.mathworks.com/products/matlab/.
[18] B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers, "A methodology to design programmable embedded systems - the y-chart approach," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, 2002.
[19] A. Sangiovanni-Vincentelli, "Quo vadis sld: Reasoning about trends and challenges of system-level design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, March 2007.
[20] "Autosar, specifications 4.0," http://www.autosar.org/, 2010.
[21] J. Martinez, P. Merino, and A. Salmeron, "Applying mde methodologies to design communication protocols for distributed systems," in *Proceedings of the First International Conference on Complex, Intelligent and Software Intensive Systems*, 2007.
[22] "Marte," http://www.omgmarte.org/.
[23] "East architecture description language (adl)," http://www.east-adl.info/.
[24] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits, "Composition and cloning in modeling and meta-modeling," *IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling*, vol. 12, pp. 263–278, 2004.
[25] F. Balarin, L. Lavagno, C. Passerone, and Y. Watanabe, "Processes, interfaces and platforms. embedded software modeling in metropolis," in *Proceedings of the Second International Conference on Embedded Software*, ser. EMSOFT '02, 2002, pp. 407–416.
[26] "The eclipse modeling project," http://www.eclipse.org/modeling/.
[27] ""atl transformation language"," http://www.eclipse.org/atl/.
[28] G. Raghav, S. Gopalswamy, K. Radhakrishnan, J. Delange, and J. Hugues, "Model based code generation for distributed embedded systems," in *Proceedings of the 2010 on Embedded Real Time Software and Systems (ERTSS'10)*, 05 2010.
[29] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the prototype to the final embedded system using the ocarina aadl tool suite," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 4, pp. 42:1–42:25, Aug. 2008.
[30] "The timmo2use project web site," http://timmo-2-use.org/.
[31] "The gene-auto project web site," http://geneauto.gforge.enseeiht.fr/.
[32] "The project p web site," http://www.open-do.org/projects/p/.
[33] "Osek-vdx standard," http://www.osek-vdx.org/.
[34] "Osek-vdx implementation language," http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf.
[35] "Erika enterprise rtos," http://erika.tuxfamily.org/.