Received July 25, 2016, accepted August 25, 2016, date of publication August 29, 2016, date of current version October 31, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2604018

UML-Based Development of Embedded Real-Time Software on Multi-Core in Practice: Lessons Learned and Future Perspectives

FEDERICO CICCOZZI[1], (Member, IEEE), TIBERIU SECELEANU[2], (Senior Member, IEEE), DIARMUID CORCORAN[3], AND DETLEF SCHOLLE[4]
[1]Mälardalen Real-Time Research Centre, School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
[2]ABB Corporate Research, Västerås, Sweden
[3]Ericsson AB, Kista, Sweden
[4]Alten Sweden AB, Kista, Sweden

Corresponding author: F. Ciccozzi (federico.ciccozzi@mdh.se)

This work was supported by the Knowledge Foundation through the SMARTCore Project led by Mälardalen University and run in cooperation with Ericsson AB, ABB Corporate Research, and Alten Sweden AB.

ABSTRACT Model-driven engineering has got a foothold in industry as an effective way to tame the complexity of modern software, which is meant to run on embedded systems with real-time constraints by promoting abstraction, in terms of prescriptive models, and automation, in terms of model manipulations. In the plethora of modeling languages, the unified modeling language (UML) has emerged and established itself as a de facto standard in industry, the most widely used architectural description language and an ISO/IEC standard. In the SMARTCore project, we have provided solutions for the UML-based development of software to run on multicore embedded real-time systems with the specific focus of automating the generation of executable code and the optimization of task allocation based on a unique combination of model-based and execution-based mechanisms. In this paper, we describe the lessons learned in the research work carried out within SMARTCore and provide a set of perspectives that we consider to be highly relevant for the forthcoming future of this research area to enable a wider adoption of UML-based development in industry in general, and in the multicore embedded real-time domain in particular.

INDEX TERMS Model-driven engineering, UML, ALF, task allocation optimization, embedded, real-time, code generation.

I. INTRODUCTION

Software is idiosyncratically complex and the fact that it is often meant to run on embedded systems with real-time constraints does not simplify its development. Already in the 90's it became clear that the very fast growth of embedded software would lead to the need of defining more powerful and flexible development approaches [1] based on *abstraction* [2]. While abstraction can effectively help in taming software complexity, it also introduces additional artefacts and development phases (e.g., design, transition from design to implementation); this can make the software engineering process more intricate [3]. A common way to mitigate the complexity of an engineering process is boosting *automation* in the various engineering phases.

Abstraction and automation are the two key aspects advocated by Model-Driven Engineering (MDE), which represents an effective way to tackle the difficulty of code-centric engineering approaches to effectively tame complexity, express domain-specific concepts and support human communication [4]. It does that by promoting (i) prescriptive models, for defining abstractions of the software under development, and (ii) model transformations, for automating development steps, to first-class development artefacts. Rules and constraints for building models have to be properly described through a corresponding *modelling language* which includes a *metamodel* describing the set of available concepts and well-formedness rules that a correct model must conform to [5]. Following the MDE paradigm, software is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements are performed through transformations between models. Model transformations are manipulations of models that provide the links between domain abstractions. A transformation is intended as a process that converts source models into target artefacts (i.e., models or text) related to the same system by means of a transformation specification [6]. Several modelling languages have been proposed throughout the years. Among them, the Unified Modeling Language (UML) represents a de facto standard in industry [7], the most widely used architectural description language [8] and an ISO/IEC (19505-1:2012) standard; in the research work reported in this paper we focused specifically on UML and related sub-languages.

Generally MDE aims at reducing final product verification and validation effort and cost, as well as shortening time-to-market, by providing *correctness-by-construction* [9], in contrast to the more costly *correctness-by-correction* typical of code-centric approaches. To do so, one of the key aspects of MDE is the ability to produce executable artefacts from design models. The automated production of executable programs is considered pivotal in industry [7] for economic reasons but also for mitigating the accidental complexity introduced by modelling activities [10]. In this paper we discuss on the automatic code generation from UML models for embedded real-time systems. Addressing functional modelling is clearly crucial in software development, but in the complex world of embedded real-time systems it is not enough. In fact, while on the one hand these systems demand ever-increasing performance to deal with their real-time constraints, on the other hand they always face a certain resource limitation typical of their embedded nature. Commonly, higher performance demand is addressed by increasing the processing power, which in this kind of systems is achieved through the adoption of multicore[1] and manycore solutions; in our research we focus on the former. While a solution with multiple processing units is beneficial for increasing processing power, it also introduces a new challenge that affects system development, namely *how to deploy (allocate) software units to cores to best utilise the hardware platform*. In this paper we focus on this aspect too.

In the *SMARTCore* project (see Acknowledgements) we provided solutions for UML-based development of embedded real-time software with particular focus on: (i) modelling, (ii) task allocation optimization, and (iii) automatic code generation. Regarding optimization it is important to stress the fact that our aim is *not* to provide a design-space exploration technique. In this paper we expose the lessons learned from the research work carried out in SMARTCore to achieve the goals described in [11], as well as the future perspectives we envision for researchers and practitioners interested advancing the states of the art and practice in the research area.

The remainder of the paper is organised as follows. Section II introduces the context of the SMARTCore project and the solutions developed within it. In Section III we introduce a running example which is used for describing the various steps of the approach developed in SMARTCore. The approach itself is described in Section IV, while a discussion on lessons learned and future perspectives is given Section V. A snapshot of the literature describing work related to SMARTCore is provided in Section VI and the paper is concluded with Section VII.

II. CONTEXT

The goal of the SMARTCore approach is to provide a solution for UML-based model-driven engineering of multicore embedded soft[2] real-time systems. Since we are mostly interested in timeliness, the extra-functional properties (EFPs) we focus on are timing-related attributes such as: *end-to-end response time* (elapsed time between the point when the task starting the chain becomes ready for execution, and the point when the last task in the chain finishes the corresponding execution instance), *deadline misses* and *core load* (percentage of core's busy portion). These properties are dependent on the dynamic intertwining of tasks and we are interested in the average-case behaviour. For these reasons we leverage dynamic (based on simulations and system runs) model-based and execution-based optimization mechanisms, since the aforementioned EFPs cannot be derived analytically from task parameters.

A. TASK MODEL

Regarding tasks we support *periodic* tasks, whose execution is triggered after a defined fixed time-interval (i.e., period), and *event-triggered* tasks whose execution is triggered by other tasks finishing their execution. A *task chain* represents the flow of execution, and it is defined by a periodic task starting the chain and a set of event-triggered tasks triggered in ordered sequence. Each task is provided with a set of parameters: *priority*, used for scheduling purposes, *allocation*, pointing to the core the task is allocated to, *best-case execution time* (BCET) and *worst-case execution time* (WCET) which we employ for simulation purposes. Moreover, periodic tasks have one additional parameter, namely *period*, defining at which time interval the task execution is triggered. Currently, only a uniform distribution of task execution times is implemented and we assume synchronous communication.

End-to-end response times are EFPs that are defined at chain level, and represent the duration between the point in time when the periodic task at the start of the chain begins its execution, until the point in time when the last task in the chain finishes its execution. During their execution tasks do not move between cores, as they are statically allocated. Each core has a *preemptive fixed priority-based scheduler* in charge of running the tasks assigned to it.

[1]A *multicore processor* is a single chip that contains two or more processing units (i.e., cores) which are tightly coupled in order to preserve, e.g., energy-efficiency.

[2]*Soft* in the sense that accurate timing behaviour is crucial for the correct functioning of the system, (a logically correct result that is produced at the wrong time point is equivalent to a logically incorrect result), but occasional deadline misses are tolerated (as opposed to hard real-time systems where the absence of deadline misses must be guaranteed beforehand).

In the following we formalise the aforementioned notions of task, periodic task, event-triggered task and chain.

Definition 1: A task T is a non-instantiable tuple $T = \langle B, W, pr, a \rangle$, where B represents T's BCET, W represents the WCET, pr represents T's scheduling priority and a represents the affinity parameter identifying the core to which T is allocated.

Definition 2: A periodic task PT is an instantiable specialisation of T defined as the tuple $PT = \langle T, pe \rangle$, where T represents the tuple $\langle B, W, pr, a \rangle$ and pe represents PT's period.

Definition 3: An event triggered task ET is an instantiable specialisation of T defined as the tuple $ET = T$, where T represents the tuple $\langle B, W, pr, a \rangle$.

Definition 4: A chain C is a non-empty ordered set of tasks $\{PT, T_1, T_2, \ldots, T_n\}$ with $|C| \geq 1$ and where the first element is always represented by a periodic task PT.

B. MODELLING LANGUAGE AND ENVIRONMENT

The modelling language leveraged in the SMARTCore approach is represented by UML [12] for functional descriptions and the Action Language for Foundational UML (ALF) [13] for defining fine-grained behaviours. Functional models are decorated with extra-functional information through the stereotypes provided by the Modeling and Analysis of Real Time and Embedded systems (MARTE) profile [14]. For describing the hardware model in terms of available cores and deployment information in terms of allocation of tasks to cores we exploit specific concepts provided by MARTE. The approach is implemented and runs on top of Papyrus,[3] an open source integrated environment for editing EMF [15] models and particularly supporting UML and related profiles, on the Eclipse platform.

C. MODEL TRANSFORMATIONS

Following the MDE paradigm, a system is developed by designing models and refining them starting from higher and moving to lower levels of abstraction until code is generated; refinements are performed through transformations between models. A *model transformation* translates a source model to a target model while preserving their well-formedness [6]. More specifically, in SMARTCore we exploit the following kinds of model transformation:

- **Model-to-model (M2M):** which translates between source and target models that can be instances of the same or different languages;
- **Model-to-text (M2T):** which is a particular case of M2M where the target artefact is represented by text;
- **Text-to-model (T2M):** that operates in the opposite direction as the M2T, generating a model from a textual representation.

Moreover, any of these types of model transformation can be defined as *in-place*, meaning that source (or one of the sources) and target are represented by the same model; in this case, the transformation provides as output an updated version of (one of) the model(s) in input. Except for the in-place transformations which are by nature *endogenous*, the other transformations entailed in SMARTCore are *exogenous* meaning that they operate between artefacts expressed using different languages [6]. M2M transformations are implemented with the Operational QVT (QVTo)[4] language, M2T and T2M transformations with Xtend.[5]

III. A RUNNING EXAMPLE

The system we leverage for showing the various steps of our approach by a carrier robot self-orienting in a closed environment. This terrestrial robot is meant to travel between checkpoints in a delimited and known environment and operate item retrieval and delivery. The application is expected to provide the robot with the ability to orient itself around obstacles of simple shapes; obstacles are created in different places, but within the environment's delimitations. In a similar way, a set of pick-up spots and one drop-off spot are created too.

From a behavioural point of view, the robot is initialised, it is placed in the drop-off spot and starts its mission. The robot has then to fetch items from the pick-up spots and release them within the drop-off area. It does that by moving towards pick-up spots and constantly updating its direction until it intersects with the target item. Once the robot has picked up an item, it moves back to the drop-off zone and releases the item. The robot has 3 possible directions: forward, right, and left. It sorts directions prioritising the one leading to the closest pick-up spot and it moves in a direction as long as it does not intersect with any obstacle. If no direction can be taken, the robot turns back by turning twice by 90 degrees to left or right. The application stops its execution when the robot has picked up and released all items.

The system is conceived as object-oriented as follows. *Robot* is the main class and exploits two classes, *Vector* and *Hitbox*, for moving in the environment and identifying sensitive spots (its body, pick-up and drop-off spots, obstacles), respectively. More specifically, the classes are defined as follows:

- *Vector*: used to define vectorial movements. It contains two properties, *X* and *Y*, defining 2-dimension coordinates, one constructor (*Vector(..)*), and three methods, *vecRotateLeft()*, *vecRotateRight()* and *eq(..)*, which are exploited by the robot to perform movements;
- *Hitbox*: used to describe sensitive spots such as pick-up and drop-off areas as well as position and size of obstacles, and the robot's body size. Positions are identified through *pos* of type *Vector*, while sizes through the properties *height* and *width*. The *Hitbox(..)* method represents the class constructor and the *intersectsWith(..)* method allows the robot to check if the movement trajectory intersects an obstacle.

[3] https://eclipse.org/papyrus/

[4] http://www.eclipse.org/mmt/?project=qvto

[5] http://www.eclipse.org/xtend/

- *Robot*: represents the main class and contains a number of properties and the methods used by the robot to carry out its mission. More specifically, *Robot(..)* is the constructor, *fetch(..)* and *fetchList(..)* are used to retrieve single items and the initial items list, respectively, while the remaining methods allow the robot to move and orient itself in the environment.

In the next sections we use the robot system for showing the proposed model-driven approach.

IV. THE SMARTCore APPROACH

The goal of the SMARTCore approach is to provide a solution for UML-based model-driven engineering of multicore embedded real-time software. More specifically, the approach provides the following features:

- Modelling the system in terms of software functionalities, tasks, available cores and allocation of software functionalities to tasks and of tasks to cores;
- Combined model-based and execution-based task allocation optimization;
- Automatic generation of full-fledged code from system models.

A. MODELLING THE SYSTEM

The SMARTCore approach is depicted in Fig. 1. The first step is to model software in terms of functional elements (classes and/or components), tasks, hardware in terms of cores, and allocations of functional elements to tasks and of tasks to cores. The design model (Fig. 1.A) is defined in terms of UML, ALF and MARTE, and it is composed of four main parts: (1) *software components*, where we define the hierarchical structure using UML and the functional description in terms of fine-grained behaviours in ALF, (2) *software tasks*, in which we focus on the definition of schedulable tasks using UML and MARTE, (3) *hardware platform*, where we provide a simple description of the available cores and schedulers, and (4) *allocations* of software components to tasks and tasks to cores. Note that the allocation of software components to tasks is meant to be decided at design time too but it is not manipulated (i.e., optimised) by the approach; for this reason it is not shown in detail in the paper. On the other hand, the allocation of tasks to core is the one that we aim at optimising and on which we lay our focus throughout the paper. In Fig. 2, we show the UML structural model of the self-orientating carrier robot defined in terms of classes and associations in a UML class diagram.

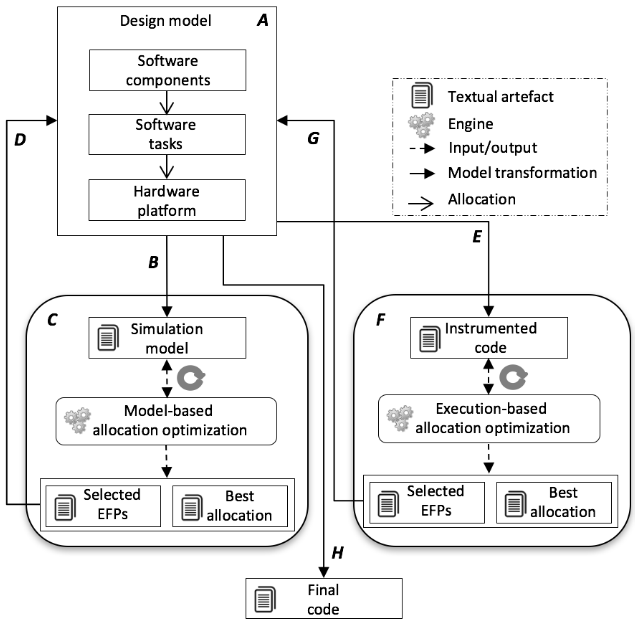
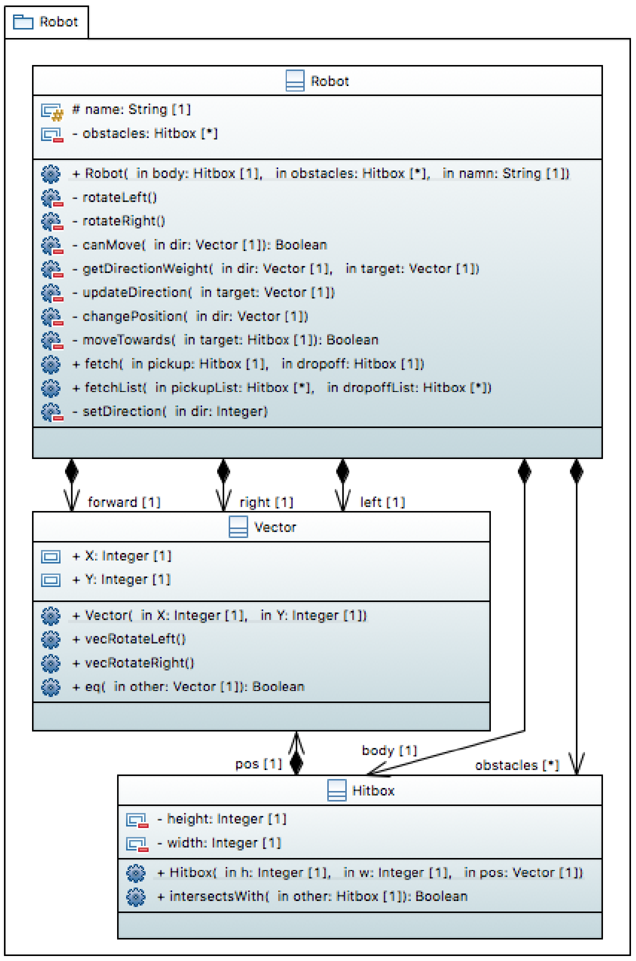


FIGURE 1. The SMARTCore approach.



Robot
Robot
# name: String [1]
- obstacles: Hitbox [*]
+ Robot(in body: Hitbox [1], in obstacles: Hitbox [*], in namn: String [1])
- rotateLeft()
- rotateRight()
- canMove(in dir: Vector [1]): Boolean
- getDirectionWeight(in dir: Vector [1], in target: Vector [1])
- updateDirection(in target: Vector [1])
- changePosition(in dir: Vector [1])
- moveTowards(in target: Hitbox [1]): Boolean
+ fetch(in pickup: Hitbox [1], in dropoff: Hitbox [1])
+ fetchList(in pickupList: Hitbox [*], in dropoffList: Hitbox [*])
- setDirection(in dir: Integer)
forward [1]
right [1]
left [1]
Vector
+ X: Integer [1]
+ Y: Integer [1]
+ Vector(in X: Integer [1], in Y: Integer [1])
+ vecRotateLeft()
+ vecRotateRight()
+ eq(in other: Vector [1]): Boolean
pos [1]
body [1]
obstacles [*]
Hitbox
- height: Integer [1]
- width: Integer [1]
+ Hitbox(in h: Integer [1], in w: Integer [1], in pos: Vector [1])
+ intersectsWith(in other: Hitbox [1]): Boolean

FIGURE 2. Functional model.

To model the dynamic behaviour of the system, the body of each UML operation (e.g., `updateDirection(..)` owned by the main class `Robot`) is described in terms of ALF. An extract of the ALF code describing `updateDirection(..)` is provided in Listing 1.

In Fig. 3 we show an extract of the remaining elements of a SMARTCore design model, that is to say tasks, hardware and allocations of tasks to cores. Schedulable tasks (`t1`–`t14`) are

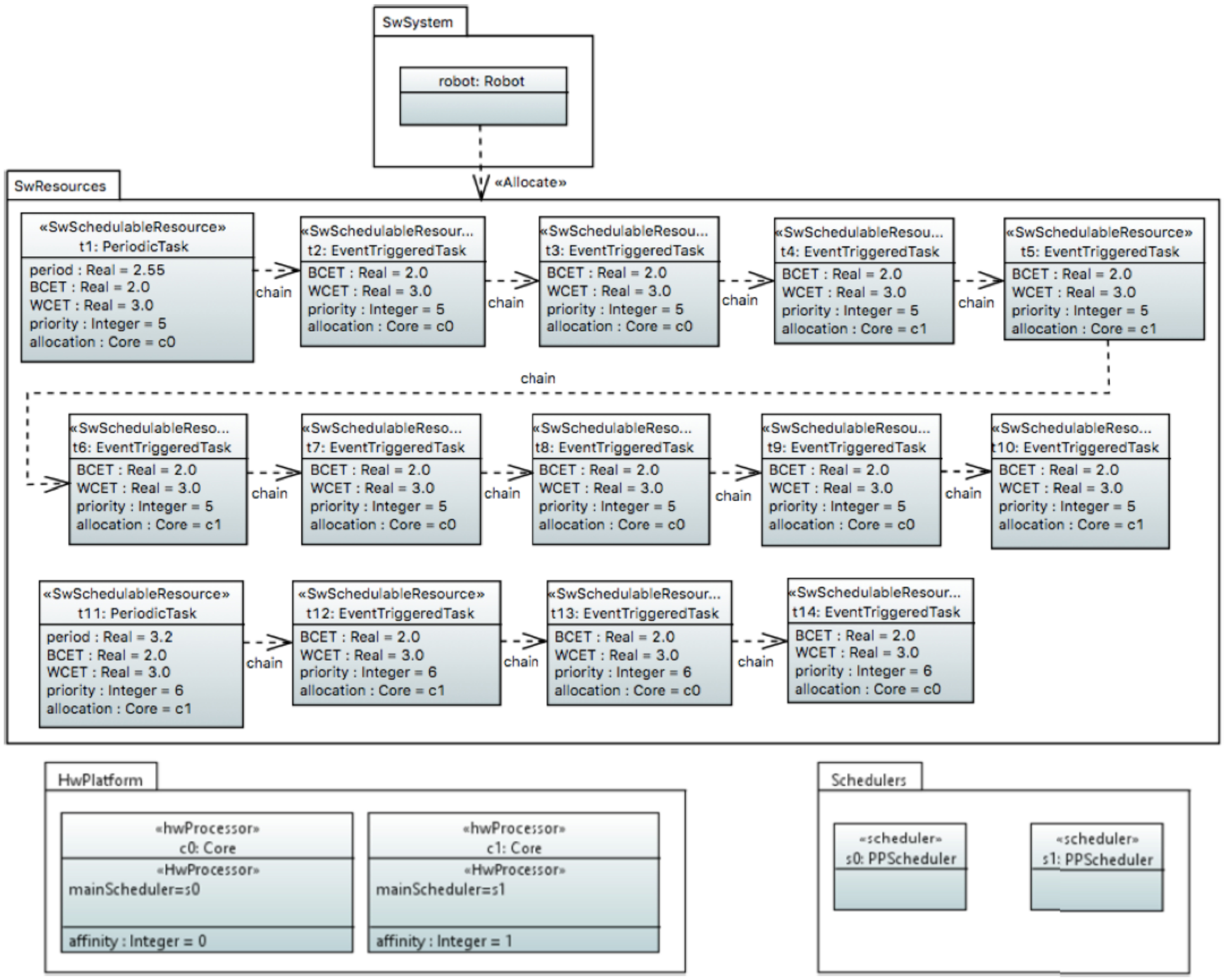


FIGURE 3. Tasks, hardware and allocation.

```
private updateDirection(in target : Vector) :
    Boolean{
  let movePrio : Vector[] = Vector[]{new
      Vector(this.forward.X, this.forward.Y),
      new Vector(this.left.X, this.left.Y), new
      Vector(this.right.X, this.right.Y)};
  [...]
  while(this.getDirectionWeight(movePrio[1],
      target) <
      this.getDirectionWeight(movePrio[2],
      target)){
    movePrio[1] = movePrio[2];
    [...]
  }
  [...]
}
```

Listing 1. Extract from ALF description of `updateDirection(..)`.

defined as *«SwSchedulableResource»* MARTE elements and allocated through a one-to-one connection to a core (stereotyped as *«hwProcessor»*) through the attribute *allocation*. As aforementioned, we support two kinds of tasks: (i) periodic (defined as *PeriodicTask* in the model, e.g. `t1`), and (ii) event-triggered (defined as *EventTriggeredTask*, e.g. `t2`). Cores are defined as *Core* (`c0, c1`) in the model and they have a main scheduler, defined in the stereotype's property *mainScheduler* that refers to a scheduler instance (stereotyped as *«scheduler»*). In the example two instances `s0` and `s1` of a fixed priority preemptive scheduler (defined as *PP_scheduler*) are shown. Since the task allocation optimization mechanism is based on task chains, we model this information through directed dependency links called *chain*. In Fig. 3 we can see two chains, one starting with periodic task `t` and the other one starting with the periodic task `t11`.

According to the task model[6] defined in Section II-A, software tasks and their allocations as well as task chains shown in Fig. 3 can be formalised as follows:

[6]Times are defined in *milliseconds*.

- **Event-triggered tasks:**
  `t2`= ⟨2, 3, 5, `c0`⟩
  `t3`= ⟨2, 3, 5, `c0`⟩
  `t4`= ⟨2, 3, 5, `c1`⟩
  `t5`= ⟨2, 3, 5, `c1`⟩
  `t6`= ⟨2, 3, 5, `c1`⟩
  `t7`= ⟨2, 3, 5, `c0`⟩
  `t8`= ⟨2, 3, 5, `c0`⟩
  `t9`= ⟨2, 3, 5, `c0`⟩
  `t10`= ⟨2, 3, 5, `c1`⟩
  `t12`= ⟨2, 3, 6, `c1`⟩
  `t13`= ⟨2, 3, 6, `c0`⟩
  `t14`= ⟨2, 3, 6, `c1`⟩
- **Periodic tasks:**
  `t1`= ⟨2, 3, 5, 2.55, `c0`⟩
  `t11`= ⟨2, 3, 6, 3.2, `c1`⟩
- **Chains:**
  `tc1`= {`t1,t2,t3,t4,t5,t6,t7,t8,t9,t10`}
  `tc2`= {`t11,t12,t13,t14`}

B. ALLOCATION OPTIMIZATION

From the design model, a simulation model in Java is generated through M2T transformations (Fig. 1.B). The optimization mechanism is composed of two modules: model-based and execution-based. optimization starts with the model-based module which runs the simulation model (Fig. 1.C). In each iteration the simulation model is enriched with details about a particular allocation configuration according to the previous iteration in order to be analysed and evaluated. The simulation produces data from which we derive average end-to-end response times and deadline misses for task chains, and information about core load. These performance metrics are exploited to compare different allocation configurations. The best allocation configuration is kept, and used to generate an enhanced simulation model to be run in the next iteration. In order for the allocation optimization to stop (or the first module to stop for the second module to take over), we can define the following stop criteria: predefined maximum number of iterations, time limit, number of consecutive iterations that have not resulted in an improvement, a certain value for a particular EFP, or a combination of the above.

Since model-based analysis is performed on an abstraction of the system, it can typically produce performance predictions fast. This is the reason why we use model-based optimization to quickly asses a large number of allocation configurations and find a good one. Once the model-based optimization is not able to improve the allocation configuration (or reaches the pre-fixed time limit or maximum runs), the best allocation configuration is back-propagated to the design model in terms of in-place T2M transformations (Fig. 1.D) which update the model with the new allocation configuration. More specifically, it modifies the *allocation* attribute of the re-allocated *«SwSchedulableResource»* elements. From the enhanced design model, the approach automatically generates full-fledged and instrumented C/C++ code by means of M2T transformations (Fig. 1.E).

At this point, the execution-based optimization is run (Fig. 1.F) on the generated code. In each iteration, the mechanism modifies the generated code in order to execute and evaluate a particular allocation configuration. As for the model-based one, once the execution-based optimization is not able to improve the allocation configuration (or reaches the pre-fixed time limit or maximum runs), the best allocation configuration is back-propagated to the design model in terms of in-place T2M transformations (Fig. 1.G). Compared to the model-based optimization, this one is slower but, since it leverages the good allocation configuration identified in the model-based step as starting point, it usually needs to run for fewer iterations. Additionally, EFPs obtained by run-time measurements are more accurate than the model-based predictions. To summarise, we combine the speed of model-based analysis with the accuracy of performance measurements to provide code with improved allocation of tasks to cores.

In both modules, in order to find a new allocation configuration, a "problematic" task is identified and re-allocated. In order to identify problematic tasks we exploit the delay matrix heuristic defined by Feljan and Carlson [16]. This heuristic takes into consideration the fact that the most problematic task on a core is either the one that considerably delays tasks that belong to the chain to be optimised, or the one that is considerably delayed by other tasks. For identifying a new allocation, the heuristic takes the current best allocation specification and moves the problematic task to a core that has lowest load.

Task allocation optimization is performed with respect to end-to-end response times. The goal of the optimization mechanism is to find the best allocation of tasks to cores that permits to keep the number of deadline misses in the system below a desired boundary, while minimising the average end-to-end response time for a selected task chain. In the example, we wanted to minimise end-to-end response time of `tc2`, the optimization was set to run for 100 iterations (stop criterium 1) and the limit of allowed deadline misses is set to 0 (stop criterium 2). Once the optimization completed its job, the best allocation, with no deadline misses and an average response time for chain `tc2` of 13.79, is found and back-propagated to the design model. More specifically, the best allocation configuration is the following (in bold the modified allocations):

`t1`= ⟨2, 3, 5, 2.55, **`c1`**⟩
`t2`= ⟨2, 3, 5, `c0`⟩
`t3`= ⟨2, 3, 5, `c0`⟩
`t4`= ⟨2, 3, 5, **`c0`**⟩
`t5`= ⟨2, 3, 5, `c1`⟩
`t6`= ⟨2, 3, 5, `c1`⟩
`t7`= ⟨2, 3, 5, **`c1`**⟩
`t8`= ⟨2, 3, 5, `c0`⟩
`t9`= ⟨2, 3, 5, `c0`⟩
`t10`= ⟨2, 3, 5, `c1`⟩
`t11`= ⟨2, 3, 6, 3.2, `c1`⟩
`t12`= ⟨2, 3, 6, `c1`⟩

`t13`= $\langle 2, 3, 6, \texttt{\textbf{c1}} \rangle$
`t14`= $\langle 2, 3, 6, \texttt{c1} \rangle$

Several experiments have been carried out in order to evaluate the two optimization modules in isolation as well as their combination (which resulted the most efficient). For the interested reader, more details about the optimization mechanisms and the experiments can be found in [17].

C. CODE GENERATION

From the design model we can automatically generate an executable multithreaded C/C++ application instrumented for execution-based optimization (Fig. 1.F) or a 'clean' one for deployment (Fig. 1.H) through M2T transformations. The transformations navigate the model in all its parts and generate a corresponding description in C/C++. The tasks are transformed through a 1-to-1 mapping to threads. In the case of instrumented code, depending on the task type, specific triggering code (according to the modelled period for periodic tasks, at triggering task completion for event-triggered tasks) is generated too.

```
1  bool Robot::updateDirection(shared_ptr<ALF2CPP::
2   Robot::Vector> target)
3  {
4   vector<shared_ptr<ALF2CPP::Robot::Vector>>
5    movePrio = vector<shared_ptr<ALF2CPP::Robot
6     ::Vector>>({make_shared<ALF2CPP::Robot::Vector>(
7     this->forward->X,this->forward->Y),
8     make_shared<ALF2CPP::Robot::Vector>(this->
9      left->X,this->left->Y),
10    make_shared<ALF2CPP::Robot::Vector>(this->
11     right->X,this->right->Y)});
12
13  while((this->getDirectionWeight(movePrio[0],
14   target) < this->getDirectionWeight(
15   movePrio[1],target))){
16    (movePrio[0] = movePrio[1]);
17    [...];
18  }
19 }
```

Listing 2. Extract of generated C++ for `updateDirection(..)`.

While structural translation of UML models to code is not new, the translation from its standardised action language ALF to C/C++ proposed in SMARTCore is the first of its kind (Listing 2). Moreover, to the best of our knowledge the translation of tasks and their allocation to multithreaded applications in combination with behaviours defined using ALF had not been attempted before SMARTCore. When it comes to the translation of ALF to C/C++, we provide a translation of the syntactical *minimum conformance* (as described in the ALF specification [13]), that is to say the subset of ALF that is used for writing textual action language snippets as behaviours within a larger graphical UML model and that includes all the capabilities available in a traditional, procedural programming language. Moreover, we provide a translation of part of the concepts, addressed as ALF *units*, that are used to textually describe structural portions of a UML model. By providing this option, we allow the developer to, besides defining actions within a graphical UML model, even define the structural parts of the model in terms of ALF to get corresponding executable C++ generated entirely from an ALF model describing a fully-functional software system. A specific way to manage memory is not enforced by UML, while in order to produce executable code we had to provide one. For this reason we investigated a set of possible alternatives and opted for a memory management based on *smart pointers*.[7]

This kind of pointer defines a reference counter for each allocated object. Once the reference counter hits zero, the related object is released from memory. On the one hand, this solution might be costly when dealing with frequent acquisition-release cycles of time-demanding objects. On the other hand, other memory management mechanisms may cause pausing of the whole program execution for the memory to be examined, replaced, or freed. Smart pointers are used to manage objects of non-primitive type (i.e., classes and their instances) in the following way. A non-primitive type `T` is wrapped as **`shared_ptr`**`<T>` (e.g., line 1 of Listing 2), which represents a smart pointer to an object of type `T`. When instantiating the object (i.e., class instance) the construct **`make_shared`**`<T>` (e.g., line 6 of Listing 2) is exploited to initialise the smart pointer referencing to the specific object and taking care of memory management.

In the generated code, each task corresponds to a POSIX[8] thread with read-execute-write semantics meaning that a thread first reads input, then performs calculations and finally writes output. The code used for execution-based optimization is instrumented with code for measuring the end-to-end response times for chains. The experiments were run on the Ubuntu 12.04 LTS operating system (kernel version 3.2.29) patched with the PREEMPT RT patch (version 3.2.29-rt44),[9] to give Ubuntu a hard real-time kernel. Since the approach does not depend on PREEMPT RT's APIs, generated code is portable to other real-time operating systems.

V. LESSONS LEARNED AND PERSPECTIVES

The solutions proposed in the SMARTCore project have been iteratively validated against artificial as well as real in-house and industrial systems. In this section we provide a set of lessons learned and key perspectives for researchers and practitioners to benefit from our experiences and build upon them. More specifically, we discuss on technological aspects, related to the selected languages and tools, and methodological aspects, concerning the actual techniques developed in SMARTCore.

A. TECHNOLOGICAL ASPECTS

1) MODELLING LANGUAGES

We chose UML since it is regarded the de facto standard in industry [7] and the most widely used architectural description language [8]. Moreover, two out of the three industrial partners in SMARTCore already had UML in their

[7]http://www.cplusplus.com/reference/memory/shared_ptr/
[8]http://pubs.opengroup.org/onlinepubs/9699919799
[9]https://rt.wiki.kernel.org/index.php/Main_Page

development process, while the third partner was interested in experimenting with UML. For the first two, there were no issues with the language selection. In order for the latter to evaluate the impact of a hypothetical shift to UML, we had to make a comparison with the modelling language currently used, Simulink/Matlab.[10] The differences between UML and Simulink/Matlab are idiosyncratic and it is not realistic to provide a generic comparison between them. Nevertheless, we could notice some interesting aspects. On the one hand, thanks to its general-purpose nature, UML provides very powerful (object-oriented) abstract and concrete syntaxes for modelling complex software architectures composed of different kinds of hierarchical assemblies and putting together heterogeneous software components in a seamless manner. On the other hand, it is not able to express dynamicity of systems in a flexible and compact way as Simulink/Matlab does. MARTE provides notions for different sample time types, but it is hardly comparable to Simulink/Matlab, which naturally provides 9 sample time types (discrete, continuous, asynchronous, fixed in minor step, inherited, hybrid, constant, variable, triggered). When it comes to data types, UML comes with a minimal set of primitive types, enriched with MARTE's data types specifically for embedded real-time systems. Also in this case, Simulink/Matlab comes with a reacher set of default data types. On the other hand, custom data types can only be defined either in an S-function using C, or in a Level-2 MATLAB S-function by inheritance from other model blocks. UML provides instead metamodelling mechanisms for easily defining custom data types to be exploited using textual and graphical concrete syntaxes for both structural modelling and definition of fine-grained behaviours through action language.

2) TRANSFORMATION LANGUAGES

SMARTCore adheres to the MDE vision of models and model transformations as core development artefacts. In order to automate the various development steps, from generation of analysis and execution artefacts to model-based optimization and back-propagation of analysis/monitoring results to models, we employed models transformations. Among the many model transformation languages available, we evaluated several options and eventually chose QVTo for M2M transformations and Xtend for M2T and T2M transformations. When selecting a transformation language, we took into account its typology, maturity, and, when needed, the effort to include it in a toolchain based on Eclipse. We evaluated the following three different options for M2M transformations:

- Atlas Transformation Language[11]: it is a hybrid (mixed of declarative and imperative) language originally defined for expressing unidirectional model transformations. It is a mature technology and seamlessly integrated in Eclipse.
- Janus Transformation Language[12]: it is a relational language specifically defined for bidirectional non-bijective transformations and change propagation. It has a mature back-end, but it falls short when it comes to usability. It is currently not integrated in Eclipse.
- QVTo: it is an imperative language defined for expressing unidirectional model transformations. It is a mature technology, implements the operational portion of the OMG Query/View/Transformation standard,[13] and it is seamlessly integrated in Eclipse.

Since we needed to define non-bijective transformations, JTL represented our initial first choice. Unfortunately, the effort needed to bring it in our toolchain was heavier than the possible benefits coming along with it. Nevertheless, we have initiated a collaboration with the research group responsible for JTL to improve the usability of the language. Between ATL and QVTo, we selected the latter since it provides a wider spectrum of imperative constructs which could help us in programmatically controlling the non-bijective relations among source (e.g., UML model) and target (e.g., C/C++ code) artefacts.

Concerning M2T and T2M transformations, we considered the following three trasformation languages:

- Acceleo[14]: it is a hybrid template-based language proposing a pragmatic implementation of the OMG Model to Text Language standard. It is a mature technology and seamlessly integrated in Eclipse.
- Xpand[15]: it is an imperative language based on expansion rules specifically defined for code generation. It is a mature technology and seamlessly integrated in Eclipse.
- Xtend: it is an evolution of Xpand. It is regarded as a flexible and expressive dialect of Java and seamlessly integrated in Eclipse.

Since the three languages provide similar features and display the same level of integration in Eclipse, we selected Xtend for two reasons. (1) Xtend's concrete syntax represents a powerful mask on top of Java, enabling the use of both Xtend-specific as well as pure Java constructs under the same hood; since Eclipse is Java-based, the programmatic effort of leveraging Xtend in combination with other Eclipse features was negligible. (2) Xtend code is transparently compiled to Java, which is then executed for performing the defined transformations; being able to exploit the Java Virtual Machine gives Xtend a notable edge compared to Acceleo and Xpand, which are instead interpreted by ad-hoc engines. As an example of this, a transformation defined in Xtend displays an execution time which is close to 10 times lower than the same transformation defined using Xpand.

[10]http://www.mathworks.com/products/simulink/
[11]https://eclipse.org/atl/
[12]http://jtl.di.univaq.it/
[13]http://www.omg.org/spec/QVT/1.1/
[14]https://eclipse.org/acceleo/
[15]http://wiki.eclipse.org/Xpand

3) TARGET LANGUAGES

When it comes to target languages, our code generator produces C/C++, which was unanimously preferred by the three industrial partners. Moreover, no proprietary libraries nor hard-coded domain-specific hacks were accounted in the development of our solutions to not jeopardise the generalisability and dissemination of SMARTCore's results for the community to use and build upon. Java is used for the description of simulation models employed for model-based optimization and SMARTCore provides a set of model transformations for generating them. While the generator of C/C++ is domain-agnostic and reusable in ideally any context where executable C/C++ has to be generated from UML models, the generation of Java simulation models is specifically designed for serving the purpose of the model-based optimization module of our approach.

4) DEVELOPMENT ENVIRONMENT

Since the goal of SMARTCore was to provide an open-source solution to run on state-of-the-art and practice technologies, we chose to leverage Papyrus, which is the de-facto standard open source modelling environment for UML and the reference modelling environment of the PolarSys[16] initiative. Moreover, the Papyrus Industry Consortium[17] has recently been formed and approved by the Eclipse Foundation. At the time of this writing, some Papyrus-related issues with the support of ALF had to be solved within SMARTCore for exploiting it for modelling and code generation. Thanks to the collaboration established during SMARTCore with the research group providing Papyrus, feedback from SMARTCore have helped out in identifying issues related to the ALF implementation in Papyrus.

B. METHODOLOGICAL ASPECTS

1) MODELLING THE SYSTEM

Originally UML was mainly seen as an abstraction of object-oriented 3GLs to be used for better human communication. After 19 years, UML has matured a lot and it has now everything to replace 3GLs in MDE. The majority of tools providing code generation for UML established in industry employs 3GLs rather than proper action languages (i.e., ALF) for defining behaviours within UML models in order to more easily generate code (e.g., [18]–[20]). By doing so, the developer can infer assumptions on the target system (e.g., memory management, parallelism, communication), which hinder the generation of code for different targets from the same input models. In short, while using 3GLs to define behaviours in UML models simplifies code generation since behaviours are already provided in the target language, it complicates model validation, analysis, consistency checking and generally imperils many benefits of using models. In order to avoid that, in SMARTCore we embraced the recently formalised platform-neutral action language for UML, ALF, for which we provided the first of its kind translator to C++ [21].

By employing an action language like ALF, action code is given full knowledge of the surrounding model [22], [23] and this brings about several benefits, among which simplified model-based analysis, model simulation and consistency checking at modelling level. When it comes to the reusability of models, since ALF is not bound to any specific target platform, code generators like the one we defined in SMARTCore can target different variations (providing some degree of reusability for code generators too) of one language (e.g., different memory management mechanisms depending on the user's selection) or different languages, from the same model. Regarding user experience of using ALF instead of C/C++ for defining behaviours in the UML models, industrial actors in SMARTCore agreed in finding it very intuitive thanks to its Java-like syntax and they appreciated the naturalness by which it seamlessly integrates with UML. Clearly, some domain-specific C/C++ constructs could not be used and a workaround using ALF concepts had to be found.

2) AUTOMATIC CODE GENERATION

Working on code generation, we experienced that translation of UML models to 3GLs still raises two fundamental issues [24]. (1) The intermediate translation step from source model to 3GL code creates discontinuity between the final executable and its source model. This is exasperated by the fact that translators from UML to 3GLs do not *account UML-specific semantics*. Discontinuity, meant as executables which are *not semantically consistent to their source models* and *suffer from information loss*, undermines model observability (e.g., model debugging, models@runtime). (2) Existing compilers are designed to fully exploit the semantics of their language (i.e., in our case C/C++). This allows them to implement numerous time and memory optimizations. However, they are *unable to exploit the UML semantics*; the result is *suboptimal* executables. Consequently, while suboptimality, a certain level of information loss and semantic inconsistencies are bearable in non-critical domains, they have historically hindered the full embracement of MDE and UML in safety-critical domains where predictability is key [25]. We argue that direct compilation of UML by-passing the use of 3GLs is needed to (i) provide full control over the manipulations that models undergo to produce executables and (ii) enable optimizations based on model-based analysis that can not be achieved by translating to 3GLs. More specifically, a compiler specifically designed for UML would diminish semantic issues, enhance model observability, enable fruitful optimizations at model-level and eventually enable the generation of optimised and more predictable executables.

[16]https://www.polarsys.org/

[17]https://www.polarsys.org/ic/papyrus

A specific technique for memory management in UML is not enforced but rather left intentionally open to several options. When generating code we needed to choose a suitable memory management mechanism for the generated code and we opted for smart pointers. The alternatives we investigated were:

- Allocate basic typed variables on the stack and more complex objects on the heap. This solution would result in decent code performance and it would be fairly easy to implement. The issue with it resides in the fact that it does not ensure prevention from stack overflows since allocations of basic typed variables may require more space than available on the call stack.
- Allocate everything on the heap through *smart pointers*. This solution would give good code performance and would not be extremely difficult to implement. Moreover, it would prevent from stack overflows and memory leaks, since none of the objects are allocated on it.
- Perform a smart allocation based, e.g., on the scope of use and the size of the objects. On the one hand, this solution would provide the best code performance and prevent from stack overflows. On the other hand, it would be very complex to implement and maintain since it would require (i) an analytical engine to determine object-specific allocations and (ii) a way to automatically generate destructors to handle memory deallocation of user-defined objects.

The main focus in SMARTCore was to achieve a sound solution that was not too costly to develop and maintain, while providing the possibility to be complemented with other solutions. That is why we decided to go for allocation on the heap through smart pointers. Performance-wise, smart pointers add two counting variables to manage the memory needed by an object. On the one hand, this represents a tiny overhead in memory usage, usually outplayed by the security that is provided by smart pointers; very few modern embedded systems would not be able to afford such a small overhead. On the other hand, for hard real-time applications whose criticality resides in their timeliness, a deeper analysis would be needed to assess how smart pointers can affect deterioration from a timing perspective.

Overall, we believe that effort should be spent to define mechanisms for guiding the automatic generation of executables in achieving smart memory management in terms of *minimisation of dynamic memory handling* and *optimization of the tradeoff between allocations on heap and stack*. The possibility to allocate on stack, in fact, brings about several advantages for real-time and safety-critical embedded systems: automatic, faster and more time-predictable allocation/de-allocation operations, reduced overhead for the management of the objects in the slower heap, no need for synchronization on local objects (synchronization elision), and reduced memory fragmentation [26].

3) ALLOCATION OPTIMIZATION

In SMARTCore we exploit a combined model-based and execution-based optimization mechanism. As aforementioned, during the optimization process new allocation configurations to be analysed are identified. The identification could be done randomly by relocating a task to a random core. For efficiency reasons (i.e., make the optimization faster), we exploited the delay matrix heuristic. This heuristic takes into consideration the fact that the most problematic task on a core is either the one that considerably delays tasks that belong to the chain to be optimised, or the one that is considerably delayed by other tasks. For identifying a new allocation, the heuristic takes the current best allocation specification and moves the problematic task to a core with low load. Clearly, this is only one possibility, which is particularly suitable for the purposes of SMARTCore. In different application scenarios, other heuristics, even partially random, could be preferred and thereby should be evaluated against the delay matrix.

A relevant aspect of the combined optimization mechanism is the selection of the point when the change from model-based to execution-based module should happen. We tried to provide a general rule that would work no matter what the scenario was, but we noticed that this was not achievable. The reason is that the switch from model-based to execution-based modules depends on (i) how well they agree on what a good allocation is, and (ii) the difference in time needed for obtaining performance predictions and performance measurements. Regarding (i), if performance prediction is not precise enough, the model-based module could misdirect the optimization before the execution-based module takes over. Regarding (ii), if the time taken by the model-based module is comparable to the one taken by the execution-based one, there is no real reason for the model-based module to run for many iterations (or even to run at all!). On the contrary, if the two modules have similar accuracy, then it is worth running the model-based module for many iterations before handing over to the execution-based module, since the latter is usually slower. Similarly, when model-based optimization is much faster than execution-based, than it is worth to run the model-based module for a higher number of iterations. If performance predictions could be obtained analytically (without simulation), the model-based module would be much faster and could thereby process a much larger portion of the solution space than the execution-based module in the same amount of time. We believe that more effective (i.e., compact) ways for modelling and analysing the solution space of allocation configurations shall be sought. In particular we advocate the exploitation of what in decision theory is called *known uncertainty* [27] for compact modelling of the solution space to be leveraged by parametric model-based analyses which should be able to exploit explicitly modelled uncertainty.

Future perspectives

- The use of action languages compliant to UML such as ALF is crucial to avoid that action code specified using a different language would violate standard UML semantics. To this end, we believe that a stronger effort needs to be put on the mechanisms to support the use of such UML-compliant action languages.
- The combination of model-based and execution-based optimization mechanisms is very valuable as long as they are effectively intertwined and rely on suitable heuristics. For improving the benefits given by this combination, model-based optimization should be made more effective. Our suggestion is to enhance modelling and model-based analysis to entail known uncertainty explicitly modelled for expressing a large solution space in a much more compact manner.
- Bidirectional transformation languages can represent the key for non-bijective relations among artefacts as in the case of code generation. For them to be valid options in industrial settings, an engineering effort is needed to make them more accessible and usable.
- In order to increase predictability in terms of timeliness and memory management, efforts in the direction of direct compilation of modelling languages, by-passing translation to 3GLs as execution enabler, are essential. Direct compilation of models to executables will boost the adoption of MDE and UML for modern real-time and safety-critical systems (e.g., cyber-physical systems, internet-of-things, autonomous systems).
- For the same reasons, we believe that efforts in providing smart memory management in terms of minimisation of dynamic memory handling and optimization of the tradeoff between allocations on heap and stack are needed.

VI. RELATED WORK

Several approaches have been proposed for model-driven engineering of embedded real-time systems both for singlecore [28] as well as for multicore systems starting from different abstraction levels. Piat et al. [29] propose the generation of static C code for multicore systems starting from synchronous data flow graphs. Cha et al. [30] present an automatic code generation scheme for multicore systems by inserting user-defined S-Functions for Simulink applications. An approach operating at a very low level of abstraction is proposed by Collins et al. [31] in terms of Huckleberry, a tool for automatically generating parallel implementations for multicore platforms from sequential recursive divide-and-conquer programs. Overall, in current approaches inputs for the generation process vary from source code to formal mathematical models, thus in contrast with SMARTCore not considering more intuitive graphical design models as starting point for the generation of multicore systems. In [32] the authors provide a tool which automates the generation of the System-Level Architecture Model (S-LAM) from a UML annotated with MARTE in order to ease the development towards Massively Parallel Multi-Processors System-on-Chip (MP2SoC). Nevertheless the proposed approach does not automatically generate the entire executable code from UML models.

Navabi et al. [33] in the early 90's, and some years later Mahadevan and Armstrong [34], came up with different approaches for back-annotating behavioural descriptions with timing information; however, both operate horizontally[18] in terms of abstraction levels and no automation is provided. Hegedüs et al. [35] propose back-propagation for enabling execution traces retrieved by model checkers or simulation tools to be integrated and replayed in modelling frameworks. Guerra et al. [36] address back-propagation of analysis results to the original model by means of triple graphical patterns. The mentioned approaches are all meant to horizontally operate at modelling level with propagation of data among models. Apart from horizontal back-propagation, SMARTCore focused on vertically propagating analysis results observed at code level back to design models for better understanding and exploiting hard-to-predict EFPs [37].

Regarding measurements of EFPs at system implementation level, besides runtime monitoring, other verification techniques (e.g., static analysis) can be used for small and simple systems, but their application for large and complex systems might not always be practical and economical [38]. Even when feasible, conditions that cause invalidation of the analysis results at runtime may happen. An example is the difference between the ideal execution environment (considered for performing analysis) and the actual one which leads to the violation of the assumptions that were taken into account when performing static analysis [39]. Therefore, the information gathered through monitoring the execution of a system is not only interesting and useful for observing the actual behaviour and to detect violations at runtime, but also to be used for making adaptation decisions, as well as to induct enforcement and preservation of properties. Saadatmand et al.'s work [40] serves as an example of using monitoring information for balancing timing and security properties in embedded real-time systems; in SMARTCore we leveraged similar solutions. In [41] the authors try to estimate the effects of monitor overheads on different types of embedded multicore applications and give suggestions on the number and type of monitors to use for those. This kind

[18]Horizontal and vertical are used for specifying the direction of data transitions among artefacts either at the same (i.e., horizontal, from model to model) or at different (i.e., vertical, from code to model) level of abstraction.

of approaches were investigate in SMARTCore, but did not fit in the project's scope.

In the literature there exist approaches dealing with allocation optimization based on measurements at system implementation level as described in [42]. The COMPAS framework by Mos and Murphy [43] is a performance monitoring approach for J2EE systems. For performance prediction of the modelled scenarios, the approach suggests using existing simulation techniques, which are not part of the approach. Based on the COMPAS framework, two further approaches have been proposed: AQUA, by Diaconescu and Murphy [44], and PAD, by Parsons and Murphy [45]. The goal of these approaches is to identify performance problems in the running system and adapt the implementation (at code level) to make it able to fulfil EFPs requirements. Instead, the uniqueness of the SMARTCore approach consists in introducing a new dimension to allocation optimization at model level to be used in combination with measurements gathered at system implementation level, as introduced in [46].

VII. CONCLUSION

In this paper we described the solutions for UML-based development of embedded real-time software on multicore provided within the SMARTCore project. More specifically, we described how we achieved modelling, combined model-based and execution-based task allocation optimization, and automatic code generation. For researchers and practitioners to leverage our experiences in SMARTCore, we described a set of lessons learned related to both technological and methodological aspects and we stressed five key perspectives that we find fundamental for boosting the adoption of (UML-based) MDE of embedded real-time systems in industry. A follow-up of the SMARTCore project is already being planned and will aim at investigating some of these perspectives as follows.

The experiments carried out to evaluate the optimization mechanisms resulted in the combined optimization being on average more efficient than model-based and execution-based modules singularly. Nevertheless, the scope of the optimization was narrow since we aimed at optimising end-to-end response times and deadline misses of task chains only. It would certainly be valuable to run further experiments with the aim of optimising a multiple set of extra-functional properties to better understand under which circumstances the combined approach is favourable.

Concerning code generation, an interesting direction to enhance it would be to make the involved transformations configurable by the modeller through parameters in order to guide the generation towards a specific target. For instance, the modeller could be given a set of different mechanisms for memory management among which to select the one to use when generating code. From a methodological perspective, we have already started an effort towards direct compilation of UML and ALF, without intermediate translations to programming languages such as C++, to maximise the preservation of execution semantics of UML and ALF in the generated executables.

As aforementioned, smart memory management, meant as the minimisation of dynamic memory handling and the optimization of the tradeoff between allocations on heap and stack, is key for the production of predictable executables suitable for embedded real-time and safety-critical systems. We are planning an investigation on the theory behind escape [26] and unbounded structure analysis [47] as well as memory management for real-time object-orientation [48] to understand whether and how they can be applied to the compilation of UML models.

REFERENCES

[1] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4, Dec. 1996, Art. no. 196.

[2] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, 1992.

[3] F. Brooks, Jr., *No Silver Bullet: Essence and Accidents of Software Engineering*. New York, NY, USA: IEEE Press, Apr. 1987.

[4] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006.

[5] S. Kent, "Model driven engineering," in *Proc. IFM*, 2002, pp. 286–298.

[6] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, no. 3, pp. 621–645, 2006.

[7] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proc. ICSE*, 2011, pp. 471–480.

[8] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 869–891, Jun. 2013. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.74

[9] R. Chapman, "Correctness by construction: A manifesto for high integrity software," in *Proc. SCS*, 2006, pp. 43–46.

[10] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Softw. Syst. Model.*, vol. 7, no. 3, pp. 345–359, 2008.

[11] F. Ciccozzi, D. Corcoran, T. Seceleanu, and D. Scholle, "SMARTCore: Boosting model-driven engineering of embedded systems for multicore," in *Proc. ITNG*, 2015, pp. 89–94.

[12] Object Management Group. (2011). *UML Superstructure Specification V2.3*, accessed on Apr. 11, 2012. [Online]. Available: http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

[13] OMG. *Action Language for Foundational UML*, accessed on Oct. 8, 2016. [Online]. Available: http://www.omg.org/spec/ALF/1.0.1/

[14] OMG. (2011). *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, V1.1*. [Online]. Available: http://www.omg.org/spec/MARTE

[15] F. Budinsky, D. Steinberg, and E. Merks, *Eclipse Modeling Framework*. Reading, MA, USA: Addison-Wesley, 2003.

[16] J. Feljan and J. Carlson, "Task allocation optimization for multicore embedded systems," in *Proc. SEAA*, 2014, pp. 237–244.

[17] J. Feljan, F. Ciccozzi, J. Carlson, and I. Crnkovic, "Enhancing model-based architecture optimization with monitored system runs," in *Proc. SEAA*, 2015, pp. 216–223.

[18] E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini, "SystemC/C-based model-driven design for embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 8, no. 4, 2009, Art. no. 30.

[19] F. Mischkalla, D. He, and W. Mueller, "Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems," in *Proc. DATE*, 2010, pp. 1201–1206.

[20] J. Vidal, F. de Lamotte, G. Gogniat, P. Soulard, and J.-P. Diguet, "A co-design approach for embedded system modeling and code generation with UML and MARTE," in *Proc. DATE*, 2009, pp. 226–231.

[21] F. Ciccozzi, "Dethroning programming languages as endorsed means for fine-grained UML behaviour modelling in open source MDE," in *Proc. OSS4MDE*, 2015, pp. 44–53.

[22] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier, "Formalizing execution semantics of UML profiles with fUML models," in *Proc. MODELS*, 2014, pp. 133–148.

[23] B. Selic, "The less well known UML," *Formal Methods for Model-Driven Engineering*, vol. 7320. Berlin, Germany: Springer, 2012, pp. 1–20.
[24] B. Selic, "What will it take? A view on adoption of model-based methods in practice," *Softw. Syst. Model.*, vol. 11, no. 4, pp. 513–526, 2012.
[25] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proc. RTSS*, 2001, pp. 73–83.
[26] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, "Stack allocation and synchronization optimizations for Java using escape analysis," *ACM Trans. Programm. Lang. Syst.*, vol. 25, no. 6, pp. 876–910, 2003.
[27] C. C. Chow and R. K. Sarin, "Known, unknown, and unknowable uncertainties," *Theory Decision*, vol. 52, no. 2, pp. 127–138, 2002.
[28] J. Porter *et al.*, "The ESMoL modeling language and tools for synthesizing and simulating real-time embedded systems," in *Proc. 15th IEEE Real-Time Embedded Technol. Appl. Symp.*, San Francisco, CA, USA, Apr. 2009.
[29] J. Piat, S. S. Bhattacharyya, M. Pelcat, and M. Raulet, "Multi-core code generation from interface based hierarchy," in *Proc. DASIP*, 2009.
[30] M. Cha, K. H. Kim, C. J. Lee, D. Ha, and B. S. Kim, "Deriving high-performance real-time multicore systems based on simulink applications," in *Proc. DASC*, 2011, pp. 267–274.
[31] R. L. Collins, B. Vellore, and L. P. Carloni, "Recursion-driven parallel code generation for multi-core platforms," in *Proc. DATE*, 2010, pp. 190–195.
[32] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid, "Automatic generation of S-LAM descriptions from UML/MARTE for the DSE of massively parallel embedded systems," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Berlin, Germany: Springer, 2016, pp. 195–211.
[33] Z. Navabi, S. Day, and M. Massoumi, "Investigating back annotation of timing information into dataflow descriptions," in *Proc. VIUF*, 1992, pp. 185–195.
[34] G. Mahadevan and J. R. Armstrong, "Investigating back annotation of timing information into dataflow descriptions," in *Proc. VIUF*, 1995.
[35] A. Hegedüs, G. Bergmann, I. Ráth, and D. Varró, "Back-annotation of simulation traces with change-driven model transformations," in *Proc. SEFM*, 2010, pp. 145–155.
[36] E. Guerra, D. Sanz, P. Díaz, and I. Aedo, "A transformation-driven approach to the verification of security policies in Web designs," in *Proc. ICWE*, 2007, pp. 269–284.
[37] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "Round-trip support for extra-functional property management in model-driven engineering of embedded systems," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1085–1100, 2012.
[38] A. Wall, J. Kraft, J. Neander, C. Norström, and M. Lembke, "Introducing temporal analyzability late in the lifecycle of complex real-time systems," in *Proc. RTCSA*, 2003, pp. 513–528.
[39] S. E. Chodrow, F. Jahanian, and M. Donner, *Monitoring and Debugging of Distributed Real-Time Systems*. France: ECSI, 1995, pp. 103–112.
[40] M. Saadatmand, A. Cicchetti, and M. Sjödin, "Design of adaptive security mechanisms for real-time embedded systems," in *Proc. ESSoS*, 2012, pp. 121–134.
[41] S. Chun-Yi *et al.*, "Adaptive performance monitoring for embedded multicore systems," in *Proc. ICWPP*, Sep. 2011, pp. 222–228.
[42] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, 2010.
[43] A. Mos and J. Murphy, "A framework for performance monitoring, modelling and prediction of component oriented distributed systems," in *Proc. WOSP*, 2002, pp. 235–236.
[44] A. Diaconescu and J. Murphy, "Automating the performance management of component-based enterprise systems through the use of redundancy," in *Proc. ASE*, 2005, pp. 44–53.
[45] T. Parsons and J. Murphy, "Detecting performance antipatterns in component based enterprise systems," *J. Object Technol.*, vol. 7, no. 3, pp. 55–90, 2008.
[46] F. Ciccozzi, M. Saadatmand, A. Cicchetti, and M. Sjödin, "An automated round-trip support towards deployment assessment in component-based embedded systems," in *Proc. CBSE*, 2013, pp. 179–188.
[47] I. Grabe, "Static analysis of unbounded structures in object-oriented programs," Ph.D. dissertation, Leiden Inst. Adv. Comput. Sci., Faculty Sci., Leiden Univ., Leiden, The Netherlands, 2012.
[48] J. M. Enery, D. Hickey, and M. Boubekeur, "Empirical evaluation of two main-stream RTSJ implementations," in *Proc. JRTES*, 2007, pp. 47–54.

FEDERICO CICCOZZI (M'09) received the Ph.D. degree from the School of Innovation, Design and Engineering, Mälardalen University, Sweden, in 2014. In his research activity, he was with several companies and research institutions, such as Ericsson, ABB, Alten, Thales, and CEA list. He is currently an Assistant Professor with the School of Innovation, Design and Engineering, Mälardalen University. His research focuses on the definition of metamodels and model transformations for several automation aspects in the model-driven development of component-based embedded real-time systems, such as code generation, preservation of system properties, back-propagation, multi-paradigm modeling, model versioning, (co)evolution and synchronisation, and the application of model-driven and component-based techniques to (multi-)robot systems. He has co-authored over 35 publications in journals and international conferences and workshops in these areas. He has been serving the community as the Conference Track and Workshop Organiser, the Expert Panelist, the Program Committee Member, and a Reviewer for conferences, workshops, and international journals.

TIBERIU SECELEANU (SM'04) received the Ph.D. degree in computer science from Åbo Akademi, Turku, Finland. He is currently a Principal Scientist with ABB Corporate Research and an Adjunct Professor of Computer Engineering with Mälardalen University, Sweden. His research interests are in embedded system architectures and design methodologies, formal methods in system design, and wireless communication.

DIARMUID CORCORAN has many years experience in software architecture and the development of embedded systems with deep knowledge on model-driven techniques applied to large-scale projects in industry. He was an Expert and an Industrial Ph.D. Student with the Research and Development Department, Ericsson AB. He is interested in exploiting high-level UML modeling formalisms for the description of embedded systems and the automatic generation of full-fledged code optimized for multicore.

DETLEF SCHOLLE has excellent knowledge of real-time embedded systems, from small device components to large distributed systems. He was the Program Manager with Alten Sweden AB, where he has been involved in several embedded real-time systems research projects, such as DysCAS, MBAT, iFEST, Frames, GEODES, and PDFRAME. His main interest is on robust, dependable and reliable embedded systems in diverse applicative domains.