An MDE Approach for the Design of Platform-Aware Controls in Performance-Sensitive Applications

Matteo Morelli
Scuola Superiore S. Anna, Pisa

Marco Di Natale
Scuola Superiore S. Anna, Pisa

***Abstract*—Model-Based Design is widely adopted in control domains for the early validation of systems properties using simulation or formal verification and the possibility of automatic generation of code. Most tools used in the industrial practice allow for the representation of the controller functionality abstracted from the implementation details. These models may be inaccurate in those cases in which computation and communication delays affect the performance of the controls. To address this problem, we propose a Model-Driven approach in which a Simulink functional model of controls is matched to a model of the execution platform through a mapping model, representing the implementation as a set of tasks and messages. The platform and the implementation are modeled in SysML/MARTE and are used to automatically generate a new Simulink model with an additional set of blocks representing the execution time of the tasks running under the control of a selected scheduler. Acceleo and QVTo model-to-text and model-to-model standard transformation languages are used to automatically generate the intermediate models, the task and scheduler blocks.**

I. Introduction

The use of models for the advance analysis of the system properties and verification by simulation, the documentation of the design decisions and the automatic generation of the software implementation is an industrial reality. In the Model-Based Design approach (MBD) models are based on a formal (synchronous reactive) execution model. Examples of available commercial tools are Simulink [1] and SCADE [2]. These tools allow the modeling of continuous-, discrete-time, and hybrid systems. They allow verifying the system functionality against a dynamic model of the controlled system (plant), but lack the capability of modeling physical computing architectures (and to some degree tasks and resources), as well as computation and communication delays that depend on the platform.

Model-Driven Engineering (MDE) and Architecture Description (ADL) languages are very good at representing architectural aspects and are designed for being easily extended. Also, they typically provide mechanisms to transform models expressed in a language into another. MDE and ADL languages may support the modeling of the execution platform [3], but the tools supporting these languages seldom allow for simulation and the automatic generation of the behavioral code (with some exceptions).

When the communication and computation delays arising from the platform implementation may affect the behavior of the controls it is highly desirable to be able to assess this impact at simulation time on a virtual model of the controller, the plant and the software implementation of the system. The analysis of computation and communication delays can be performed using the Truetime blockset in Simulink. However, this solution creates a model in which the functional solution is interspersed with platform-specific implementation blocks. If the implementation platform, or the task placement, or in general the task configuration is modified, the user must change all the affected blocks inside the model.

In our work, we provide:

A *framework and a methodology* for importing the functional definition of the controller part from Simulink into the Eclipse Modeling Framework (EMF - based on a custom metamodel expressing the execution constrainst of the Simulink model of execution) and model-to-model transformation rules to translate the imported EMF model into a SysML model in Papyrus.

A *set of customized profiles* for the modeling of execution platforms and task implementations in SysML/MARTE. A methodology for mapping the functional model into the execution platform components and defining the task and message model. *Model-to-code transformation rules* to automatically generate a set of Matlab-language files that extend the Simulink model and backannotate it with blocks representing the execution of the Simulink controller model into a set of real time tasks, with finite execution times, under the control of a scheduler.

A *modular representation of tasks and schedulers in Simulink* using a set of custom blocks and a framework that allows co-simulating real-time scheduling together with the hybrid Simulink models of the controller and the plant.

We show the application of our framework to the design of a concurrent, three task implementation of electric engine PID controls in Simulink. The example will show how the design of the functional model and the platform implementation can be kept separated and how automation allows the automatic generation of the simulation models that allow evaluating the impact of different task designs or different scheduler selections.

State of the Art

In the model of complex (cyber-physical) systems, separation of the functional and platform models is advocated by many. Examples from the academia are the Y-chart [7] and the Platform-based design (an implementation in the Metro II tool [5]) approaches. The OMG (a standardization organization) in its Model-Driven Architecture (MDA)

[6] defines a three stage process in which a Platform-Independent Model or PIM is transformed in a Platform-Specific Model or PSM by means of a Platform Definition Model (PDM). Finally, the automotive industry AUTOSAR standard [4] defines a virtual integration environment for platform-independent software components and a separate model for the (distributed) execution architecture, later merged in a deployment stage (supported by tools). The TIMMO/TIMMO2 [8] projects focus on the modeling infrastructure and the capability of modeling timed events in AUTOSAR. Unfortunately, AUTOSAR does not have a formal model for the behavior of the functions and especially the dynamics of the plant. Therefore, an external tool or the actual code is needed for functional modeling and simulation. Raghav et al. [9] and Hugues et al. [10] proposed two 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.

The development of a platform model for (large and distributed) embedded systems and the modeling of concurrent systems with resource managers (schedulers) requires domain-specific concepts. The OMG MARTE [11] standard is general, rooted on UML/SyML and supported by several tools. MARTE has been applied to several use cases, including automotive projects [12]. GeneAuto [13], ProjectP [14], the Rubus Component Model [15] and AADL [16] put emphasis on the modeling of task sets and their interactions and the code generation infrastructure, without including simulation capabilities or an explicit formal metamodel for the internal behavior of tasks. The BIP framework [17] is another example of a formal modeling methodology for the verification of timing properties.

Several authors [18] acknowledge that future trends in model engineering will encompass the definition of integrated design flows exploiting complementarities between UML or SysML and Matlab/Simulink, although the combination of the two models is affected by the fact that Simulink lacks a publicly accessible meta-model [18].

A very large number of projects target the evaluation of scheduling policies and the analysis of task implementations (more than 6 million hits when searching the keywords *real time scheduling simulator* in Google). A necessarily incomplete list includes Yartiss [19], Storm [20], ARTISST [21], Cheddar [22], Stress [23].

TrueTime [24] is a freeware Matlab/Simulink-based simulation tool that has been developed at Lund University since 1999. It provides models of multi-tasking real-time kernels and networks that can be used in simulation models for networked embedded control systems and study the (simulated) impact of lateness and deadline misses on controls. The TrueTime Kernel block simulates a *computer node* with a generic real-time kernel, A/D and D/A converters, external interrupt inputs and network interfaces. The block is configured via an initialization script to create tasks, timers and interrupt handlers and define the scheduling policy and the communication resources. Because of the monolithic architecture and the number of code artifacts that are needed for system configuration, the current TrueTime implementation is hardly compatible with an automatic model generation flow.

II. Proposed Approach

In the development flow considered in our work (summarized in Figure 1), the *Simulink functional model* is the starting point. Once the simulation results are satisfactory, the designer uses the model exporter to generate an abstract view of functional model. The abstract view is an exported XML file that conforms to the Ecore meta-model for SR systems shown in Fig. 3. The Ecore view preserves all the structural properties of the Simulink model, such as the types and interfaces of the blocks and the connections among the blocks, and also accounts for the information related to the timed execution events, including rate and partial order of execution constraints.

Next, the Ecore representation of the functional Simulink model is translated using QVT scripts into a SysML model in Papyrus (leveraging a profile definition). Here, it is extended with the platform and mapping models.

The *Platform model*, as well as the model of the tasks and messages, is generated using SysML [25], [26]. A specialized profile, built on top of the OMG standard MARTE (Modeling and Analysis of Real-Time and Embedded systems) [11] profile, is used for modeling embedded platforms and systems.

The *mapping model* associates functional elements to tasks, and tasks to processing (HW) elements. Matlab code is generated from the SysML model using the Acceleo [27] open model-to-text generator. The generated code operates on the original Simulink model and adds to it a set of custom blocks (with connections), representing the implementation of the Simulink subsystems of the controller in tasks, executing under the control of a scheduler.

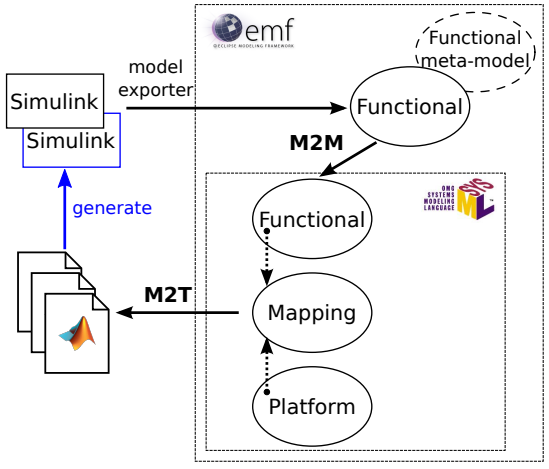

Figure 1. The development flow for the proposed Model-Driven approach.

III. Modeling Concepts

A. Functional Modeling

The functional model is created by importing in EMF a Simulink model that includes the controller part and the model of the plant. The Simulink model must comply with the restriction that there is a decomposition level in which

the controller part consists of a collection of subsystems, in which each subsystem only contains periodic blocks with the same period (each subsystem has a single rate). This may require reworking some of the existing models, possibly adding one more hierarchical level in the model decomposition.

Figure 2 shows a Simulink model in which three DC-servo systems are controlled by a PID (derived from a corresponding TrueTime example). Each DC-servo is described by a continuous-time SISO transfer function (a `TransferFcn` block), and is controlled by a dedicated discrete-time PID regulator. The three PID controllers have the same proportional, derivative and integral coefficients K_p, K_d, K_i, and are modeled as (masked) subsystem blocks. A standard `SignalGenerator` block produces the reference signal for the controllers.

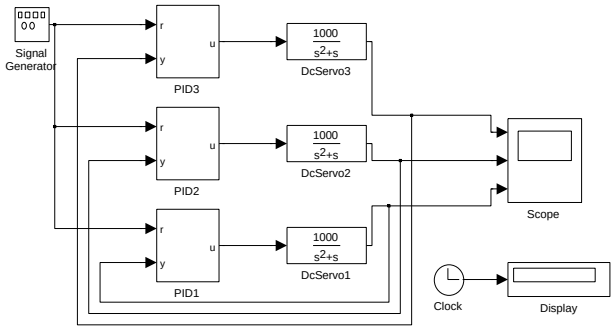

Figure 2. The application example from TrueTime [24], PID control of three DC-servo systems.

A Matlab script uses the Simulink modeling API (programming interface) to parse the model structure and export an XML view of the controller subsystems. The XML conforms to a schema created in accordance with an Eclipse Ecore metamodel, defined for representing the execution constraints that apply to the Simulink subsystems (Figure 3). This metamodel is not too dissimilar from the one proposed in the GeneAuto project [13] (actually, a simplified version of it), but contrary to GeneAuto, it is formally available as an Ecore definition.

After the functional model is imported in the Eclipse EMF framework, a QVT model-to-model transformation is executed to import the model in SysML, which conforms to the profile for SR systems in Figure 4. Generic `Block` entities are mapped to standard SysML blocks; `Subsystem` entities are mapped to `SRSubsystem` instances. The input/output ports and the corresponding connections are suitably translated from the source model (Ecore) to the destination model (SysML).

Figure 5 shows how the parts of our three servo (SysML) functional model are connected.

B. Platform Modeling in SysML

For the modeling of the physical (HW) part of the execution platform we rely on the concepts provided in the `HRM::HW_Logical` package of MARTE, and we define our own taxonomy of stereotypes for Basic Software (BSW) components and for the deployment of BSW modules onto

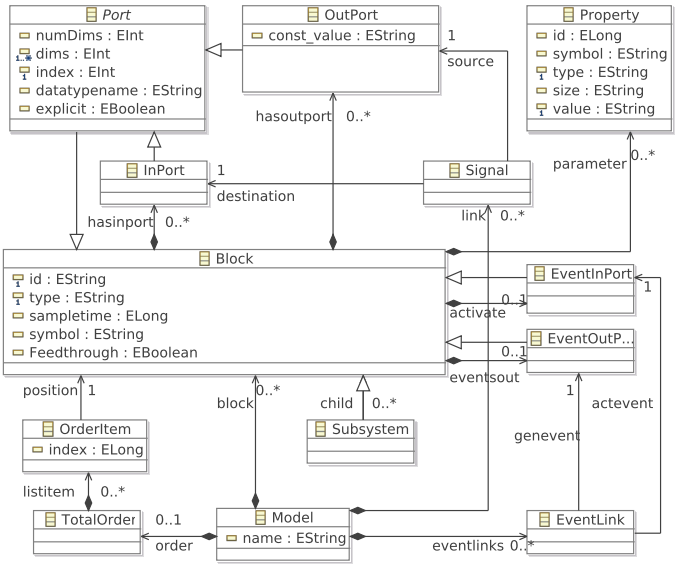

Figure 3. The Ecore meta-model for the functional part.

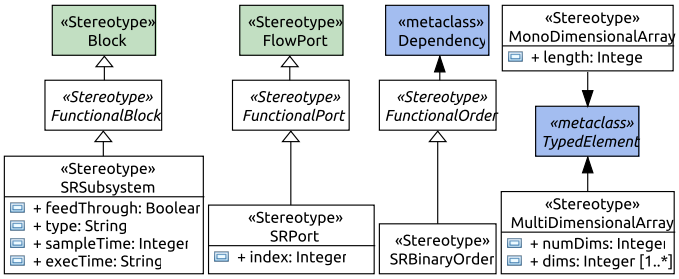

Figure 4. The SysML profile for the functional part.

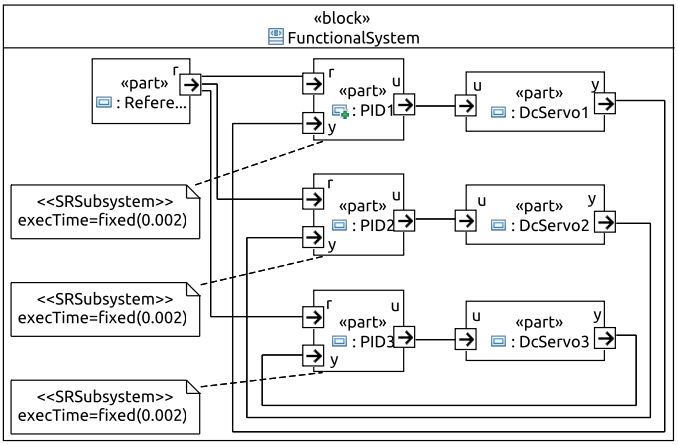

Figure 5. Structure of the functional model. Parts and their connections are *automatically generated* by the QVTo M2M transformation.

the HW. The execution-platform meta-model concepts are organized in two packages: `HwResources` and `BswRTOS`.

The `HwResources` package introduces an element representing a HW board (`HwBoard`) and other HW modeling entities, including computing, storage, communication, timing, or device resources, imported from the MARTE `HW_Logical` and its sub-packages. The `HwProcessor` stereotype provided in the `HW_Computing` sub-package matches

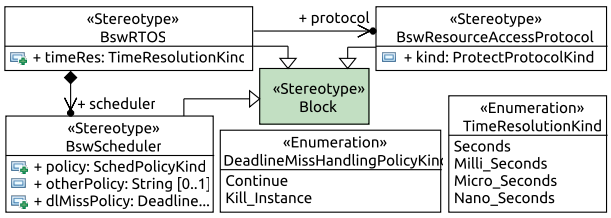

Figure 6. Structure of the `BswRTOS` meta-model (UML extensions).

the concept of CPU and provides the attribute `nbCores` to specify the number of cores, thus enabling the modeling of multi-core architectures.

The `BswRTOS` package provides a set of stereotypes to model RTOS concepts. The stereotype `BswRTOS` denotes an RTOS and inherits from the general block concept of SysML. The RTOS (kernel) contains a scheduler, denoted as `BswScheduler`, which is responsible for executing tasks according a given scheduling policy, and corresponds to the scheduling policy kind defined in the MARTE model library (`SchedPolicyKind`). The allocation relationship `BswRTOSDeployment` specifies which processor executes the RTOS.

In the case of our three-servo example, the execution platform is single-core and does not include networks. Therefore, only the CPU node that performs the computations is modeled. The example assumes that regulators' codes execute on a single-core architecture, under the control of a RTOS providing a (SW) clock resolution of the order of millisecond (`timeRes`). The RTOS scheduler operates according to the EDF scheduling policy (`policy`). Tasks trespassing their deadlines are allowed to continue their execution (`dlMissPolicy`).

C. Mapping Models

The mapping model represents the execution of functional elements by tasks and the allocation of tasks on cores. The `Concurrency` package classifies concurrent execution contexts in terms of processes and threads. The `Interaction` package defines the signal variables, implementing functional communication signals. Finally, the `Allocation` package specifies a set of dependencies that define mappings/deployments as extensions of the standard SysML Allocation concept.

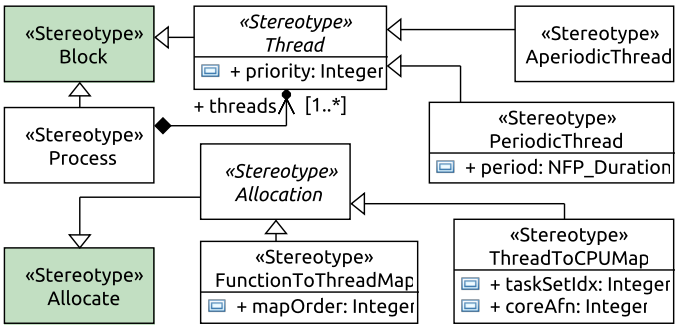

Figure 7. Structure of the `BswRTOS` meta-model (UML extensions) for use by M2M transformation tools.

There are two concepts that are central to the definition of a mapping model: *threads*, represented by the stereotype `Thread`, and *signal variables*, denoted as `ComImpl`. A `Thread` is a unit of concurrent execution that runs on one of the system cores under the control of an RTOS. Each `Thread` is contained in a `Process` and is characterized by a `priority` value. Concrete specializations of `Thread` are `AperiodicThread` and `PeriodicThread` (with its `period`).

The concept of *allocation* completes the specification of mapping meta-model. The `FunctionToThreadMap` denotes the mapping of a functional subsystem into a `Thread`. When multiple subsystems are mapped into the same `Thread`, the attribute `mapOrder` defines how the execution of their `step()` methods will be serialized in the generated thread code. The *mapping order* must be consistent with the partial order of execution imposed by the Simulink model semantics. The `ThreadToCPUMap` models the deployment of a `Thread` to an `HwProcessor`. The attribute `coreAfn` enables the binding of the thread to a physical processor core (processor affinity).

Our three-servo application example considers the case of three periodic control tasks, namely `Task1_1`, `Task2_1` and `Task3_1`, running concurrently on the CPU. Tasks have different periods, respectively equal to $6ms$, $5ms$ and $4ms$. Each task executes the PID control logics of one regulator subsystem (the i-th task, `Taski_1`, executes the i-th regulator subsystem, `PIDi`). The execution times of all subsystems are set to 2ms. The mapping model for our three servo example is shown in Figure 8. Individual application elements are associated to individual execution platform elements by means of `Allocation` instances. The `«FunctionToThreadMap»` allocation denotes the mapping of a functional subsystem into a `Thread`. In this case, the `mapOrder` attribute is set to 1 for all control tasks, since each task executes one single PID subsystem. The `«ThreadToCPUMap»` allocation models the deployment of a `Thread` onto the `HwProcessor`. The attribute `taskSetIdx` allows the designer to specify the position (index) of each task in the task-set.

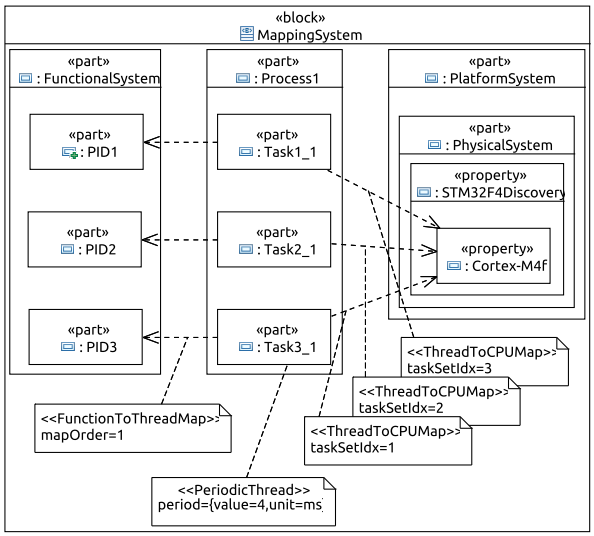

Figure 8. Mapping model.

IV. A Modular Representation of Tasks and Schedulers in Simulink

A Real-Time scheduler simulator is a Discrete Event System (DES) implementing an event handling mechanism (typically with a queue). It reacts to tasks arrival events and dispatches the currently active tasks from the ready queue according to a fixed or dynamic priority-based scheduling algorithm. Tasks arrival events can be asynchronous or periodic, and are ordered in the event queue in ascending order following the event occurrence time and the event priority. At any point in time, the next scheduling event can be the termination of the task currently in execution, or the arrival event of a task, that can possibly cause a preemption (if the new task has higher priority) and a context switch or a task dispatch. In an RT simulator, tasks execute according to a model of (time-consuming) computations. Our framework assumes the same model as in TrueTime (which is also suited to the typical code generation process for Simulink models). The execution of a task is split in units that are atomic from the standpoint of execution time granularity, but can be preempted, called *segments*, informally corresponding to the execution of a function called by the task main code. Each segment is identified by an execution time (possibly randomly generated according to a given distribution) and all segments in a task are executed according to a pre-defined sequence.

When the RT simulator is integrated with Simulink, segments represent the execution of Simulink subsystems and the execution order of the segments in a task must match the partial order of execution imposed on the subsystems. The time duration of each segment corresponds to the execution time of the code implementing the subsystem. The execution time of segments can be estimated in several ways. One possibility is to generate the code for each subsystem using the Embedded Coder tool. Once the code implementation is available, it can be executed on a (virtual or real) platform prototype measuring the execution times. In alternative, it can be compiled for the desired target and have the worst case timing analysis estimated by a dedicated tool like AiT from AbsInt.

At simulation time, the Simulink engine computes the model update in an outer loop, in which *major steps* are evaluated. A major step is a point in time in which the inputs and outputs of the model blocks are computed and updated. Inside each major step, there is an inner loop, in which *minor steps* are evaluated, allowing for the updates of the state of the continuous parts of the model.

Our real-time scheduling simulator is implemented as a set of custom blocks that execute at all the major steps and interact with the Simulink main engine (and capture all the relevant events from the simulated environment). The major steps of the Simulink simulation include all the periodic activation times of tasks, as well as the aperiodic events that lead to the activation of other tasks. The custom blocks define major steps in the simulation at all the points in time in which a scheduling event occurs. Every time a major step occurs, the block implementing the real-time scheduling simulator is invoked and processes (if there is any) the task arrival event and any other event that is active at the same time. The task activation instant corresponds to the activation of the first segment. Next, the real-time scheduler determines the tasks to be set in execution and the execution time for their current segments, determining the point in time when the current segments are expected to complete. These times are set as future major step times in the Simulink simulation.

Our framework adds the capability of simulating real-time task execution of Simulink models on single- and multi-core platforms through two custom blocks: `Kernel` and `Task`, implemented as C++ S-Functions (custom block implementations, blocks are shown in Figure 9). The interactions between the Simulink simulation engine and our custom blocks occur through a predefined set of API functions that allow setting inputs and outputs and forcing a simulation event (zero crossing point).

The block `Kernel` models an event-based real-time kernel and the scheduler inside it on a single- or multi-core computer node according to a given scheduling policy. Each task is modeled with one instance of the block `Task` and consists of the serialized execution of the segments/-subsystems.

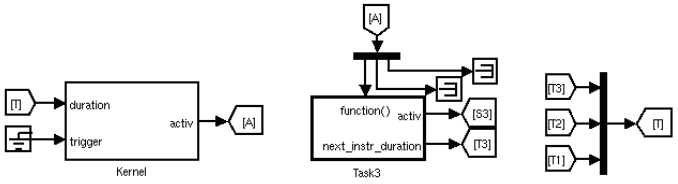

Figure 9. `Kernel` and `Task` blocks.

Each block `Task` is a triggered subsystem, executed on the occurrence of a function call event received on its port `function()`. Its output interface consists of two ports: `activ` and `next_instr_duration`. The first one is an array of function call events with size equal to twice the number of subsystems managed by the task. This port is used to issue activation and termination events to the Simulink subsystems executed in the the task segments. The second port outputs a scalar signal representing the duration of next segment executed by the task. Each time `Task` is triggered, it issues the termination signal for the previously executed segment (if any), outputs the activation signal for the current segment, and transmits the execution time of the new segment to the block `Kernel`. The duration of segments executed by `Task` is set by a variable in the Matlab workspace. The computation time of a segment can be fixed or random (uniform, exponential and Dirac delta distributions).

The block `Kernel` has two input ports: `duration` and `trigger`. On the `duration` port receives an array of values, one for each `Task` block, with the indication of the duration of the next segment to be executed. On the second port, it receives the array of activations signals of aperiodic tasks (from external sources). The block has one output port, named `activ`, which is used to signal to each task the execution of the current segment. The block `Kernel` is responsible for keeping the scheduling simulation aligned with the system simulation. At each activation, it checks for any aperiodic requests. If there is any, it activates the corresponding aperiodic tasks. Next, it advances the RT

scheduler simulator. Two types of events are relevant for the simulation: the *segment completion* and *task completion*. In case of a segment completion, `Kernel` reads the input signal on the port `duration` and dynamically creates a new instruction for the corresponding task. In case of a *task completion*, `Kernel` resets the internal state of the corresponding task.

A number of parameters configure the (simulated) kernel and are set through the `Kernel` mask dialog. The scheduling policy (Deadline Monotonic - DM, Fixed-Priority - FP, and Earliest Deadline First - EDF), with its options (deadline miss recovery), and the number of cores on which the task execution is simulated. The current implementation of the scheduler simulator is obtained through an abstract interface from the open source RTSim project (rtsim.sssup.it). RTSim supports multi-core architectures with global scheduling policies.

The actual start and completion times of the task segments must correspond to the times in which the corresponding subsystems reads or sample their inputs and produce their outputs. To guarantee this execution semantics that accounts for CPU availability, the activation of the (formerly periodic) subsystem blocks by Simulink must be inhibited. The subsystem blocks must be changed from periodic to function activated (Figure 10) and a latch barrier must be added on all its outputs. The signals activating the subsystem (and its input sampling) and the output latch are generated by the task blocks upon the beginning of the execution and the completion of the corresponding task segment.

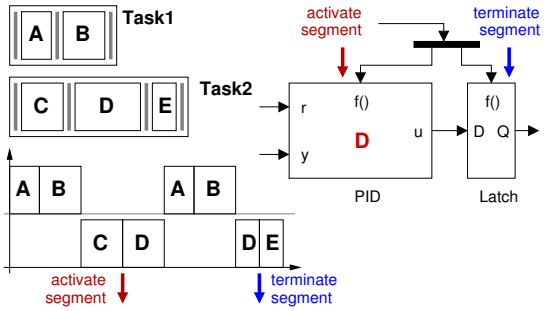

Figure 10. The execution of subsystems modeled through segments.

Figure 10 describes the activation mechanism of a block (D in this example). When a task segment starts executing, the block is activated (activation signal). The output signal u is latched and enabled to the output only when the segment terminates the execution.

V. Generation of Back-Annotated Models from the Mapping Model

A set of Acceleo Model-to-text generation templates processes the SysML Papyrus model and generates automatically the Simulink blocks for the task and scheduler implementation. The current set of scripts (and the Simulink custom blocks) handle the case of single-core and multicore execution under global scheduling policies. The Acceleo scripts are invoked from a common main template that performs the following sequence of operations: 1) The Simulink custom library of tasks and scheduler blocks is opened. 2) The functional model is saved and a new model is created for its backannotated version. 3) A Matlab script is generated, that creates the initialization variables for the kernel and the task attributes. 4) Another Matlab script is generated for the generation of the kernel and the task blocks. 5) Finally, another set of .m files is created to modify the input model by changing the subsystem blocks to triggered, adding latches on the output links and rerouting the connections (removing the old links and adding new ones that go through the latches).

As an example, Figure 11 shows the most relevant part of the template file that generates the kernel block.

```
[template public generate_kernel(mdl : Model) post(trim
  ())]
...
[file ('kernel_gen_commands.m', false, 'UTF-8')]
[** - Adding the Kernel block  */]
[generateKernelBlock(mdl_name, cpu, rtos)/]
[**  - Adding the infrastructure for the activation of
  tasks and signals with task duration */]
[generateTasksManagementInfrastructMulti(mdl_name,
  t2c_set)/]
[**  - Adding the Duration of next task instruction */]
[generateBlocksOfNextDuration(mdl_name)/]
[/file]
```

Figure 11. Acceleo template instructions for the generation of the kernel and task blocks.

Figure 12 shows the template sections that generate the Matlab instructions for the creation of the kernel block.

```
[template public generateKernelBlock(mdl_name : String,
  cpu : Class, rtos : Class) post(trim())]
[** Add and configure the Kernel block */]
...
add_block('yaks/Kernel', '[mdl_name/]_bn/Kernel1', '
  Position', kern1_pp);
[** Configure 'taskset_descr_name'  */]
set_param('[mdl_name/]_bn/Kernel1', 'taskset_descr_name
  ', 'task_set_descr');
[** Compute the other mask parameters by using the Class
   instances cpu and rtos */]
[** Set the scheduling policy ('scheduling_policy') */]
[let sched : Class = rtos.getSchedulerFromRtos()]
[setKernelMaskParamSchedPolicy(mdl_name, sched)/]
[** Set the Deadline miss rule ('dead_miss_rule') */]
set_param('[mdl_name/]_bn/Kernel1', 'dead_miss_rule',
'[sched.getValueOfStereotypePropertyEnumLit('
  BswResources::BswRTOS::BswScheduler',
'dlMissPolicy')/]');
[/let]
[** Set the Time resolution ('time_res') */]
set_param('[mdl_name/]_bn/Kernel1', 'time_res',
'[rtos.getValueOfStereotypePropertyEnumLit('BswResources
  ::BswRTOS::BswRTOS', 'timeRes')/]');
[** Set the Number of cores ('core_num') */]
set_param('[mdl_name/]_bn/Kernel1', 'core_num',
'[cpu.getValueOfStereotypeProperty('MARTE::
  MARTE_DesignModel::HRM::HwLogical::HwComputing::
  HwProcessor',
'nbCores')/]');
[** Set the Underlying engine ('under_engine') */]
set_param('[mdl_name/]_bn/Kernel1', 'under_engine', '
  RTSIM');
...
[/template]
```

Figure 12. Acceleo template instructions for the generation of the kernel block.

VI. Application Example

The three servo Simulink example with the PID controllers of Figure 2 and Figure 5 (in SysML) is mapped

onto a single-core platform. The result of the mapping is the structure of tasks in Figure 8. The example assumes that each segment of control code, implementing the PID functionality of one subsystem, takes a *fixed* amount of time equal to $2ms$ to execute.

The mapping model includes all the information needed to automatically generate and add as back-annotations the kernel and task blocks to the original Simulink model. The Acceleo M2T transformation processes the mapping model and generates a collection of Matlab scripts that contain the back-annotation commands. The execution of the Matlab scripts produces the Simulink model of Figure 13.

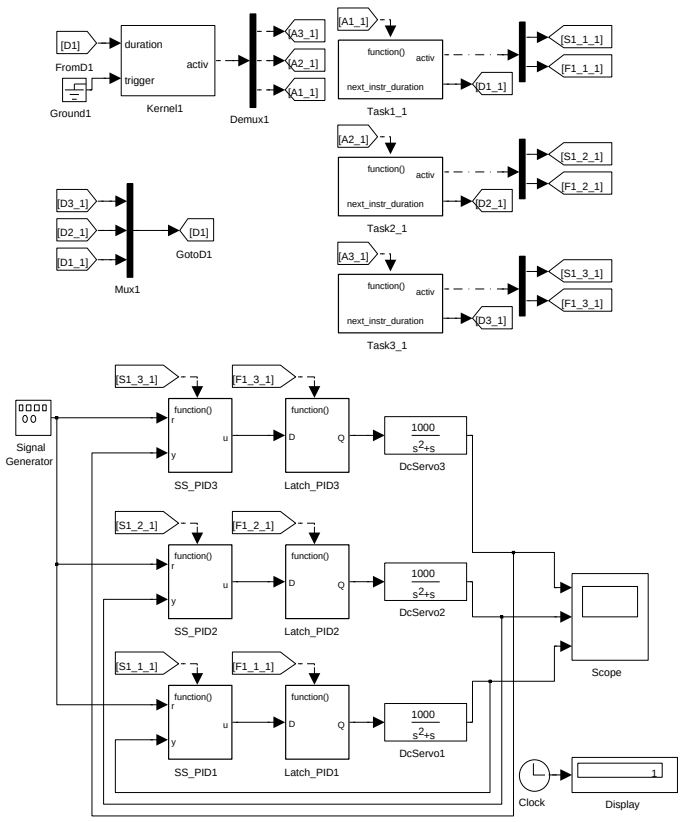

Figure 13. Simulink model with back-annotations of kernel and task models.

One instance of kernel block (`Kernel1`) and three instances of task blocks (the names are the same of the corresponding SysML modeling artifacts) are added to the functional model. Since the example does not consider aperiodic tasks, a ground block is connected to the `trigger` port of the kernel block (top-side).

`Kernel1` outputs task-activation signals in the order specified by the `taskSetIdx` attributes of the «`ThreadToCPUMap`» allocations in the mapping model. Similar considerations apply to the way tasks communicate the duration of the next segment to `Kernel1` through the `Goto-From` connections `D3_1`, `D2_1` and `D1_1`. Each PID subsystem is transformed to a triggered subsystem and a latch barrier is added on all its outputs.

Task blocks manage the activation and termination signals of the PID subsystems executing in the segments. Two function-call signals are issued by each task block. These signals are sent to the trigger port of the subsystem and latch blocks managed by the task through `Goto-From` connections. Connection labels are prefixed with `S1_` and `F1_` to indicate, respectively, the activation and termination signals of the first (and, in this case, the only) segment of each task. In the general case, the indices of activation and termination signals of a specific segment depends on the `mapOrder` attribute of the «`FunctionToThreadMap`» instance that describes the subsystem-to-task allocation relationship.

Figure 14 shows a snapshot of the generated Matlab code that adds the block `Kernel1` and configures its parameters.

```
% - Add and configure the Kernel block
kern1 = 'threeservos_bn/Kernel1';
add_block('t_res/Kernel', kern1);
set_param(kern1, 'taskset_descr_name','task_set_descr');
set_param(kern1, 'scheduling_policy', 'EDF');
set_param(kern1, 'dead_miss_rule', 'Continue');
set_param(kern1, 'time_res', 'Milli_Seconds');
set_param(kern1, 'core_num', '1');
set_param(kern1, 'under_engine', 'RTSIM');
```

Figure 14. MATLAB commands for the generation of block `Kernel1`.

A number of parameters are readily available from the platform model: (`scheduling_policy`, `dead_miss_rule`, `time_res` and `core_num`), and their values are set through the Matlab function `set_param()`.

The timing properties and the type of tasks in the task-set are described by the Matlab variable `task_set_descr` in Figure 15 (cell array).

```
% Description of timing properties of task set
task_set_descr = { ...
    % type            %iat        %rdl        %ph
    'PeriodicTask',  4*0.001     4*0.001     0; ...
    'PeriodicTask',  5*0.001     5*0.001     0; ...
    'PeriodicTask',  6*0.001     6*0.001     0; ...
};

% Sequences of pseudo instructions (task codes)
task3_1_descr = { 'fixed(0.002)' };
task2_1_descr = { 'fixed(0.002)' };
task1_1_descr = { 'fixed(0.002)' };
```

Figure 15. Definition of type and timing properties of tasks in the task-set, and of execution times of tasks activities.

Tasks types and periods (or interarrival times) are available from the mapping model of Figure 8. Relative deadlines coincide with periods. For each task, the time-scale constant is generated depending on the value of the `unit` of the task period (`NFP_Duration`), which is of type `TimeUnitKind` and for which MARTE provides a `convFactor` attribute (with respect to seconds). In this case, all the tasks periods are expressed in *milliseconds*, therefore a time-scale constant equal to 0.001 is generated. The duration of segments executed by each task is described by a Matlab cell array of strings (Figure 15). Each string that describes the computation time of a segment is available from the (SysML) functional model of Figure 5.

The back-annotated Simulink model enables the verification (by simulation) of the impact that scheduling and

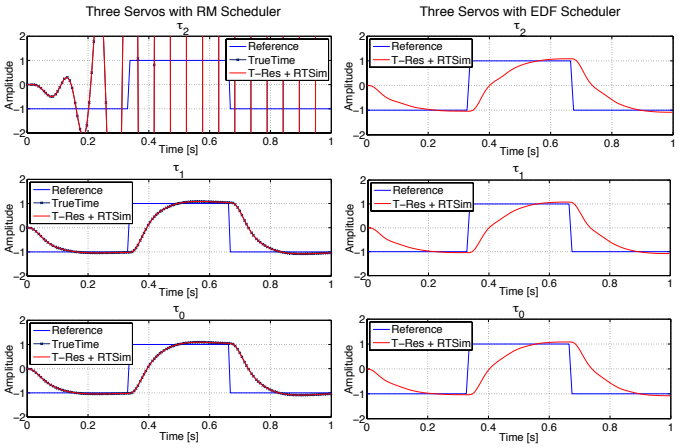

Figure 16. Verification by simulation on the back-annotated model.

execution times delays have on the performance of the controls. Figure 16 shows the output of the DC-servos with respect to the reference signal. when a Rate Monotonic (RM, on the left) or EDF scheduling policy (on the right) is used. In both cases, task `Task1_1` (on top) has the lowest priority. In this example, the CPU utilization factor is $U \simeq 1.23$. The overload condition degrades the performance of controls with respect to the results obtained from the Simulink model without back-annotations. In the case of RM, the task with the lowest priority cannot guarantee a stable control, because of too many deadline misses. In the case of EDF, the delay due to scheduling tends to be spread among the three tasks, and after an initial transient all tasks miss their deadlines. However, the motion of the DC-servos is still controlled with a reasonable error, and the overall control performance is satisfactory.

VII. Conclusions and Future Work

We present a framework for the definition of execution platforms and task implementations of Simulink models. The platform and the task implementation are defined in SysML/MARTE, together with the selection of the scheduling policies. This allows to keep the functional model separated from the implementation. An automatic generation flow allows to obtain a new Simulink model that contains custom blocks that model the computation and scheduling delays and evaluates their impact on the performance of the controls. Future work includes the extension to distributed architcetures, with the modeling of networks, messages and communication delays.

References

[1] The MathWorks, Inc. Simulink. [Online]. Available: http://www.mathworks.com/products/simulink/

[2] Esterel Technologies. SCADE Suite. [Online]. Available: http://www.esterel-technologies.com/products/scade-suite/

[3] EAST Architecture Description Language (ADL). [Online]. Available: http://www.east-adl.info/

[4] AUTomotive Open System ARchitecture (AUTOSAR). Specifications 4.0. [Online]. Available: http://www.autosar.org/

[5] A. Davare, D. Densmore, L. Guo, R. Passerone, A. L. Sangiovanni-Vincentelli, A. Simalatsar, and Q. Zhu, "metroii: A design environment for cyber-physical systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, Mar. 2013.

[6] Object Management Group. Model Driven Architecture (MDA). [Online]. Available: http://www.omg.org/mda/specs.htm

[7] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. A. Vissers, "A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach," in *Proceedings of Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, 2002, pp. 18–37.

[8] TIMing MOdel – TOols, algorithms, languages, methodology, and USE cases (TIMMO-2-USE). [Online]. Available: https://itea3.org/project/timmo-2-use.html

[9] G. Raghav, S. Gopalswamy, K. Radhakrishnan, J. Delange, and J. Hugues, "Model Based Code Generation for Distributed Embedded Systems," in *Proceedings of the European Congress on Embedded Real Time Software and Systems, ERTSS*, 2010.

[10] 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.

[11] Object Management Group. (2011) A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems. OMG Document Number formal/2011-06-02. [Online]. Available: http://www.omg.org/spec/MARTE

[12] E. Wozniak, C. Mraidha, S. Gerard, and F. Terrier, "A Guidance Framework for the Generation of Implementation Models in the Automotive Domain," in *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA*, 2011, pp. 468–476.

[13] Automatic Software Generation for Real-Time Embedded Systems (Gene-Auto). [Online]. Available: http://gforge.enseeiht.fr/projects/geneauto

[14] Project P. [Online]. Available: http://www.open-do.org/projects/p/

[15] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck, "The Rubus Component Model for Resource Constrained Real-Time Systems," in *Proceedings of the IEEE International Symposium on Industrial Embedded Systems, SIES*, 2008.

[16] Architecture Analysis & Design Language (AADL). [Online]. Available: http://standards.sae.org/as5506b/

[17] A. Triki, J. Combaz, S. Bensalem, and J. Sifakis, "Model-based implementation of parallel real-time systems," Springer LNCS vol. 7793.

[18] 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, DATE*, 2006, pp. 93–93.

[19] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, and M. Qamhieh, "YARTISS: A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms," in *Proceedings of Int. Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems, WATERS*, 2012, pp. 21–26.

[20] R. Urunuela, A.-M. Déplanche, and Y. Trinquet, "STORM a Simulation TOol for Real-time Multiprocessor scheduling evaluation," in *Proceedings of IEEE Int. Conference on Emerging Technologies and Factory Automation, ETFA*, 2010, pp. 1–8.

[21] D. Decotigny and I. Puaut, "ARTISST: An Extensible and Modular Simulation Tool for Real-Time Systems," in *Proceedings of IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing, ISORC*, 2002, pp. 365–372.

[22] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: a flexible real time scheduling framework," in *Proceedings of the ACM International Conference on Ada (SIGAda)*, 2004, pp. 1–8.

[23] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "STRESS: a Simulator for Hard Real-time Systems," *Softw., Pract. Exper.*, vol. 24, no. 6, pp. 543–564, 1994.

[24] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K. E. Årzén, "How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime," *IEEE Control Syst. Mag.*, vol. 23, no. 3, pp. 16–30, June 2003.

[25] Object Management Group. (2011) The Unified Modeling Language (UML). [Online]. Available: http://www.omg.org/spec/UML/2.4.1/

[26] ——. (2012) The Systems Modeling Language (SysML). [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[27] Obeo. Acceleo. [Online]. Available: http://www.eclipse.org/acceleo/