Softw Syst Model (2018) 17:479–508
https://doi.org/10.1007/s10270-017-0585-x

THEME SECTION PAPER

Using UML/MARTE to support performance tuning and stress testing in real-time systems

Stefano Di Alesio[1] · **Sagar Sen**[1]

Received: 7 July 2015 / Revised: 12 August 2016 / Accepted: 19 January 2017 / Published online: 10 February 2017

Abstract Real-time embedded systems (RTESs) operating in safety-critical domains have to satisfy strict performance requirements in terms of task deadlines, response time, and CPU usage. Two of the main factors affecting the satisfaction of these requirements are the configuration parameters regulating how the system interacts with hardware devices, and the external events triggering the system tasks. In particular, it is necessary to carefully tune the parameters in order to ensure a satisfactory trade-off between responsiveness and usage of computational resources, and also to stress test the system with worst-case inputs likely to violate the requirements. Performance tuning and stress testing are usually manual, time-consuming, and error-prone processes, because the system parameters and input values range in a large domain, and their impact over performance is hard to predict without executing the system. In this paper, we provide an approach, based on UML/MARTE, to support the generation of system configurations predicted to achieve a satisfactory trade-off between response time and CPU usage, and stress test cases that push the system tasks to violate their deadlines. First, we devise a conceptual model that specifies the abstractions required for analyzing task deadlines, response time, and CPU usage, and provide a mapping between these abstractions and UML/MARTE. Then, we prune the UML/MARTE metamodel to only contain a purpose-specific subset of entities needed to support performance tuning and stress testing. The pruned version is a supertype of UML/MARTE, which ensures that all instances of the pruned metamodel are also instances of UML/MARTE. Finally, we cast the generation of configurations and stress test cases as two constrained optimization problems (COPs) over our conceptual model. The input data for these COPs in automatically generated via a model-to-text (M2T) transformation from models specified in the pruned UML/MARTE metamodel to the Optimization Programming Language. We validate our approach in a safety-critical RTES from the maritime and energy domain, showing that (1) our conceptual model can be applied in an industrial setting with reasonable effort, and (2) the optimization problems effectively identify configurations predicted to minimize response time and CPU usage, and stress test cases that maximize deadline misses. Based on our experience, we highlight challenges and potential issues to be aware of when using UML/MARTE to support performance tuning and stress testing in an industrial context.

Keywords UML/MARTE · Real-time systems · Safety-critical systems · Performance tuning · Stress testing · Constrained optimization

1 Introduction

Failures in safety-critical systems, such as those in the energy, transport, and healthcare domains, could result in catastrophic consequences [1]. Therefore, the safety-related software components of these systems are usually subject to strict performance requirements, involving hard real time, soft real time, and resource utilization constraints [2]. In particular, three performance requirements that are commonplace in safety-critical systems concern *task deadlines*,

Communicated by Dr. Kai Sachs and Catalina Llado.

✉ Stefano Di Alesio
stefano@simula.no

Sagar Sen
sagar@simula.no

[1] Certus Centre for Software Verification and Validation, Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway

response time, and *CPU usage* [3]. Specifically, task deadlines state that the system tasks should always terminate before a given completion time, entailing that even a single deadline miss severely compromises the system operational safety. Response time requirements state that the system should react to external inputs within a specified time. Finally, CPU usage constraints state that the system should always keep a given percentage of free CPU time, to avoid that high computational load prevents the system to timely respond to safety-critical alarms.

However, safety-critical systems are progressively relying on real-time embedded systems (RTESs), where software applications interact with the environment through sensors and actuators [4]. In large and complex RTESs, the software components often communicate with a large number of different devices, whose interface is provided by software drivers. In particular, one of the main goals of device drivers is to provide a smooth data transfer between hardware devices and software components. This is especially true in safety-critical systems, where external data should always be processed in brief time to guarantee a prompt reaction to critical events [5]. Therefore, device drivers are usually designed as concurrent applications, whose task timing has to be configured in order to correctly operate with the specific devices connected. Nonetheless, tuning the timing of driver tasks without violating constraints on task deadlines, response time, and CPU usage, is complicated by two main factors [6]. First, the drivers parameters related to temporal properties, such as task delay times, offsets, and periods, range in a large domain of values, typically expressed in milliseconds. Second, the impact specific parameter values have over the system performance is hard to evaluate without executing the system. This is because, in concurrent systems where tasks depend on each other, a minimal variation in a single task timing may trigger unpredictable interactions between other tasks.

Furthermore, device drivers are often subject to software safety certification, whose purpose is to assure that the system is deemed safe for operation. Widely used safety standards, such as IEC 61508 and IEC 26262, state that performance testing is highly recommended for the highest Safety Integrity Levels (SIL) [7]. In particular, these standards remark the importance of stress testing, whose goal is to identify scenarios that exercise a system in a way to either violate performance requirements, or be as close as possible to doing so [8]. These worst-case scenarios are usually characterized in terms of reproducible environmental conditions, such as external events triggering the system tasks. However, the timing of these events varies in a large domain depending on the environment state, and can never be fully predicted prior to system execution. For this reason, stress-testing task deadlines, response time, and CPU usage, poses challenges similar to those of performance tuning.

As a consequence, it is often practice in industry to carry out performance tuning and stress testing by relying only on the engineers expertise and knowledge of the system. This renders tuning drivers parameters and identifying worst-case scenarios significantly time-consuming and error-prone processes [9]. Nevertheless, research in this direction states the benefits of model-based approaches performing software performance analysis at early development stages [10]. In particular, recent approaches integrate performance analysis and verification in model-driven engineering (MDE) development processes [11], applying UML/MARTE in industrial settings [12]. However, MARTE is a very large profile, and its effectiveness in industrial projects is subject to the availability of guidelines targeting specific needs [13].

Traditionally, performance verification has mostly been addressed through model checking approaches [14], which require complex formal modeling of the system that often leads to the well-known state explosion problem [15]. To overcome these practical limitations, approaches based on metaheuristic search have been proposed for both identifying configuration parameters likely to satisfy CPU usage requirements [16], and to stress test task deadlines of real-time systems [17]. However, our previous experiments [18] suggest that complete search strategies, such as those based on Constraint Programming (CP), can potentially be more effective than Genetic Algorithms (GA) in finding solutions closer to the global optimum, and are hence worth investigating for performance tuning and stress testing.

In this paper, we propose a methodology that combines UML/MARTE modeling and constraint programming to support performance tuning and stress testing of RTESs. First, we provide a conceptual model that captures, independently from any modeling language, the abstractions required to support performance tuning and stress testing of task deadlines, response time, and CPU usage. Then, we map our conceptual model to UML/MARTE, effectively bridging the gap between the complexity of the standard and the more focused scope of performance tuning and stress testing. The subset of UML/MARTE mapped to out conceptual model contains stereotypes and stereotype properties extending entities in UML class, deployment, and sequence diagrams, which are popular for modeling concurrent systems such as RTESs, and intuitive to most developers [19]. The conceptual model and its mapping to UML/MARTE enables casting (1) performance tuning and (2) stress testing as constraint optimization problems (COPs) over our conceptual model. Specifically, the COPs aim at identifying (1) scenarios characterized by tunable task parameters, i.e., *configurations*, where tasks are as far as possible from their deadlines, and exhibit low response time and CPU usage [20], and, (2) scenarios characterized by external events, i.e., *stress test cases*, where tasks are as likely as pos-

sible to violate their requirements on task deadlines, response time and CPU usage [21]. We use a metamodel pruning approach [22] on the UML/MARTE metamodel, in order to obtain a smaller metamodel that is also a super type [23] of UML/MARTE. This implies that all instances of the pruned metamodel are also instances of the original UML/MARTE metamodel and all transformations developed for the pruned UML/MARTE metamodel, such as the mapping from our conceptual model, are also reusable for instances of the original metamodel [24]. This pruned metamodel forms the basis to define a mapping between the UML/MARTE stereotypes and stereotype properties required for our analysis, and the Optimization Programming Language (OPL) [25], a widely used language for specifying COPs. Specifically, OPL separates the definition of the model logic, i.e., constants, variables, constraints and objective functions, from the input data. In particular, we use OPL to encapsulate the data needed to automate the search for system configuration and stress test cases. The mapping between the pruned UML/MARTE and OPL is implemented through a model-to-text (M2T) transformation in Acceleo, an open-source implementation of the OMG MOF model-to-text language (MTL) standard. We validate our modeling approach on a RTES from the maritime and energy domain concerning safety-critical device drivers, showing that our modeling guidelines can be applied in an industrial setting with reasonable overhead. In particular, we report results from previous work showing that the COPs enabled by our conceptual model effectively find configurations satisfying, and stress test cases violating the system performance requirements. Finally, we discuss our experience on applying UML/MARTE for performance tuning and stress testing, outlining challenges and potential issues that practitioners may face in industrial contexts.

Contributions of this paper The contributions of this paper build upon our previous work in the area of performance tuning and stress testing. Specifically, we initially considered the problem of generating stress test cases characterized by task arrival times that maximized deadline misses [26], response time, and CPU Usage [27]. In particular, we developed an early version of the conceptual model for the purpose of supporting stress testing of CPU usage requirements in RTESs [27]. Then, we compared our constraint-based approach with metaheuristic search techniques [18], improving the data structures of our model [21]. We illustrated how a combination of constraint programming and genetic algorithms is more likely to scale to large systems [28], and we finally focused on generating system configurations characterized by task delay times predicted to satisfy the system performance requirements [20]. Specifically, we summarize the contributions of this paper as follows:

1. We revise the first version of our conceptual model and its mapping to UML/MARTE [27] to include support for both performance tuning and stress testing.
2. We prune the UML/MARTE metamodel to a smaller one containing only 26 concepts. This pruned metamodel is a subset, and supertype of UML/MARTE. The model type matching ensures that all instances of the pruned version are also valid UML/MARTE instances, and all mappings on the pruned metamodel are also correct mappings for the original UML/MARTE profile.
3. We develop a mapping between the pruned UML/MARTE metamodel and OPL. This mapping is implemented through a model-to-text transformation in Acceleo, and is a significant step toward the full automation of our approach for performance tuning and stress testing.
4. We highlight our experiences and lessons learned toward a UML/MARTE framework for performance tuning and stress testing. In particular, we discuss on the need of methodologies building on UML/MARTE, and point out potential issues in the definition of the metamodel that could render UML/MARTE models potentially hard to understand.

Structure of the paper This paper is structured as follows. In Sect. 2 we introduce our motivating case study, detailing the industrial context and the challenges posed by developing safety-critical I/O drivers. We discuss related model-based approaches for software performance tuning and stress testing in Sect. 3. We initially present an overview of our approach for performance tuning and stress testing in Sect. 4, introducing our COP for the generation of configurations and stress test cases predicted to satisfy and violate performance requirements, respectively. We present the contributions of the paper, detailing our modeling choices in Sect. 5 and summarize past validation results in Sect. 6. Finally, we report our experience on using UML/MARTE in an industrial context, together with potential alternatives and limitations of our approach in Sect. 7. We conclude the paper in Sect. 8.

2 Motivating case study

The main motivation behind our work originates from a case study in the maritime and energy domain concerning a fire and gas monitoring System (FMS) in oversea oil extraction platforms. The FMS is developed by Kongsberg Maritime (KM),[1] a leading company in the production of systems for positioning, surveying, navigation, and automation to merchant vessels and offshore installations. The goal of the system is to monitor potential gas leaks, and trigger an alarm in case a fire is detected. The system monitors and displays to

[1] http://www.km.kongsberg.com.

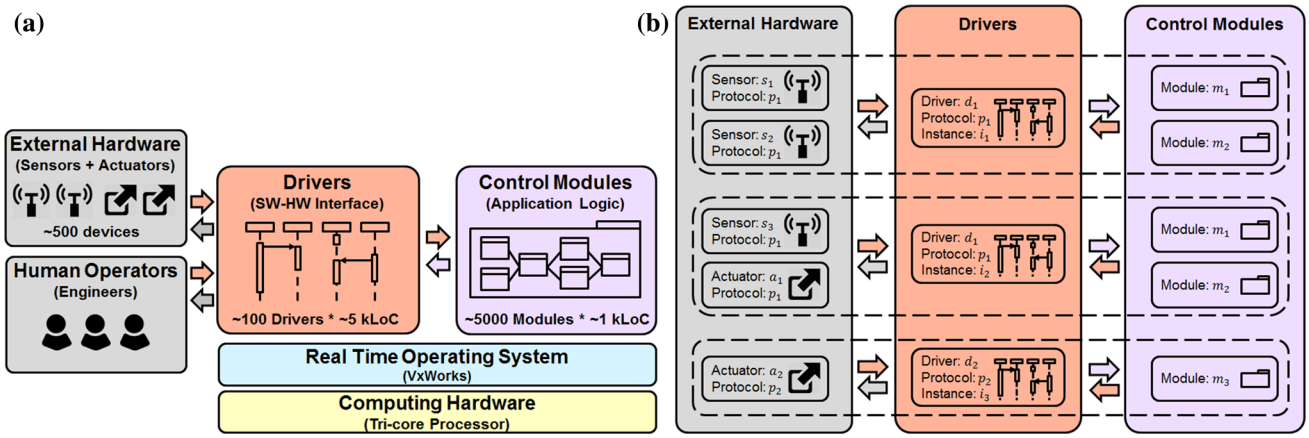

Fig. 1 Description of the fire and gas monitoring system (FMS). **a** Architecture of the FMS. **b** An example showing three communication scenarios in the FMS

human operators data coming from smoke/heat detectors, and gas flow sensors. When the system receives critical data from the hardware sensors, it automatically triggers actuators, such as fire sprinklers and audio/visual alarms. Technicians constantly monitor the system, and can also directly interact with it, for instance to manually tune operational parameters or control events raised by incoming data. The FMS software architecture is shown in Fig. 1.

The software part of the system consists of drivers and control modules. Drivers implement I/O communication between the system and the external environment, such as hardware sensors, actuators, and human operators. Control modules implement the application logic of the FMS, i.e., they process data coming from the environment and accordingly decide the operations to perform. Drivers and control modules are the main software components of the FMS, and run on a Real-Time Operating System (RTOS), namely VxWorks,[2] that is configured with a fixed-priority preemptive scheduling policy, where task priorities are statically defined as part of system design. VxWorks is installed on a tri-core computing platform. This architectural design is common in many industry sectors relying on embedded systems [29]. The whole FMS consists of approximately 5000 control modules and 500 sensors and actuators that communicate with the system through more than 100 different drivers. Drivers and control modules have an approximate size of 5 and 1 thousands lines of C/C++ source code (kLoC) each.

In typical FMS operating scenarios, drivers communicate externally with sensors and/or actuators, and communicate internally with control modules. To cope with the large number of external hardware devices, the system runs in parallel several instances of each driver. Moreover, the variety of sensors and actuators built by different vendors motivates the need for having several types of drivers. Indeed, each FMS driver implements a specific communication protocol, and hence communicates only with sensors and actuators that implement the same protocol. Figure 1b shows an a example of three communication scenarios in the FMS, represented by a dashed rectangle. In the scenario at the top, the instance i_1 of driver d_1, which implements the protocol p_1, communicates with the sensors s_1 and s_2 and the control modules m_1 and m_2. In the scenario at the center, the instance i_2 of d_1 communicates with the sensor s_3, the actuator a_1, and the modules m_1 and m_2. In the scenario at the bottom, the instance i_3 that runs the driver d_2 communicates with the actuator a_2 and the module m_3. Note that, in each scenario, the driver implements the same communication protocol of the sensors and actuators involved. The key entities of the FMS which are relevant to our study are shown in Fig. 2.

Drivers constitute the most critical part of the FMS. Indeed, one of the main complexity factors in drivers is that they are meant to bridge the timing discrepancies between hardware devices and software controller modules. Hence, their design typically consists of concurrent tasks that communicate asynchronously to smooth the data transfer between hardware and software components. Therefore, drivers are subject to strict requirements to ensure that their flexibility does not come at the cost of performance. Specifically, in each FMS driver (1) no task should miss its deadline, (2) the response time should be <1 s, and (3) the CPU usage should be below 20%. Note that the FMS is a hard real-time system, for which missing even a single deadline severely threatens the system safety. For this reason, the FMS cannot have fault tolerance and reconfiguration mechanisms with respect to these requirements, as it often happens in firm and soft-real-time systems [30]. The constraints on task deadlines, response time, and CPU usage

[2] http://www.windriver.com/products/vxworks.

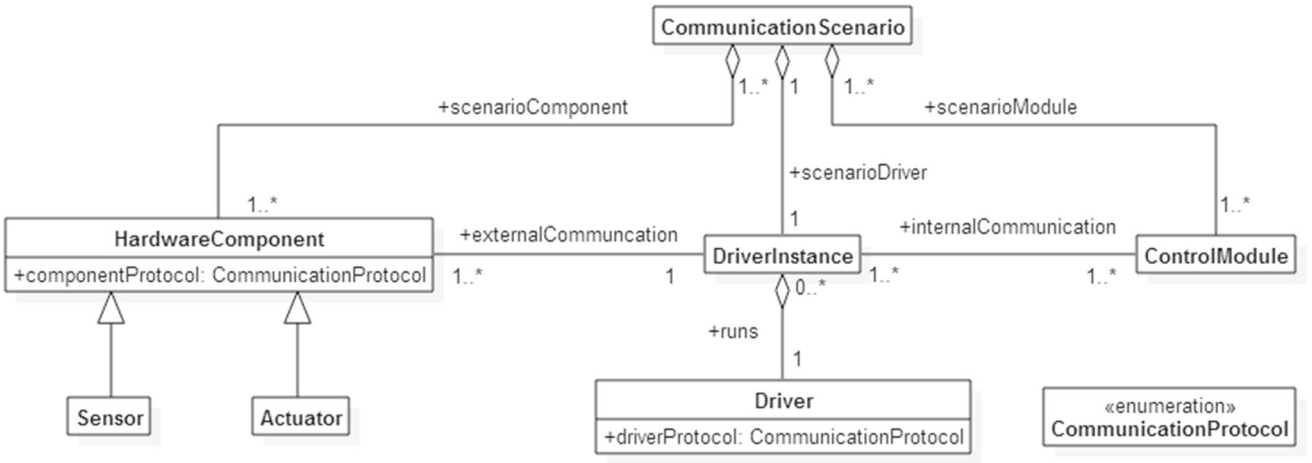

Fig. 2 A class diagram representing the key entities in the fire and gas monitoring system (FMS)

are stated in the FMS Requirements Specification Document.

Three important context factors in the FMS case study influence the definition of our approach to generate system configurations and stress test cases.

1. Different instances of a given driver are independent, in the sense that they do not communicate with one another and do not share memory.
2. The purpose of the constraint on CPU usage is to enable engineers to estimate the number of driver instances of a given monitoring application that can be deployed on a CPU. These constraints express bounds on the amount of CPU time required by one driver instance. Therefore, in this paper we focus on individual driver instances. The independence of the drivers (first factor above) is the key for being able to localize CPU usage analysis to individual instances in a sound manner.
3. The drivers are not *memory-bound*, i.e., task deadlines, response time, and CPU usage, are not significantly affected by activities such as disk I/O and garbage collection. To ensure this, KM engineers over-approximates the maximum memory required for each driver instance by multiplying the number of hardware devices connected to the driver instance and the maximum size of data sent by each device. Execution profiles indicate that the drivers are extremely unlikely to exceed this limit during their lifetime.

Drivers in the FMS share the same design pattern, where periodic and aperiodic tasks communicate asynchronously through buffers. There exist two major types of driver implementations, one with four aperiodic tasks (Sect. 2.1), and another with a *singular task* consisting of four *activities* in an infinite loop (Sect. 2.2).[3] However, regardless of the implementation, all the drivers have to satisfy the same performance requirements. Nonetheless, the real-time properties determining whether or not the performance requirements are satisfied at runtime closely depend on the implementation used.

2.1 Implementation 1: Data transfer with one singular task and four activities

The first implementation of the FMS drivers consists of three tasks communicating through three buffers. More precisely, in this implementation a generic driver consists of:

- Three communication buffers, namely *BoxIn*, *Queue*, and *BoxOut*. These buffers serve as temporary storage locations for the data transiting from the hardware devices to the control modules. Moreover, the buffers have a fixed capacity, and are accessed by software tasks with mutual exclusion, i.e., no two task can simultaneously access a buffer. *BoxIn* and *BoxOut* contain formatted data coming from the external hardware and going toward the control modules, respectively. *Queue* contains a priority-ordered list of commands extracted from incoming data, that have to be forwarded to the control modules.
- Two periodic tasks, namely *PullData* and *PushData*. These tasks are periodically activated by a *scan* signal, and transfer data from the hardware sensors, and to the control modules respectively.
- One singular task, namely *IODispatch*, enclosing four activities in an infinite loop. The activities read from

[3] Note that the scheduling theory defines as singular a task which is executed only once during the system execution, and an activity as a sequence of operations that a task executes.

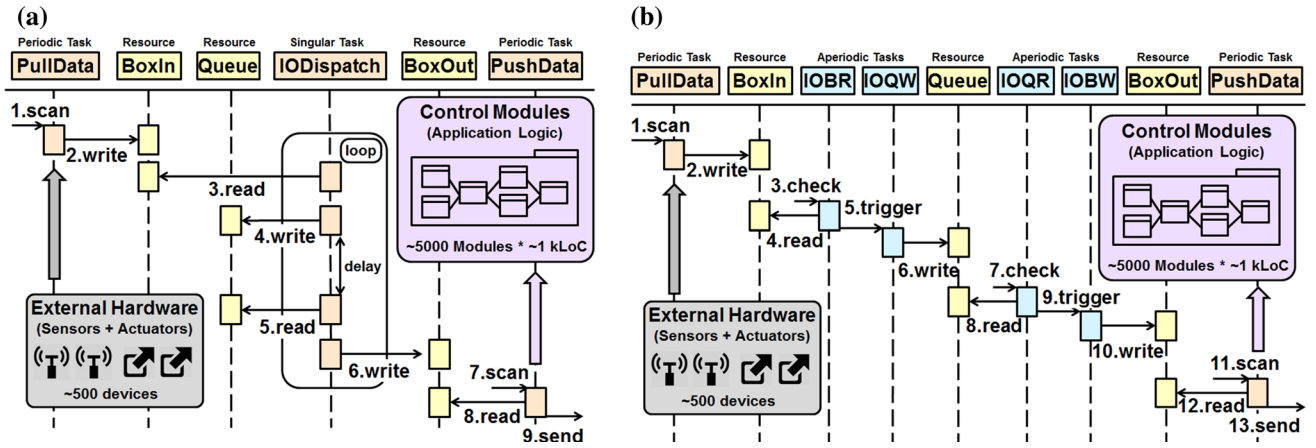

Fig. 3 Two implementations of the typical operating scenario of drivers in the fire and gas monitoring system (FMS), consisting of a unidirectional data transfer between external hardware sensors and control modules. **a** Implementation 1 consisting of two periodic tasks, an aperiodic task enclosed in an infinite loop, and three buffers. **b** Implementation 2 consisting of two periodic tasks, four aperiodic tasks, and three buffers

BoxIn, temporarily store data in the priority *Queue*, and finally write it in *BoxOut*.

The activities that read *BoxIn* and write *Queue* are separated by the activities that read *Queue* and write *BoxOut* by a *delay* time. The delay typically corresponds to a *sleep* call in the drivers source code, which ensures that the control modules receive data from the sensors at a slow enough rate so that the FMS can process it. Note that *IODispatch* is executed only once, in particular when the system starts, and its behavior is determined by the delay time between activities at each loop iteration.

Figure 3a shows how tasks in the first driver implementation collaborate in the typical scenario, that is a unidirectional data transfer between hardware sensors and control modules. (1) *PullData* periodically receives data from sensors or human operators, formats the data in an appropriate command form, and (2) writes it in the buffer *BoxIn*. (3) *IODispatch* reads the data from the buffer, extracts the commands from the data, and (4) stores them in the priority *Queue*. After a given delay time, (5) *IODispatch* reads the highest priority command and (6) writes it to *BoxOut*. When the periodic *scan* signal (7) activates *PushData*, the task (8) reads the commands from *BoxOut* and finally (9) sends them to the control modules for processing. As mentioned above, this asynchronous design is necessary for drivers to smooth the data transfer between external devices and control devices. However, the data transfer functionality is subject to strict performance requirements in terms of task deadlines, response time, and CPU Usage. Specifically, (1) each of the six tasks that implement a driver has to finish before its deadline, typically in the range of milliseconds. Consider for instance a scenario where *PushData* is blocked on *BoxOut* by *IODispatch*, and thus is late in alerting the control modules that a fire has been detected. In this case, the system will fail to timely activate the alarm and the sprinklers, with potential severe consequences. These task deadlines are hard real-time constraints that have to be met to ensure that the system safely reacts in case of fire. However, the FMS is also subject to soft-real-time constraints such as (2) response times. Indeed, the interval of time between an execution of *PullData* and the execution of *PushData* that sends the commands to the control modules has to be bounded. Consider for instance a scenario in which some data in *BoxIn* is not promptly emptied. If too much time passes after that data is collected by the external hardware, the new data coming from the same sensor will arrive when the old data has still not been processed. Therefore, the FMS will perform the commands corresponding to the first chunk of data when the environment state has already changed. This behavior prevents the system from reacting promptly to external changes. Finally, driver instances are independent from each other, and thus concurrently executed in the same hardware platform. For this reason, (3) each driver instance must not exceed a given threshold of CPU usage. For example, if a fire and gas monitor is starved of CPU time due to computational overload, it can have a delayed or miss response to a fire or gas leak with potentially serious consequence. The main variables determining whether or not these requirements are satisfied at runtime are the delay times between the first and the last two activities of *IODispatch*. Indeed, if the delay times are too short, *IODispatch* is continuously running and keeps the CPU busy, eventually exceeding the given threshold on CPU usage. On the other hand, if the delay times are too large, *pullData* may fill up *BoxIn*, and be blocked waiting for *IODispatch* to empty the buffer. As a result, *pullData* is

not able to terminate before its next *scan* signal arrives, missing its deadline. These scenarios can arise because the delay times determine the arrival time of the activities in *IODispatch*, which in turn can preempt or be preempted by other tasks. However, the delay times of the *IODispatch* iterations are tunable parameters that engineers can set when configuring the drivers. Therefore, in order to generate configurations that satisfy task deadlines, response time, and CPU usage in the I/O drivers, we need a strategy to search for all the possible delay times between activities. In particular, the objective of the search is finding scenarios that are predicted to satisfy the requirements above, possibly exhibiting minimal completion time for tasks, response time, and CPU usage.

2.2 Implementation 2: Data transfer with four aperiodic tasks

There also exists a second implementation of the FMS drivers, where the functionality of *IODispatch* is realized by two aperiodic tasks, namely *IOBoxRead* ((*IOBR*) and *IOQueueRead* ((*IOQR*), and two triggered tasks, namely *IOBoxWrite* ((*IOBW*) and *IOQueueWrite* ((*IOQW*). In particular, *IOBoxRead* and *IOQueueRead* are activated by the *check* signal fired by the RTOS when *BoxIn* and *Queue* are full and need to be emptied. Furthermore, *IOBoxWrite* and *IOQueueWrite* are activated by a *trigger* signal from *IOBoxRead* and *IOQueueRead* respectively, when they finish reading from the *BoxIn* and *Queue* buffers.

Figure 3b shows how tasks in the second driver implementation collaborate in the scenario of a unidirectional data transfer between hardware sensors and control modules. (1)*PullData* periodically receives data from sensors or human operators, formats the data in an appropriate command form, and (2) writes it in the buffer *BoxIn*. (3) When *BoxIn* is almost full, the *check* signal activates *IOBoxRead* that (4) reads the data from the buffer and (5) triggers *IOQueueWrite*. *IOQueueWrite* extracts the commands from the data, and (6) stores them in the priority *Queue*. When *Queue* reaches a critical capacity, (7) the *check* signal activates *IOQueueRead* that (8) reads the highest priority commands and (9) triggers *IOBoxWrite* which in turn (10) writes the commands to *BoxOut*. When the periodic *scan* signal (11) activates *PushData*, the task (12) reads the commands from *BoxOut* and finally (13) forwards them to the control modules for processing.

There exists a fundamental difference between the first drivers implementation described in Sect. 2.1, and this second one. In the former, the main variables determining whether or not the driver performance requirements are satisfied at runtime are the delay times that separate the second and third activities of *IODispatch* at each loop iteration. In this second implementation, the performance requirements of the drivers mostly depend on the arrival times of the *check* signal, which are not tunable parameters, but rather depend on the state of the three buffers. In turn, the buffers state depends on data sent by the hardware sensors via *PullData*, which is determined by unpredictable environmental conditions. The arrival times also vary across different system executions, as a consequence of the impossibility to predict the data coming from the sensors. Therefore, in order to generate stress test cases likely to violate requirements on task deadlines, response time, and CPU usage, we need a strategy to search for all the possible task arrival times. In particular, the objective of the search is finding scenarios that are predicted to violate the requirements above, or be as close as possible to doing so.

3 Related work

Traditionally, real-time embedded systems have been approached with scheduling analysis methodologies based on real-time theory [31], which assumes restrictive conditions on the target system and execution platform [32]. On the other hand, recent work stated the benefits of model-based approaches where software performance analysis is carried out starting from early development stages [10]. For this reason, these approaches traditionally belong to the field of model-driven engineering (MDE). The idea behind these approaches is to analyze the schedulability of RTESs in a system model that captures the properties of real-time tasks, such as periods, WCET, priorities, and dependencies. This provides the flexibility to incorporate specific domain assumptions and to analyze a range of possible scenarios, including the worst cases [33]. Indeed, opposite to theorems from the real-time scheduling theory, model-based approaches can better adapt to large and complex RTESs where interdependent aperiodic tasks run on multi-core processors.

In general, model-based approaches for performance and scheduling analysis are based on the explicit modeling of time and concurrency aspects of the target RTES [34]. Examples of such approaches include queuing networks [35], stochastic Petri nets [36] and automata networks [37]. Recently, there has been a growing interest in developing standardized languages to enhance the adoption of performance analysis concepts and techniques in the industry [38]. The most notable these languages is the UML profile for Modeling and Analysis of Real-Time Embedded Systems (UML/MARTE or MARTE), that extends UML with modeling abstractions supporting the definition of quantitative analysis methodologies for RTESs. However, UML/MARTE is a large profile that accounts for a variety of aspects in quantitative analysis of RTESs, and does not include guidelines on what abstractions are needed for a particular analysis [13].

For this reason, in this paper, we identify a subset of UML/MARTE required for performance tuning and stress testing, along with guidelines on how to apply the entities in such subset. In particular, we provide a conceptual model capturing the timing and concurrency abstractions needed for the generation of tunable performance-related parameters and stress test cases. Then, we provide a mapping from this conceptual model to a pruned version of UML/MARTE. We obtain the pruned metamodel using a pruning algorithm [22] that ensures type conformance [23] of the pruned metamodel with UML/MARTE. In particular, devising a conceptual model to tailor UML profiles for a specific methodology has been successfully used in the past, especially in the field of performance engineering and testing. Examples include methodologies for deadlocks detection based on the predecessor of UML/MARTE, the UML profile for Schedulability, Performance, and Time (UML/SPT or SPT) [39], and for early scheduling analysis to design RTESs in such a way that they comply to their timing constraints [40]. However, to the best of our knowledge, we are not aware of the use in such methodologies of a pruned UML/MARTE metamodel.

In the field of model-driven software verification, model checking (MC) has been successfully used to verify performance properties expressed in a model [41]. In these approaches, properties typically represent conditions that should never hold in the system at any given time. These properties are formulated as reachability queries of a faulty state in a finite state machine (FSM), and model checkers verify if there exists a path from the initial state of the FSM to this faulty state. MC is mostly used in software verification to compare a model with its specification, e.g., to check the absence of deadline misses in a FSM modeling task executions. In particular, real-time model checkers, e.g., UPPAAL [42], are commonly used for the evaluation of time-related properties. We identify three main differences between our work and MC approaches used in the context of performance analysis and testing. First, MC approaches are mostly used for *verification*, i.e., to check if a given set of real-time tasks satisfy some property of interest. Even though our approach can also be used for a loose design-time verification, the focus of this article is on performance tuning and stress testing. For this reason, our approach is complementary, and not alternative, to MC approaches. Second, software testing approaches that use MC for test case generation cast the performance property to be checked as a *boolean* reachability property over a FSM, in a way that a particular scenario either violates the property or does not. On the other hand, the search approaches used in this article for performance and stress test cases generation are based upon the optimization of a *quantitative* objective function that expresses the extent to which a given scenario violates the performance requirement. Third, to adapt model checkers for checking different properties of real-time applications, the target system FSM has to be modeled in such a way that the target property can be formulated as a reachability query. For example, consider the problem of verifying whether the CPU usage of a system exceeds a given threshold. This problem is solved by augmenting the system FSM with an *idle* state that keeps track of the CPU time used by the tasks [33]. In this way, the error states are the ones in the FSM that can be reached only when the CPU usage threshold is violated. On the other hand, the approach in this article is based upon the optimization of objective functions representing the performance requirements to be satisfied by the parameters, and violated by the test cases. This means that, to adapt our approach to generate values for tunable performance-related parameters and stress test cases concerning performance requirements other than those we consider, one only needs to formulate new objective functions.

Note that, when MC approaches verify that a given property does not hold in a model, they also provide counterexamples similar to the scenarios that are the focus of this article. However, MC faces limitations when it comes to generating best- or worst-case scenarios with respect to time-related properties such as task deadlines, response time and CPU usage. (1) Model checking requires complex formal modeling of the system, which often leads to the well-known state explosion problem that has not been solved in the general case [15]. (2) For stress-testing purposes, engineers are also interested in deadline near-misses, i.e., those scenarios where tasks are predicted to be close to missing a deadline. Indeed, since model checking approaches are based on estimates for the task execution times, even such scenarios have to be tested because they can lead to deadline misses during execution. (3) Model checkers usually do not provide a usable result prior to termination. However, for practical use, performance tuning and stress testing have to be performed within a time budget. Therefore, to be effective, the generation of scenarios has to produce an usable output within the time budget, which is not the case if MC does not terminate soon enough. To the best of our knowledge, we are not aware of model checking approaches targeted at verifying task deadlines, response time, and CPU usage properties that overcome these three issues. This aspect motivated us in investigating alternatives to generate best- or worst-case scenarios that characterize configurations and stress test cases, such as search strategies based on constraint programming.

4 Approach overview

The approach presented in this paper builds upon our previous work [27] for deriving test cases exercising the CPU usage requirements of RTESs running on multi-core platforms. Specifically, the approach we extended the approach to both (1) derive configurations characterized by task delay

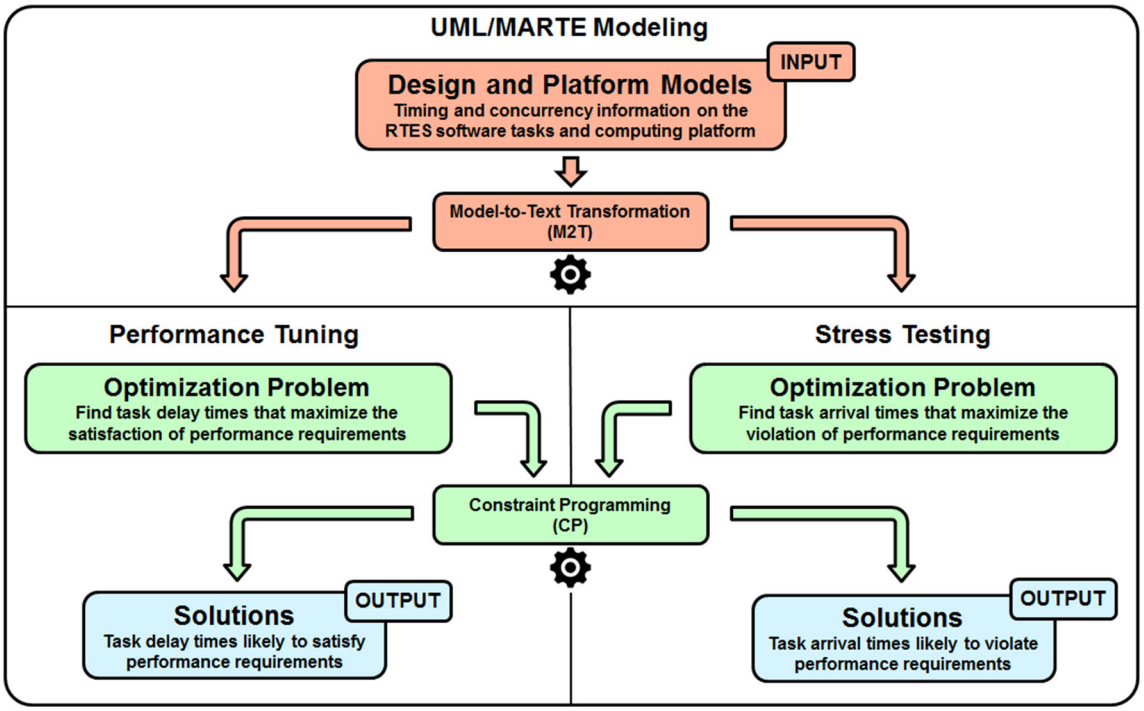

Fig. 4 Our UML/MARTE approach to support performance tuning and stress testing in RTES

times that maximize the *satisfaction* of requirements on task deadlines, response time, and CPU usage, and (2) derive stress test cases characterized by task arrival times that maximize the *violation* of requirements on task deadlines, response time, and CPU usage. The approach combines UML/MARTE modeling, to capture the timing and concurrency aspects of the system design and platform, and automated search based on constraint programming, to generate configurations and stress test cases. An overview of the approach is shown in Fig. 4, where we highlight with a gear icon the automated steps. In the following, Sect. 4.1 introduces the UML/MARTE modeling part of the approach, which is detailed in Sect. 5. Section 4.2 describes the automated search to generate configurations in terms of tunable parameters and stress test cases, which is part of previous work [20,21].

4.1 Modeling timing and concurrency abstractions in UML/MARTE

The approach we propose builds upon a conceptual model that captures abstractions of the RTESs timing and concurrency aspects, which enable the performance analysis for the generation of configurations and stress test cases. In particular, entities in the conceptual model capture abstractions of the software application, e.g., tasks with their priorities, periods, dependencies, and abstractions of the computing platform, e.g., processing cores and scheduling policies. To simplify the application of the conceptual model in model-driven engineering approaches, the entities are mapped to UML/MARTE stereotypes and stereotype properties that apply to class, sequence, and deployment diagrams. In this way, software design and platform models stereotyped with UML/MARTE effectively organize the input data for our approach in the UML standard. In order to further simplify the process, we use a metamodel pruning technique on the UML/MARTE metamodel [22]. This pruning significantly reduces the number of concepts, leaving in the metamodel only those which are relevant for effectively using UML/MARTE to support performance tuning and stress testing. In particular, pruning UML/MARTE allows to specify a model-to-text transformation from the pruned UML/MARTE metamodel to OPL data files, which specify the input data for the automated search of configurations and stress test cases.

4.2 Generating system configurations and stress test cases with constrained optimization

Starting from design and platform models stereotyped in UML/MARTE, our approach uses automated search to generate configurations characterized by delay times, and stress test cases characterized by task arrival times. Specifically, we cast the generation of configurations and stress test cases as optimization problems over the abstractions characterizing design and platform models. The goal of these optimization problems is to find task delay and arrival times that minimize

and maximize the satisfaction of performance requirements on task deadlines, response time and CPU usage. The optimization problems are solved with a search strategy based on constraint programming, which is inspired by work in the area of constraint-based scheduling [43]. These problems model the system design, real-time properties, executing platform, and performance requirements. Specifically, we cast the search for delay times that characterize best-case schedules and for task arrival times that characterize worst-case schedules, as the constraint optimization problem (COPs) $\mathcal{M}$ and $\mathcal{M}'$, respectively [20,21].

Casting these problems as COPs is subject to the following assumptions.

1. The RTOS scheduler checks the running tasks for potential preemptions at regular and fixed intervals of time, referred to as *time quanta*. Therefore, each time value in our problem is expressed as a multiple of a time quantum. Accordingly to the specification of the RTOS executing the FMS, we consider the length of ten milliseconds for time quanta.
2. The interval of time in which the scheduler switches context between tasks is negligible compared to a time quantum, and hence negligible with respect to the tasks execution time.
3. The RTOS overhead for managing tasks in negligible with respect to their execution and interarrival times. This assumption allows us to consider, for scheduling purposes, a task $j = [a_1, a_2, \ldots a_n]$ with priority p consisting of n activities a_i as a sequence $[j_1, j_2, \ldots j_n]$ of n tasks with priority p, where the duration of j_i is equal to the duration of a_i, and where j_i triggers j_{i+1}. In this case, each task j_i inherits the dependencies and triggering relationships of the corresponding activity a_i [20].

We found these three assumptions to be commonplace in the context of RTESs, as the scheduling rate of operating systems varies in the range of few milliseconds, while the time needed for context switching is usually in the order of nanoseconds [44]. These assumptions allow us to consider time as discrete in our analysis, and model the COPs as Integer Programs (IPs) over finite domains.

$\mathcal{M}$ aims at *minimizing* the objective functions, and performs a multi-objective search. This is because the system configurations should achieve an optimal trade-off between conflicting requirements such as response times and CPU usage, where ideally all the requirements are far from being violated. On the other hand, $\mathcal{M}'$ aims at *maximizing* objective functions expressed in terms of task deadlines, response time, and CPU usage. The model is used to perform repeated single-objective searches, each time with a different objective. This is because even a single violation on either task deadlines, response time, and CPU usage compromises the system safety. Even though $\mathcal{M}$ and $\mathcal{M}'$ are substantially different in scope, they syntactically differ only in the definition of delay times as variables, in the number of objective criteria, and in the optimization goal of the search (maximizing/minimizing). This fact shows that casting scheduling analysis of RTESs as a COP is a flexible approach that can be easily tailored to support activities in different phases of software development, such as stress testing and performance tuning. Note that, being these two different and independent activities, $\mathcal{M}$ and $\mathcal{M}'$ are solved separately.

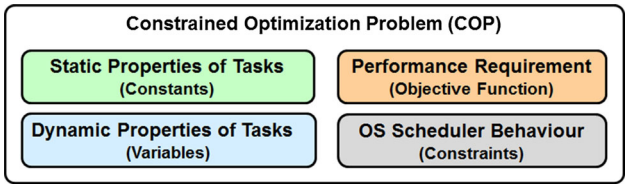

Fig. 5 Architectural overview of the constraint optimization models $\mathcal{M}$ and $\mathcal{M}'$

The models are specified in OPL, and are solved with the IMB ILOG CPLEX CP Optimizer,[4] one of the leading CP solvers available in the market. Figures 5 and 6 show the architecture of $\mathcal{M}$ and $\mathcal{M}'$ and an excerpt of their OPL implementation, respectively. In particular, the COPs architecture reflects the key idea behind our formulation, which relies on the following four main points.

1. We model the system design, which is static and known prior to the analysis, as a set of constants (lines 1–26 in Fig. 6). The system design consists of the tasks of the real-time application, their dependencies, offsets, periods, Worst-Case Execution Times (WCETs), deadlines, and priorities. In particular, dependencies relations between tasks are defined symmetric, i.e., if a task j_1 depends on another task j_2, then also j_2 depends on j_1. Moreover, RTESs task execution times depend on the system inputs, and can be estimated through WCET analysis techniques. Note that, in the literature, a task WCET is defined as the maximum time that the task executes, and hence keeps the CPU busy [45]. This definition does not take into account the time that the task has been preempted by other tasks or the time the task is blocked waiting for computational resources, because in these cases the task in not using the CPU.
2. We model the system properties that depend on runtime behavior, and those that are configurable parameters, as a set of variables (lines 29–44 in Fig. 6). This is because, in general, it is hard to predict how external inputs and timing parameters affect the tasks execution, and hence an appropriate search strategy is needed to identify such

[4] http://www.ibm.com/software/.

```
/* I. Constants */

// T: Observation Interval (range of time quanta)
int tq = ...;
range T = 0..tq-1;

// c: Number of Processor Cores
int c = ...;

// Task: Definition of task and task properties
tuple Task {
    key string id; int priority; int deadline;
    int period; int min_interarrival_time;
    int max_interarrival_time; int min_delay_time;
    int max_delay_time; int duration; }

// Depends: Definition of Task Dependencies
tuple Depends { Task j1; Task j2; }

// Triggers: Definition of Task Triggerings
tuple Triggers { Task j1; Task j2; }

// J,DS,TS: Task Set, Dependencies, and Triggerings
{Task} J = ...;
{Depends} DS with j1 in J, j2 in J = ...;
{Triggers} TS with j1 in J, j2 in J = ...;


/* II.a Independent Variables */
dvar int task_executions[J] in X[j];
dvar int delay_time[J, K[j]] in D[j];
dvar int arrival_time[J, K[j]] in T;
dvar int active[J, K[j], P[j]] in T;


/* II.b Dependent Variables */
dexpr int start[j in J, k in K[j]] =
    active[<j,k,0>];
dexpr int end[j in J, k in K[j]] =
    active[<j,k,j.duration-1>] + 1;
dexpr int execution_deadline[j in j, k in K[j]] =
    arrival_time[j, k] + j.deadline;
dexpr int deadline_miss[j in J, k in K[j]] =
    end[j, k] - execution_deadline[j, k];


/* III.a Well-formedness Constraints */
forall(j in J, k in K[j], p in P[j])
    wfc1: end[j, k] >= start[j, k] + j.duration;
    wfc2: active[j, k, p] <= active[j, k, p-1] - 1

/* III.b Temporal Ordering Constraints */
forall(ts in TS, k in K[ts.j1])
    toc1: arrival_time[ts.j2, k] = end[ts.j1, k];


/* IV. Objective Functions */

// f_dm: Deadline Misses Function
dexpr float f_dm =
    sum(j in J, k in K[j]) 2^deadline_miss[j, k];

// f_rt: Response Time Function
dexpr int f_rt =
    max(j in J, k in K[j])  end[j, k] -
    min(j in J, k in K[j]) arrival_time[j, k];

// f_cu: CPU Usage Function
dexpr float f_cu =
    (sum(t in T) (load[t] > 0)) / tq;


/* V. Optimization directives (alternatives) */

// M: Performance Tuning Model
minimize staticLex(f_dm, f_rt, f_cu);

// M': Stress Testing Model
maximize f_dm; // maximize f_rt; // maximize f_cu;
```

Fig. 6 Excerpt of the OPL implementation of $\mathcal{M}$ and $\mathcal{M}'$

inputs and parameter values [28, XXXX]. The main real-time properties depending on runtime behavior are the number of task executions, the arrival times of aperiodic tasks, and the specific runtime schedule of the tasks. The configurable properties we consider in our constraint model are instead the delay times between task activities. The real-time properties in these two categories represent the main output variables of the constraint models, and are used to support stress testing and performance tuning.

3. We model the RTOS scheduler as a set of constraints among such constants and variables (lines 47–54 in Fig. 6). Indeed, the RTOS scheduler periodically checks for triggering signals of tasks and determines whether tasks are ready to be executed or need to be preempted. In practice, constraints specify mathematical relations between variables and constants, and are divided into five major subsets:
   (a) *Well-formedness constraints (WFC)* specifying relations among variables directly following from their definition (lines 47–50 in Fig. 6).
   (b) *Temporal ordering constraints (TOC)* capturing the dependency and triggering relationships between tasks (lines 52–54 in Fig. 6).
   (c) *Multi-core constraints (MC)* capturing the specification of the number of cores of the computing platform.
   (d) *Preemptive scheduling constraints (PSC)* stating that each task should be preempted when a higher priority task is ready to be executed and no cores are available.
   (e) *Scheduling efficiency constraints (SEC)* ensuring that the scheduler avoids unnecessary context switching, and executes tasks as soon as enough resources are available.
4. We model the performance requirements to be satisfied during performance tuning, or violated during stress testing, i.e., task deadline misses, response time, or CPU usage, as objective functions (lines 57–70 in Fig. 6). These functions are expressed in terms of the constants and variables of the constraint model, and drive the search toward best-case task delay times in the case of performance tuning (lines 75–76 in Fig. 6), or worst-case task arrival times in the case of stress testing (lines 78–79 in Fig. 6). Note that these functions are minimized, in case of performance tuning (lines 75–76 in Fig. 6), and maximized, in case of stress testing (lines 78–79 in Fig. 6) in separate executions of $\mathcal{M}$ and $\mathcal{M}'$, respectively.

5 Supporting performance tuning and stress testing with UML/MARTE modeling

Even though UML has been used for long time to model real-time embedded systems [46], it is not flexible enough

to be effectively applied in large and complex applications. However, UML defines an extension mechanism in the form of *profiles*, which tailor the language to a specific domain by providing modeling concepts that characterize that domain. In the context of RTESs, the UML profile for Modeling and Analysis of Real-time Embedded Systems (UML/MARTE [47]) is the most acknowledged and used by practitioners. UML/MARTE is an OMG standard released in 2011 to replace its predecessor, the UML profile for schedulability, Performance, and Time (UML/SPT) aligned with UML v1.x. UML/MARTE defines a large number of abstractions which support the definition of methodologies for performance analysis and verification. Except for few examples, the profile specification does not detail how to identify the relevant stereotypes and stereotype properties for a given kind of analysis. This renders effectively applying UML/MARTE in large and complex industrial systems a challenging task, which needs to be supported by the definition of modeling guidelines [13].

In this section, we propose a conceptual model that captures, independently from any modeling language, the abstractions required to support performance tuning and stress testing in RTESs (Sect. 5.1). To simplify the application of our conceptual model in model-driven engineering (MDE) approaches, we propose a mapping of our conceptual model to UML/MARTE (Sect. 5.2). Note that we define the abstractions needed to support stress test cases in two steps, i.e., first defining a conceptual model, and then its mapping to UML/MARTE. This formalization approach is similar to that used in UML/MARTE, where concepts related to RTES are first defined in a domain model, and then formalized as stereotypes and stereotype properties. After defining the mapping to UML/MARTE, we apply model-pruning techniques (Sect. 5.3) in order to obtain a simplified and more manageable metamodel, which is a supertype of UML/MARTE and hence is still aligned to model-driven engineering standards. Starting from this pruned metamodel, we define a mapping between the entities in UML/MARTE required for our analysis and the Optimization Programming Language (OPL) (Sect. 5.4), which encapsulates the data needed to automate the search for system configuration and stress test cases.

5.1 A conceptual model for performance tuning and stress testing

Recall from Sect. 2 that the goal of our approach is finding scenarios for RTESs tasks as likely as possible to satisfy/violate task deadlines, response time, and CPU usage constraints. Therefore, our conceptual model is based upon abstractions defined in the real-time scheduling theory, such as tasks, activities, and scheduling policies [48]. Figure 7 shows an overview of the conceptual model we propose, whose entities are explained below. Classes in the conceptual model are partitioned into the *Application* and *Platform* packages.

- *Application* The software part of a RTES is an embedded application, which consists of several parallel software *tasks*, and is allocated to a *computing platform.*
  - *Activity* Recall from Sect. 2 that the scheduling theory defines an activity as a sequence of operations in a task, and that task activities within a task are sequentially executed. Each activity a has an estimated *duration* or worst-case execution time (WCET), and starts executing after a given *release* time. Consecutive activities within a task are separated by a delay time, whose bounds are specified by minimum and maximum values (*minDelay* and *maxDelay*). Activities can *trigger* other activities: For example, each activity in a task triggers the following one, thus defining a temporal ordering. This is because after the last statement in an activity is executed, the program control flow executes the first statement in the following activity. Activities can also trigger other tasks: for example, upon meeting certain conditions, an activity can spawn a new task to perform additional operations. Moreover, activities in a task can also *depend* on each other by sharing computational resources which are generally used for communication.
  - *DataDependency* Activities can depend on each other because they communicate in a *synchronous* or *asynchronous* way.
  - *Buffer* Asynchronous communication between two activities can use a buffer, whose *access* is protected by semaphores, and hence is blocking. This means that at most one activity can access a buffer at any time. Note that, in this article, we only consider the case where tasks are communicating asynchronously through buffers.
  - *Task* RTES software consists of a set of parallel tasks that have to complete before a given *deadline*. Each task also has a *priority* that determines the relative importance of a task with respect to other tasks, so that the scheduler executes higher priority tasks before lower priority tasks. *Periodic* tasks are triggered by timed events handled by the global clock, and are invoked at regular intervals. Therefore, their arrival times are fixed, and equal to multiples of one interval, called *period*, which is counted starting from an *offset*. Periodic tasks are commonly used to send and receive data at regular interval of times, e.g., *PushData* and *PullData* in Fig. 3a, b. On the other hand, the arrival times for *aperiodic*

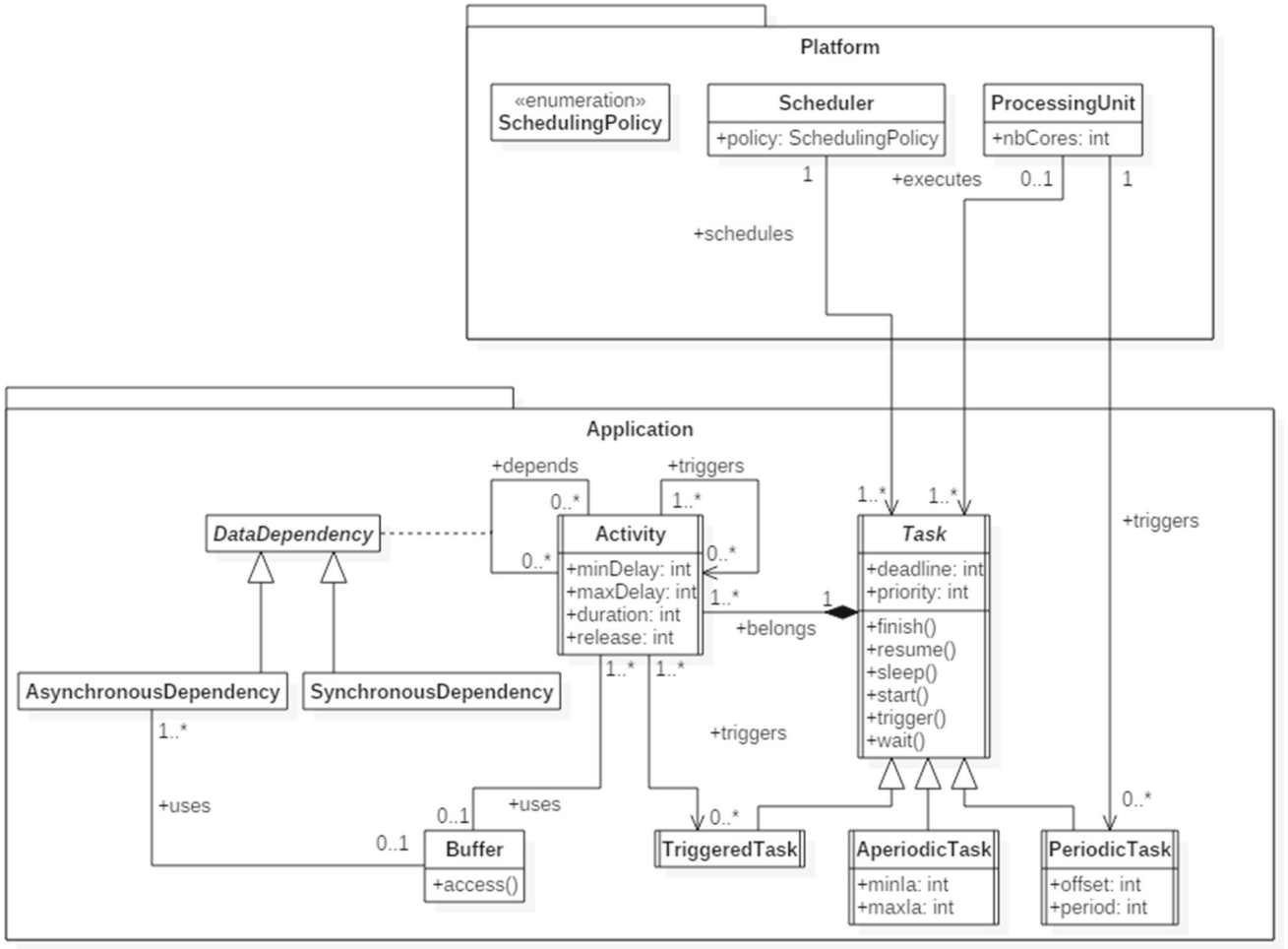

Fig. 7 Conceptual model representing the key abstractions to support performance tuning and stress testing in real-time embedded systems

tasks are bound by minimum and maximum inter-arrival times (*minIa* and *maxIa*), which indicate the minimum and maximum time intervals between two consecutive arrivals of the event triggering the task. Aperiodic tasks are instead used to process asynchronous events/communications, e.g., *IODispatch* in Fig. 3b. Finally, the arrival time of *triggered* tasks is determined by particular activities, which launch the task upon finishing their execution. During its lifetime, a task can perform the following operations.

- *Trigger* Communicates to the scheduler that the task is ready to start a new execution in response to a triggering event. The origin of the event depends on the type of the task. For periodic tasks, the event comes from an internal clock, for aperiodic tasks the event comes from the external environment or the RTOS, and for triggered tasks the event comes from another task.
- *Start* Begins execution after having been assigned to a CPU core by the scheduler. This operation precedes the execution of the first activity of each task.
- *Finish* Completes execution. This operation is performed after the last activity in a task has completed.
- *Wait* Temporarily stops execution in order to synchronize with another task, or to acquire a resource. Note that each buffer access within an activity implies an implicit *wait* by its task.
- *Sleep* Temporarily stops execution for a given amount of time. Note that a *sleep* call for a given time t at the beginning (end) of an activity corresponds to a release (delay) for that activity equal to t.
- *Resume* Communicates to the scheduler that the task is ready to resume execution after a previous *wait* or *sleep* operation.

The state machine in Fig. 8 shows the lifecycle of a task, where the operations determine transitions between states. Tasks start in the *idle* state, and only

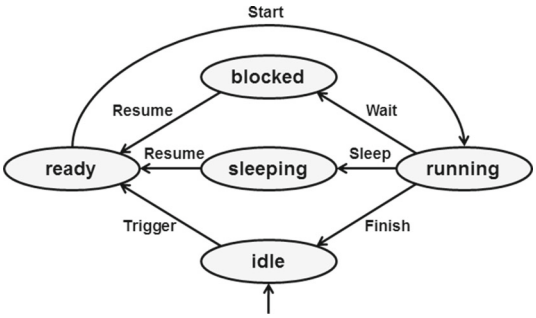

Fig. 8 The state machine representing a task lifecycle. Events triggering a task determine a transition from the *idle* state to the *ready* state

consume CPU time in the *running* state. Indeed, in our model, activities within tasks release the CPU when preempted by the RTOS scheduler, so that the CPU can be used by another activity belonging to a higher priority task. Note that, as it is common in embedded systems intended to run continuously, there is no final state.

- *Platform* A computing platform consists of the hardware and lower-level software parts of a RTES, i.e., a *processing unit*, and a real-time *scheduler*.
  - *ProcessingUnit* A processing unit represents the CPU of the computing platform. Each CPU has a number of processor cores (*nbCores*), specifying the maximum number of tasks that can be executed in parallel. As explained in Sect. 2, we do not consider RAM, disk memory, or cache in our conceptual model. This is because, given the context factors of this article, the impact memory has over task deadlines, response time, and CPU usage is negligible.
  - *Scheduler* The scheduler implements a given scheduling policy (*SchedulingPolicy*), which defines the rules to handle concurrency and execution order among tasks. Even though several policies are commonly used in RTESs, in this article we only consider the fixed-priority preemptive scheduling policy used by the FMS.

Note that, in our conceptual model, both tasks and activities are *active objects*, and hence are represented with double bars on the sides.[5]

5.2 Mapping the conceptual model to UML/MARTE

To enable effective industrial use, every approach in software engineering has to be capable of seamless integration in the companies development cycle. In the last years, model-driven engineering (MDE) has risen as a way to handle software complexity through the systematic use of models during development [49]. In the context of RTESs, reasoning about performance requirements such as deadline misses, response time, and CPU usage requires the explicit modeling of time, which is one of the key characteristics of UML/MARTE. For this reason, we provide a mapping between the abstractions in our conceptual model and UML/MARTE stereotypes and stereotype properties. This mapping shows the feasibility of extracting the abstractions required to support the generation of system configurations and stress test cases from a standard modeling language, such as UML.

5.2.1 Mapping conceptual entities to UML

Some of the abstractions in our conceptual model are already defined in UML. In particular, each active object in a sequence diagram can be associated to a *lifeline*. In this way, *activity* and *behavior execution specifications* represent, depending on the level of abstraction, the activities in our conceptual model. Similarly, *occurrence specifications* represent sending and receiving of messages, and therefore can be used to describe the synchronous and asynchronous communication defined in our conceptual model. Figure 9 shows a sequence diagram capturing the data transfer scenario of the fire and gas monitoring system described in Sect. 2.1. The driver tasks are active objects, while the buffers are passive objects. Each activity within a task is depicted using an execution specification, i.e., as a box on the task lifeline that shows the interval of time that the task performs the activity. Therefore, *pullData* has two activities, *ioDispatch* has four, and *pushData* has three. Note that the order of activations on a task lifeline implies the temporal ordering between activities of that task. In sequence diagrams, a synchronous message between two activities is shown using an arrow with a filled head, while an asynchronous message is shown by an arrow with an open head. Synchronous communication is blocking and does not necessarily require a buffer, because the sending activity must wait until the receiving activity is ready to receive the message. On the other hand, asynchronous communication typically uses buffers. Recall from Sect. 2 that in the FMS, and hence in Fig. 9, all communications are asynchronous and use buffers. Figure 10 shows a sequence diagram capturing the FMS data transfer scenario described in Sect. 2.2. In this case, *ioDispatch* is replaced by the four tasks *ioBoxRead*, *ioQueueWrite*, *ioQueueRead*, and *ioBoxWrite*. Note that the triggering relations between tasks are modeled by *create* messages, i.e., by UML *Message* objects whose *messageSort* property has value *create*. This is because each triggered task is instantiated and launched by its triggering task. Note that, to reduce the total number of objects in the diagrams, only few representative stereotype properties are shown.

[5] In UML, active objects model entities owning a process or thread, and that can initiate flow control activity.

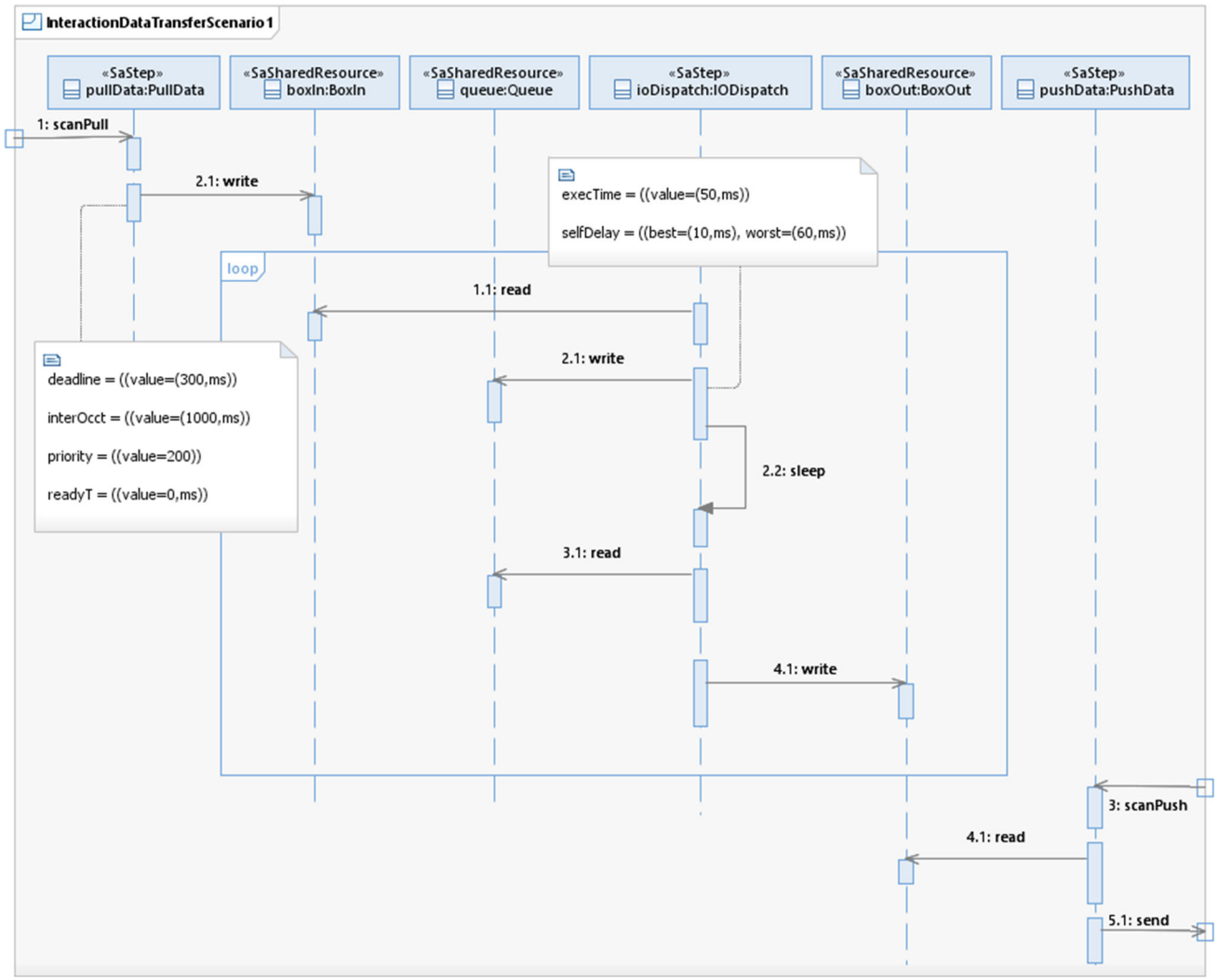

Fig. 9 Sequence diagram modeling the data transfer scenario described in Fig. 3a

5.2.2 Mapping conceptual entities to UML/MARTE

Even though UML sequence diagrams can already capture several abstractions of real-time applications, a number of concepts defined in our conceptual model do not have appropriate counterparts in pure UML. Specifically, the schedulability abstractions, and the timing and concurrency attributes of our conceptual model, are captured by entities defined in UML/MARTE. In particular, UML/MARTE defines a Generic Quantitative Analysis Modeling (GQAM) package, which is intended to provide a generic framework for collecting information required for performance and schedulability analysis. The domain model of this package includes two key abstractions that closely resemble tasks and activities in our conceptual model, namely *Scenario* and *Step*, respectively. A step is defined in the domain model of GQAM as a unit of execution, while a scenario is defined a sequence of steps. Note that, in particular, a *Step* is a *Scenario*, entailing that steps can be defined at different level of abstractions and can possibly be refined into sub-steps. Therefore, a *Task* in our conceptual model can be represented by a *Step* that contains other sub-steps, and an *Activity* can be represented as an atomic *Step* that is not detailed by other sub-steps. *Scenario* and *Step* are represented in GQAM by the stereotypes «*GaScenario*», and its specialization «*GaStep*», respectively. However, entities in the GQAM package are refined in the schedulability analysis modeling (SAM) package, which defines abstractions for schedulable resources, such as tasks, activities, and buffers. In particular, SAM defines the stereotype «*SaStep*» as a specialization of «*GaStep*» which includes details on scheduling abstractions such as offset and deadlines. Therefore, we map both *Task* and *Activity* to «*SaStep*».

Note that «*SaStep*» can be applied to a wide set of behavior-related elements in the UML metamodel, and in particular, to elements in sequence diagrams. In particular,

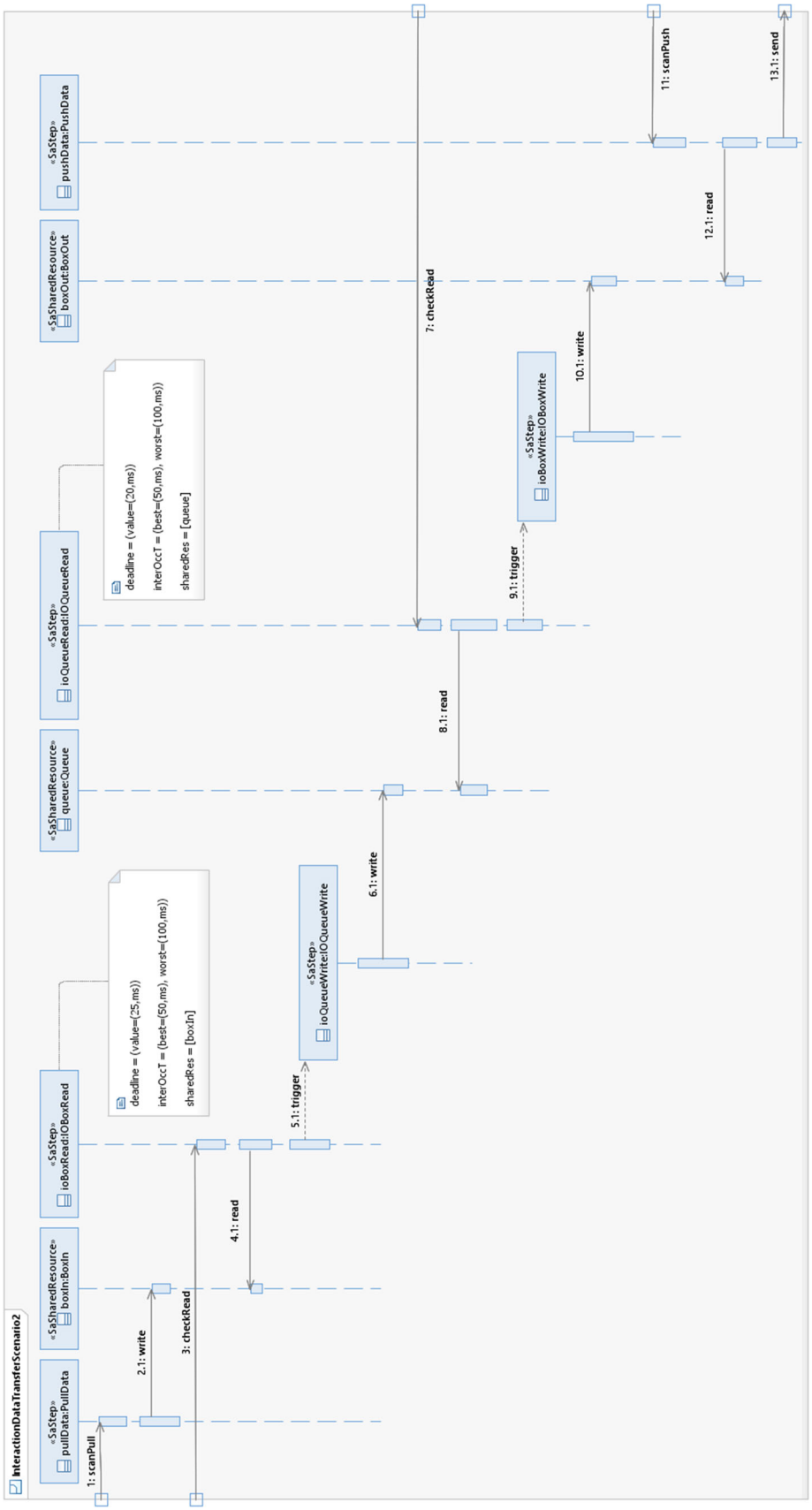

Fig. 10 Sequence diagram modeling the data transfer scenario described in Fig. 3b

«*SaStep*» inherits from both «*TimeModels::TimedProcessing*», which extends the UML metaclasses *Behavior*, *Message*, *Actions*, and «*GRM::Resource*», which extends *NamedElement*. In UML/MARTE, *Step* also includes a list of measures that are widely used for analyzing of real-time properties of embedded systems. We map attributes of tasks and activities in our conceptual model to the stereotype properties of «*SaStep*» representing those measures.

Specifically, we map *interOccTime*, the time interval between two successive occurrences of scenarios, to period, minimum, and maximum interarrival times of task. Note that *interOccTime* is a stereotype property of type *NFP_Duration*, a composite type that is generalized by *NFP_Real*. In addition to the inherited *value* type attribute, which represents a floating point, *NFP_Duration* also defines two additional *NFP_Real* attributes, namely *best* and *worst*. These values semantically represent the smallest and greatest values for *NFP_Duration*, in case the *value* attribute has no fixed value. This design of *NFP_Duration* allows to specify time durations either as a range, or as a fixed value. Therefore, we map the *period* of periodic tasks, which is fixed, to *interOccTime.value*, and *minIa* and *maxIa* to *interOccTime.worst*, and *interOccTime.best*, respectively.[6] This is because, intuitively, the worst case with respect to RTESs performance occurs when tasks arrive at high rate, hence with small interarrival times. We map the activities *minDelay* and *maxDelay* to *selfDelay.best* and *selfDelay.worst*, respectively. This is because one can configure shorter task delays, and hence a more reactive system, only when the CPU usage is not constrained to avoid overtaking the system processor cores. We also map both the periodic tasks *offset* and the activities *release* time to *readyT.value*. This is because, in scheduling theory, the offset models the time a periodic task waits after its triggering event, while the release models the time an activity waits after its preceding one. Finally, we map the *duration* of activities, i.e., their worst-case execution time, to *execTime.value*, the *deadline* of tasks to *deadline.value*, and their *priority* to the *NFP_Integer* attribute *priority*. In UML/MARTE, duration values are expressed using floating points, encapsulated in the *NFP_Real* type. However, we base our analysis for performance tuning and stress testing on discrete time values (Sect. 4.2). In the IEEE 754 single-precision binary floating-point format, which is the most commonly used standard to encode floating points, integers in the interval $[-2^{24}, 2^{24}]$ can be represented without loss of precision [50]. Previous experiments with search-based strategies for performance analysis and testing have been successfully validated with time values in the range of 10^4 [17]. Therefore, the interval above is large enough to assume that, for our purposes, integer values can be type-safely expressed as floating points.

We map our notion of buffer to the «*SaSharedResource*» stereotype, which is meant to represent entities shared among tasks or activities and regulated by exclusive access. Finally, the abstractions related to the computing platform in our conceptual model are not captured in sequence diagrams, but are rather be represented using UML/MARTE stereotypes applied to entities in class and deployment diagrams, such as *Class* and *Node*. Specifically, the Generic Resource Modeling (GRM) package defines the stereotypes «*Scheduler*» and «*SchedulingPolicy*», which are mapped to the corresponding entities in our conceptual model. Finally, we map processing units to the «*HwProcessor*» stereotype from the Hardware Resource Modeling (HRM) Package, and their number of cores to the stereotype property *nbCores*. Note that it is possible in general to have more than one «*HwProcessor*» in a model. However, in our approach, only one entity, which represents the processor executing the computing platform, should be stereotyped as «*HwProcessor*». Table 1 summarizes the mapping between entities in our conceptual model and stereotypes in UML/MARTE. In the table, we report the preferred UML metaclass each stereotype should extend, and the type for each stereotype property.

In addition to the mapping above, we also point out a set of associations between UML/MARTE stereotypes that are semantically related to associations between entities of our conceptual model (Table 2). Note that not all of these associations encapsulate data needed to enable the definition of our constrained optimization problem for performance tuning and stress testing. However, the semantic analogies between associations in UML/MARTE and associations in our conceptual model reinforce our mapping. For example, «*SaStep*» defines the *parentStep* and *steps* associations, which correspond to the association *belongs* between the classes *Task* and *Activity* of our conceptual model. In a similar way, *sharedRes* models an activity using a shared resource with exclusive access, similar to the relationship *uses* between the classes *Activity* and *Buffer* in our conceptual model.

5.3 Pruning the UML/MARTE metamodel

UML and UML/MARTE are undeniably complex standards containing a large number of entities which account for a wide range of modeling needs. For example, UML 2.0 defines over 200 metaclasses and nearly 600 properties, while UML/MARTE in addition contains over 150 stereotypes and 500 stereotype properties that extend the UML metamodel. However, in practice, one only needs a subset of all the entities in UML and UML/MARTE in order to address a specific modeling need. In Sect. 5.2 we argue that, in order to conveniently represent a model for performance tuning and stress testing, we only need the four UML/MARTE

[6] UML/MARTE does not specify whether *best* corresponds to the minimum time and *worst* to the maximum, or vice versa. Hence, this decision is left to the modelers applying UML/MARTE, and can be taken arbitrarily depending on context factors of the target system.

Table 1 Mapping of the entities in our conceptual model to UML/MARTE. *PeriodicTask*, *AperiodicTask* and *TriggeredTask* do not appear in the mapping because they inherit the stereotypes from their superclass *Task*

Conceptual model	UML/MARTE		
Class/attribute	Stereotype/stereotype property	Extended metaclass/type	Sub-profile
Application			
Task	«*SaStep*»	*Lifeline*	*MARTE_AnalysisModel::SAM*
Task::deadline	*SaStep::deadline.value* [a]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
Task::priority	*SaStep::priority*	*NFP_Integer*	*MARTE_AnalysisModel::SAM*
PeriodicTask::offset	*SaStep::readyT.value* [a,b]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
PeriodicTask::period	*SaStep::interOccT.value* [a,c]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
AperiodicTask::maxIa	*SaStep::interOccT.best* [a,c]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
AperiodicTask::minIa	*SaStep::interOccT.worst* [a,c]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
Activity	«*SaStep*»	*ExecutionSpecification*	*MARTE_AnalysisModel::SAM*
Activity::minDelay	*SaStep::selfDelay.best* [a]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
Activity::maxDelay	*SaStep::selfDelay.worst* [a]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
Activity::duration	*SaStep::execTime.value* [a,c]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
Activity::release	*SaStep::readyT.value* [a,b]	*NFP_Real*	*MARTE_AnalysisModel::SAM*
Buffer	«*SaSharedResource*»	*Lifeline*	*MARTE_AnalysisModel::SAM*
Platform			
Scheduler	«*Scheduler*»	*Class/Node* [d]	*MARTE_Foundations::GRM*
Scheduler::policy	*Scheduler::schedPolicy*	*SchedPolicyKind*	*MARTE_Foundations::GRM*
SchedulingPolicy	«*SchedPolicyKind*»	*Enumeration*	*MARTE_ Library::GRM_ BasicTypes*
ProcessingUnit	«*HwProcessor*»	*Class/Node* [d]	*MARTE_DesignModel::HRM::HwLogical::HwComputing*
ProcessingUnit::nbCores	*HwProcessor::nbCores*	*NFP_Integer*	*MARTE_DesignModel::HRM::HwLogical::HwComputing*

[a] We assume that stereotype properties of type *NFP_Real* can represent integers in the interval $[-2^{24}, 2^{24}]$ without loss of precision.
[b] Periodic task offsets and activity release times are both mapped to *SaStep::readyT.value*, because they both represent the time a task or activity waits after its arrival before starting to execute.
[c] *SaStep* can in general have more than one *interOccT* and *execTime* attribute, but in our approach we only need one. This is because each of such attributes is used to represent the period, interarrival time, or duration of a *Task*, and each *Task* only has one period, interarrival time, and duration.
[d] Both *Class* and *Node* can be stereotyped by *Scheduler* and *HwProcessor*. The choice depends on the level of modeling abstraction at which one wants to represent the computing platform

stereotypes «*SaStep*», «*SaSharedResource*», «*Scheduler*», and «*HwProcessor*». Therefore, we *prune* the unnecessary concepts in UML/MARTE, leaving only the four stereotypes above, along with their dependent stereotypes.

However, pruning the UML/MARTE profile is a challenging task which is complicated by the fact that UML/MARTE is formally defined as a profile extending UML, rather than a stand-alone metamodel. The semantics of the concepts introduced in UML/MARTE are defined externally in the profile domain models, which are in turn expressed as partially connected metamodels with textual descriptions. This means that pruning UML/MARTE outputs a metamodel that does not include semantic description of its elements, and hence has to be complemented with appropriate semantics. However, the main goal of our metamodel pruning is to identify a minimal subset of the UML/MARTE stereotypes that contains the abstractions needed to apply our methodology for performance tuning and stress testing. Therefore, rather then defining a completely new semantics, we adopt that in the UML/MARTE specification [47]. Note that, while this semantics enables a sound definition of a pruned UML/MARTE metamodel for performance tuning and stress testing, only the stereotypes and stereotype properties in Table 1 are strictly needed to represent the input data for our OPL model.

We base our pruning on the Eclipse Modeling Framework (EMF)[7] implementation of the UML/MARTE standard, where both stereotypes and classes are *EClass* entities defined in the *Ecore* metamodel.[8] For instance, in EMF, all instances of stereotype «*SaStep*» are instances of the *EClass SaStep*. Using the EMF implementation of UML/MARTE

[7] http://www.eclipse.org/modeling/emf/.

[8] The semantics of *Ecore* is aligned to that of UML and UML/MARTE.

Table 2 Relevant associations of the UML/MARTE stereotypes mapped to our conceptual model

Stereotype	Relevant associations	Inherited from	Notes
«*SaStep*»	*parentStep: GaStep* [1..*]	«*GaStep*» → «*GaScenario*»	The «*GaStep*»(s) of which this «*SaStep*» is a refinement. When applied to activities, it is the task which the activity belongs to
	steps: GaStep [1..*]	«*GaStep*» → «*GaScenario*»	The «*GaStep*»(s) that compose this «*SaStep*». When applied to tasks, it is the set of activities that the task consists of
	sharedRes: SaSharedResource [*]		The set of «*SaSharedResource*»(s) that this «*SaStep*» shares with other «*SaStep*»(s). It represent the set of buffers that tasks or activities communicate with
«*SaSharedResource*»	*scheduler: Scheduler* [0..1]	«*MutualExclusionResource*»	The *Scheduler* that implements the protection protocol, usually the same that schedules the tasks
«*Scheduler*»	*host: ComputingResource* [0..1]		The computing resource («*HwProcessor*») on which the scheduler runs. Typically the same computing resource whose access the scheduler controls
	protectedSharedResources: MutualExclusionResource [0..*]		The mutually exclusive access resources («*SaSharedResource*») whose access is regulated by this «*Scheduler*». It represents the list of buffers of the target system
	processingUnits: ProcessingResources [0..*]		The computing resource («*HwProcessor*») whose access is regulated by this «*Scheduler*». Typically the same computing resource on which the scheduler runs

The arrows (→) indicate chains of generalizations. For example, «*SaStep*» is generalized by «*GaStep*», which in turn is generalized by «*GaScenario*»

allows us to prune pure EMF models, without the issue of separately handling profiles whose semantics is externally defined. While this choice restricts the generality of our metamodel pruning, we argue that EMF implementation is the de facto reference implementation of the Essential Meta-Object Facility (EMOF), which is the reference metametamodel of OMG metamodels.

Metamodel pruning is performed by an algorithm defined in our previous work [22] that outputs an effective subset metamodel of a possibly large input metamodel. The output metamodel preserves a set of required types and properties, given as input to the algorithm, and all its dependencies, which are computed by the algorithm. Every other type and property is pruned. In the type-theoretic sense, the resulting effective metamodel is a supertype of the large input metamodel.

The principle behind pruning is to preserve a set of required types T_{req} and required properties P_{req}, and prune the rest in a metamodel. In previous work [22], we present a set of rules that, given a metamodel *MM* and an initial set of required types and properties, determine a set of required types T_{req} and required properties P_{req}. The initial set may come from various sources, such as manual specification, or a static analysis of model transformations to reveal used types. For example, a rule in the set may add all super classes of a required class into T_{req}. Similarly, if a class is in T_{req} or is a required class, then each of its properties with a multiplicity lower bound greater than zero is added to P_{req}. Apart from rules, there are options which allow a better control of the algorithm. For example, one of such options states that if a class is in T_{req}, then all of its subclasses are added into T_{req}. This optional rule may be applicable under certain circumstances, giving the user freedom in deciding the extent to which the *MM* is pruned. The rules are executed on elements of *MM* where the specified conditions match, until no rule can be executed any longer. The algorithm always terminates for a finite metamodel, because the rules do not remove elements from the sets T_{req} and P_{req}. Once the algorithm computes T_{req} and P_{req}, it finally removes the remaining types and properties from *MM* to output the effective metamodel MM_e. In Fig. 11, we present a pruned version of the UML/MARTE metamodel, which is a sufficient subset of UML/MARTE for performance tuning and stress testing of RTESs. This effective metamodel contains 26 classes and 64 associated properties.

MM_e has a number of noteworthy characteristics. In particular, using *model typing* [23], it is possible to verify that MM_e is a *supertype* of the input metamodel *MM*. Model type conformance (or substitutability) has been adapted and extended to model types based on Bruce and Vanderwaart notion of type group matching [51]. The matching relation between two metamodels, denoted $<\#$, defines a function of the set of classes the metamodels contain according to the following definition [23]. A Metamodel M' matches another metamodel M, denoted $M' <\# M$, iff for each class C in M, there is one and only one corresponding class or sub-

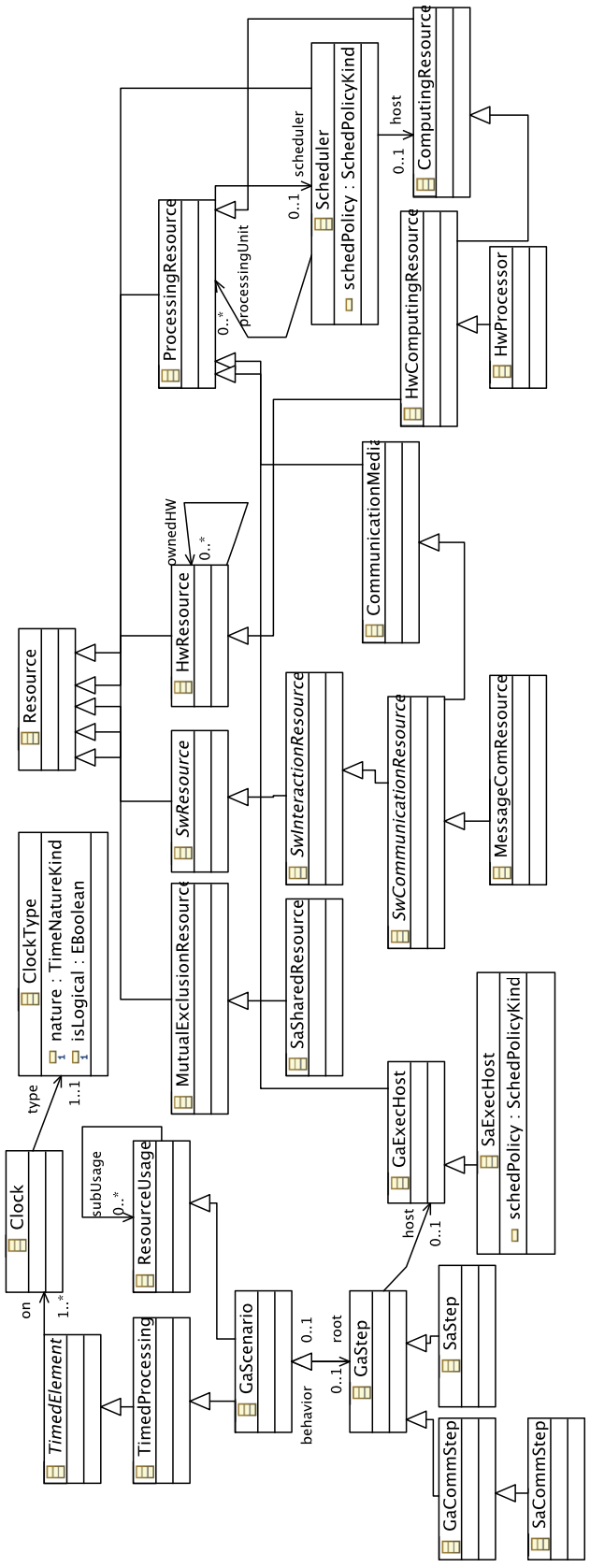

Fig. 11 Pruned UML/MARTE metamodel for performance tuning and stress testing of RTES

```
1  tq = 100;
2
3  c = 2;
4
5  J = {
6      #<id:"j0", priority:0, deadline:30, period:15,
           min_delay:0, max_delay:0,
7      min_interarrival_time:-1, max_interarrival_time
           :-1, duration:5>#,
8      #<id:"j1", priority:1, deadline:25, period:-1,
           min_delay:1, max_delay:1,
9      min_interarrival_time:15, max_interarrival_time
           :100, duration:4>#,
10     #<id:"j2", priority:2, deadline:20, period:-1,
           min_delay:0, max_delay:2,
11     min_interarrival_time:10, max_interarrival_time
           :20, duration:4>#,
12 };
13
14 DS = {<<"j0">, <"j1">>, <<"j1">, <"j0">>};
15
16 TS = {};
```

Fig. 12 Example of an OPL data file for a dual-core system with three tasks, one dependency, and no triggering relations

class C' in M' such that every property p and operation op in $M.C$ matches in $M'.C'$, respectively, with a property p' and an operation op' with parameters of the same type as in $M.C$. This notion of type conformance implies that all operations written for MM_e are also valid for MM. In our case, the pruned UML/MARTE metamodel is a supertype of the *Ecore* UML/MARTE metamodel, implying that all mappings and transformations defined on the pruned UML/MARTE are also well defined for models instances of the full *Ecore* UML/MARTE metamodel.

5.4 Mapping the pruned UML/MARTE metamodel to OPL

Recall from Sect. 4.2 that the ultimate goal of our approach is to support the generation of system configurations characterized by task delay times, and stress test cases characterized by arrival times of aperiodic tasks. In order to do so, we devise two constrained optimization problems (COPs) over the abstractions defined in our conceptual model. The COPs differ only for the definition of the independent variables of the search, i.e., task deadlines or aperiodic task arrival times, and for the optimization directive, i.e., minimization or maximization. The constraint models are specified in the Optimization Programming Language (OPL), a widely used modeling language to specify optimization problems. One of the key features of OPL is the separation of the model logic, i.e., the definition of constants, variables, constraints, and objective functions, from the data, i.e., the constant values. While the model logic is part of previous work [21,28], in this paper we focus on the definition of the model data file. Indeed, to foster adoption within model-driven engineering development processes, we propose a mapping between stereotypes and stereotype properties in the pruned UML/MARTE metamodel and entities in the OPL data specification format. Figure 12 reports an example OPL file with an observation interval of 100 time quanta ($tq = 100$), modeling a set J of 3 tasks, j_0, j_1, and j_2 running on a dual-core platform ($c = 2$). j_0 is periodic with period 15 time quanta, j_1 is aperiodic with interarrival time between 15 and 100 time quanta, and j_2 is also aperiodic with interarrival time between 10 and 20 time quanta. Note that the minimum and maximum interarrival times of periodic tasks, and the period of aperiodic tasks are set to -1. Furthermore, j_0 and j_1 share a computational resource with exclusive access, since the set DS of dependency relationships contains the symmetric tuples (j_0, j_1) and (j_1, j_0). There are no triggering relations, as the set TS is empty. Note that the five building blocks of the OPL data file in Fig. 12 are (1) the specification of the number of time quanta in the observation interval (line 1) (2) the specification of the number of cores (line 3), (3) the specification of the software tasks (lines 5–12), (4) the specification of the dependency (line 14), (5) and triggering relations (line 16). These properties are defined at the lines 4, 8, 24, 25, and 26 in the excerpt of the OPL implementation of the constraint models (Fig. 6).

By construction, the data specified in OPL has the same semantics as the stereotypes and stereotype properties in the pruned version of the UML/MARTE metamodel. Table 3 shows the mapping between stereotypes and stereotype properties in the pruned metamodel to the OPL elements defining the COPs data. Such mapping forms the basis of the UML/MARTE to OPL model transformation, that we describe hereby.

We transform UML/MARTE models to OPL using Acceleo,[9] an open-source implementation of the OMG MOF Model to Text Language (MTL) standard that allows to navigate *Ecore* models and generate code based on one or more templates. In the rest of this section, we describe the model transformation that generates the five building blocks of OPL data files. In particular, we first describe the generation of the number of time quanta and cores of the executing platform (Sect. 5.4.1), then we describe how OPL tasks are extracted from a UML/MARTE model (Sect. 5.4.2), and finally we present the templates generating the dependency and triggering sets (Sects. 5.4.3, 5.4.4).

5.4.1 Generating the number of time quanta and the number of cores

Recall from Sect. 4 that the number of time quanta in the OPL model is a parameter of the search, which is not specified in UML. Therefore, we generate for it a placeholder with a dummy value N. Note that, in practice, the user would need

[9] https://eclipse.org/acceleo/.

Table 3 Mapping of the identified stereotypes and tagged values of UML/MARTE to elements in OPL

Stereotype/tagged value	OPL element	Note
Application		
«*SaStep*»	*J*	Each task or activity is modeled as a task in the task set
SaStep::deadline.value	*deadline*	
SaStep::priority	*priority*	
SaStep::readyT.value	*offset*	
SaStep::interOccT.value	*period*	Set to −1 if not specified, i.e., if the task is aperiodic
SaStep::interOccT.best	*max_interarrival_time*	Set to −1 if not specified, i.e., if the task is periodic
SaStep::interOccT.worst	*min_interarrival_time*	Set to −1 if not specified, i.e., if the task is periodic
SaStep::selfDelay.best	*min_delay*	Set equal to *max_delay* if the task delay is constant, i.e., not part of the configuration
SaStep::selfDelay.worst	*min_delay*	Set equal to *max_delay* if the task delay is constant, i.e., not part of the configuration
SaStep::exceTime.value	*duration*	
«*SaSharedResource*»	*DS*	The dependent tasks of a task *j* are the tasks that share at least one end of the *sharedRes* association with *j*
Platform		
«*Scheduler*»	*MC, PC, SEC constraints*	The scheduler and the scheduling policy are captured by the Multi-core (MC), Preemption (PC), and Scheduling Efficiency (SEC) constraints of the model [21]. Note that, in this article, we only consider fixed-priority preemptive scheduling policies
«*SchedPolicyKind*»	*MC, PC, SEC constraints*	
«*HwProcessor*»	*c*	The processing platform and its number of cores are captured together as a constant integer value
HwProcessor::nbCores	*c*	

Note that the triggering set *TS* in OPL is not explicitly mapped from MARTE elements, because is derived from standard UML *Message* objects

```
1 [template public generateOpl(aModel: Model)]
2         tq = N;
3         c = [aModel.getOwnedMembers()->selectByType
              (HwProcessor).nbCores.toString()/];
4 [/template]
```

Fig. 13 Acceleo template that generates the number of time quanta and processor cores

to manually specify this value. We navigate to the object of type *HwProcessor* in the container *Model* to extract the number of cores, as shown in Fig. 13. This is done invoking the *getOwnedMembers* method on the input *Model*, and specifically requiring via *selectByType* the *HwProcessor* member.

5.4.2 Generating the task set

We generate the task set J and the tasks $j \in J$ using the template in Fig. 14. We navigate through every *SaStep* object in the *Model* to extract the tasks properties. In particular, we generate the task *id* from the name of the *base_NamedElement* of the *SaStep* (line 2), and the task *priority* from the corresponding *SaStep* property (line 3).

Note that the other task proprieties, such as deadlines and periods, are mapped to values of type *NFP_Real*, which are

```
1  [template public generateJobs(aSaStep : SaStep)]
2      #<id:"[aSaStep.base_NamedElement.name/]",
3      priority: [aSaStep.priority/],
4      deadline: [extract('value', aSaStep.deadline/],
5      period:[extract('value',aSaStep.interOccT->
           asOrderedSet()->at(1))/],
6      min_interarrival_time:[extract('best',aSaStep.
           interOccT->asOrderedSet()->at(1))/],
7      max_interarrival_time:[extract('worst',aSaStep.
           interOccT->asOrderedSet()->at(1))/],
8      duration:[extract('value',aSaStep.execTime->
           asOrderedSet()->at(1))/]>#
9  [/template]
10
11 [query public extract(property: String, line:String
       ):String = invoke('no.certus.simula.sosym.mt.
       main.Utility','extract(java.lang.String,java.
       ang.String)',Sequence{property,line})/]
```

Fig. 14 Acceleo template that generates the task set

specified as expressions in the UML/MARTE Value Specification Language (VSL). These expressions are persisted as *String* objects, which have to be parsed. Given the limited support for string matching in Acceleo, we implemented an ad hoc regular expression parser in Java, which is defined externally to the template and called through an Acceleo *query* (line 11). The regular expression matches the following pattern:

```
1 [template public generateDS(aSaStep : SaStep)]
2     [for (aSaSharedResource : SaSharedResource  |
          aSaStep.sharedRes->asOrderedSet())
          separator('\n')]
3         <<"[aSaStep.base_NamedElement.name/]", "[
              aSaSharedResource.base_Lifeline.name
              /]">>,
4         <<"[aSaSharedResource.base_Lifeline.name
              /]","[aSaStep.base_NamedElement.name
              /]" >>
5     [/for]
6 [/template]
```

Fig. 15 Acceleo template that generates the task dependency set

```
"("+property+")=\\(([0-9]*),([a-zA-z ]*)\\)"
```

In the pattern, the parameter *property* is specified in the Acceleo template as the first argument of the *extract* function, while the second group of regular expressions after the equals sign matches the value of the property. For example, to extract the value of a task *deadline*, we match the regular expression on the property *value* of *SaStep::deadline*. Also note that, in UML/MARTE, *SaStep::interOccT* and *SaStep::execTime* are two properties whose multiplicity is zero or more. However, in our guidelines, we specify that only one of these attributes is needed for each *SaStep* (Sect. 5.2). Therefore, when extracting values from these two stereotype properties, we only retrieve the first value by invoking the *OrderedSet:at* method with argument 1 (lines 5–8).

5.4.3 Generating the task dependency set

The dependency relations between tasks are extracted by navigating through each *SaStep* object in the *Model*, as shown in Fig. 15. Recall from Sect. 5.2 that we capture the task dependencies through the property *SaStep::sharedRes*, which has type *SaSharedResource*, and that the dependency relation between tasks is symmetric. As shown in Fig. 12, a dependency set in OPL is a set of tuples indexed by the tasks *id*, which in particular contains couples of symmetric tuples. Note that, since the *SaStep::sharedRes* property potentially contains more than one *SaSharedResource*, we have to iterate through the set with a *for* loop.

5.4.4 Generating the task triggering set

Recall from Sect. 5.2 that we capture the triggering relations between tasks through UML *Message* objects whose *messageSort* property has value *create*. Therefore, the triggering set is obtained by navigating through such *Message* objects. The name of the task that sends or receives a message is stored in *MessageOccurrenceSpecification* objects with the same name as the *Message* object. In UML, every *MessageOccurrenceSpecification* is stored together with *Message* objects in an *Interaction*. For this reason, we extract the triggering and triggered task names from the *sendEvent* and *receiveEvent* properties of *MessageOccurrenceSpecification*, respectively. Unfortunately, there is no direct way in Acceleo to navigate directly from a *Message* to the related *MessageOccurrenceSpecification*, and hence, we performed this navigation with two separate *for* loops shown in Fig. 16.

```
1  [template public generateTS(aMS : Message)]
2      [for (aM : MessageOccurrenceSpecification | aMS
           .interaction.ownedElement->selectByType(
           MessageOccurrenceSpecification)) separator
           ('\n')]
3          [if aMS.messageSort.equals("create") && aM.
               name = aMS.receiveEvent.name]
4              <<"[aM.covered.name/]",
5          [/if]
6      [/for]
7
8      [for (aM : MessageOccurrenceSpecification | aMS
           .interaction.ownedElement->selectByType(
           MessageOccurrenceSpecification)) separator
           ('\n')]
9          [if aMS.messageSort.equals("create") && aM.
               name = aMS.sendEvent.name]
10             "[aM.covered.name/]">>
11     [/for]
12 [/template]
```

Fig. 16 Acceleo template that generates the task triggering set

6 Industrial validation

The work reported in this paper originates from the interaction over the years with Kongsberg Maritime (KM). KM faces important challenges when developing the software components of their real-time systems. Through regular meetings with KM engineers, we first identified the need for a model-based testing approach defining the abstractions required for performance analysis [27]. Then, we focused on the problem of generating stress test cases violating performance requirements. casting it as an optimization problem over a mathematical model of the tasks preemptive scheduling policy. To prepare for industrial adoption, we initially evaluated our methodology in five publicly available case studies of several RTES domains [18]. This preliminary evaluation showed encouraging results when comparing constraint programming with a state-of-the-art optimization strategy based on genetic algorithms [17]. Then we provided a constrained optimization problem (COP) to automate the generation of stress test cases, and successfully evaluated it KM fire and gas monitoring system (FMS) [21]. Finally, we provided a second version of our COP to automate the generation of system configurations characterized by task delay times, and performed a second successful validation in the FMS [20]. The rest of this section briefly summarizes the experimental results from previous papers, adding further

insight on the benefits of applying UML/MARTE to support performance tuning and stress testing in RTES.

In previous validation we investigate whether a framework combining UML/MARTE modeling with constrained programming can effectively be used for performance testing in an industrial context. Therefore, answer three research questions.

1. *RQ1—Overhead* Can the input data to our approach, i.e., the values for the constants in the constraint model, be provided with reasonable effort in an industrial setting?
2. *RQ2.1—Practical usefulness for performance tuning* Can engineers use the output of our first analysis, i.e., the values for the delay time variables in the constraint model, to derive configurations that satisfy the system performance requirements?
3. *RQ2.2—Practical usefulness for stress testing* Can engineers use the output of our second analysis, i.e., the values for the arrival times in the constraint model, to derive stress test cases that violate the system performance requirements?

RQ1: Overhead Given the definition of our approach, all the information required for performance tuning and stress testing is captured by the conceptual model in Fig. 7, and in particular by its quantitative elements. To gather this information, we started by building UML sequence diagrams for the I/O drivers in KM, using the design documents and reverse engineering the implementation of the drivers. The resulting sequence diagrams were iteratively validated and refined together with KM engineers. Our industry collaboration confirmed the common belief that sequence diagrams are among the preferred methods for industrial practitioners to visualize concurrent software [27]. The timing and concurrency data captured by our conceptual model was obtained from certification design documents (task architecture and periods), drivers source code (priority and offsets), and performance profiling logs (minimum and maximum interarrival times, and estimates for the WCET of tasks). A total of 25 man-hours of effort spanned across 8 days were spent to obtain the final version of the sequence diagrams stereotyped with UML/MARTE. This was considered worthwhile by KM, because safety-critical I/O drivers are regularly certified and have long life time. The computing platform information, such as the number of processor cores and the scheduling policy, was extracted from configuration and hardware design documents. Note that the provided mapping to UML/MARTE allows any modeling development environment that complies to the model-driven architecture to use the input notation we provided. Furthermore, the pruned UML/MARTE metamodel and its mapping to OPL constitute important steps toward the fully automation of our framework, enabling the unassisted generation of data characterizing system configurations and stress test cases starting from UML/MARTE sequence diagrams.

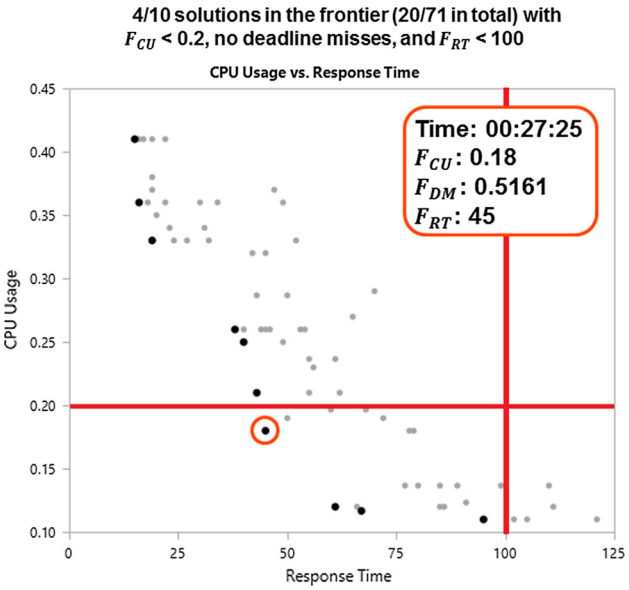

Fig. 17 Pareto-optimal frontier of F_{CU} and F_{RT} [20]

RQ2.1: Practical usefulness for performance tuning Recall from Sect. 2 that we characterize system configurations by delay times between activities in the *IODispatch* task of the FMS drivers. Therefore, such delay times are the main variables in our constraint model for performance tuning. We performed an experiment with the FMS drivers driving the search with a lexicographic multi-objective criteria [52]. In lexicographic ordering, the first criterion is considered as the most important one, and any improvement of the criterion is worth any loss on the other criteria. The second criterion is the second most important one, for the improvement of which only losses on the first criterion are not allowed. The last criterion is the least important one. Using multi-objective optimization allows us to identify a Pareto-efficient frontier of solutions achieving an optimal trade-off between the search criteria t [53]. We run our model for six times, one for each of the permutations of F_{DM} (deadline misses), F_{RT} (response time), and F_{CU} (CPU Usage). Experimental results [28] show an opposing trend between response time, and CPU usage, as confirmed by previous research [54]. Specifically, low CPU usage leads to high response time, while low response time is usually only possible with high CPU Usage.

Figure 17 shows the Pareto-optimal frontier of F_{CU} and F_{RT}, whose solutions are highlighted with a solid bullet (•). The circle (◯) highlights the first solution found in the frontier that satisfies all the requirements, for which we report the computation time and the objective values. The two lines orthogonal to the x and y-axes represent the

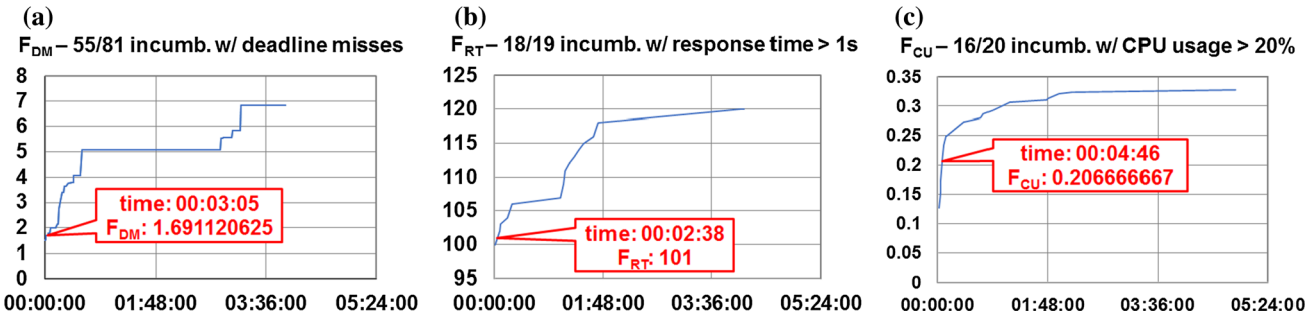

Fig. 18 Objective values of F_{DM}, F_{RT}, and F_{CU} over time, where we highlighted the time when the first incumbent predicted to violate a performance requirement was found [21]. **a** F_{DM} value over time., **b** F_{RT} value over time, **c** F_{CU} value over time

maximum threshold on F_{RT} and F_{CU}, respectively. Over the six runs, the search found 71 solutions, 20 of which satisfying the performance requirements. 10 solutions out of the total 71 are in the frontier, and 4 out of the 10 satisfy the performance requirements. The first of such solutions was found in approximately 27 minutes. By definition, the solutions in the frontier do not Pareto-dominate each other, entailing that for each solution in the frontier there does not exist any other solution with a lower CPU usage *and* a lower response time. Therefore, the solutions in the Pareto frontier achieve an optimal trade-off between CPU usage and response, and can be used by engineers to derive drivers configurations that are as likely as possible to exhibit low CPU usage, task deadlines, and response time.

RQ2.2: Practical usefulness for stress testing Similar to the case of performance tuning, the main goal of evaluating our approach for stress testing is to investigate whether engineers can use the solutions of our COP to derive stress test cases for task deadlines, response time, and CPU usage. We performed an experiment with the FMS drivers running our OPL model for three times, once for each performance requirement. Figure 18 shows the feasible solutions with the best objective value that were found within five hours. Consistent with the terminology used in integer programming, we refer to these solutions as *incumbents* [55]. In each graph, the x-axis reports the incumbent computation times in the format *hh:mm:ss*, and the y-axis reports the corresponding objective value.

Since software testing has to accommodate time and budget constraints, we investigated the trade-off between the time needed to generate test cases, and their power for revealing violations of performance requirements, recording the computation times of the first incumbents predicted to violate the three performance requirements as expressed in Sect. 2. The run optimizing F_{DM} is shown in Fig. 18a. The solver found 55 out of a total of 81 incumbents with at least one deadline miss in their schedule; the first of such solutions was found after three minutes. The solution yielding the best value for F_{DM} produced a schedule where the *PushData* task missed its deadline by 10 ms in three executions over T. Figure 18b shows the results for the run optimizing F_{RT}. The solver found 18 out of 19 incumbents with response time higher than 1 s; the first of such solutions was found after two minutes. The best solution with respect to F_{RT} produced a schedule where the response time of the system was 1.2 seconds. Finally, the solutions found by optimizing F_{CU} are shown in Fig. 18c. The solver found 16 out of 20 incumbents with CPU usage above 20%; the first of such solutions was found after four minutes. The solution with the highest value for F_{CU} produced a schedule where the CPU usage of the system was 32%. For each objective function, the solver was able to find, within a few minutes, solutions that are candidates to stress test the system as they may lead to requirements violations. Note that these solutions can be used to start testing the system while the search continues, because the highest the objective value, the more likely the solutions are to push the system to violating its performance requirements.

7 Toward a UML/MARTE framework for performance tuning and stress testing: challenges, experiences and lessons learned

Several recent studies report that applying model-driven engineering (MDE) methodologies in industrial contexts is a task complicated by several factors [56]. This is especially true when it comes to applying UML, which is one of the cornerstones of several processes implementing the model-driven architecture. Over the years, an increasing number of practitioners and researchers have raised the concern that the UML flexibility in catering a wide range of modeling notations comes at the cost of the standard being too large and hence hard to use in practice [57]. Similar concerns have also been raised on UML/MARTE [58], which, despite being introduced in 2007, still faces a severe lack of training material. In this regard, the single notable exception is rep-

resented by the recent work of Selic and Gérard [59], which complements the UML/MARTE specification with practical guidelines. Note that, even though this specification is not targeted at end users, it is still one of the few sources available containing examples on how to use the modeling notation provided in UML/MARTE. This means that, when applying the profile, practitioners often face the problem of not having a clearly defined starting point, because there is no general high-level methodology on how to use UML/MARTE in particular contexts. Therefore, work in this direction is left to researchers, who have to define generic methodologies and frameworks for specific needs, similar to that introduced in this article with respect to performance tuning and stress testing.

The lack of training material on UML/MARTE is unfortunately not mitigated by adequate tool support. While integrated development environments (IDEs) for programming have evolved into highly dependable, configurable, and flexible frameworks, modeling tools are still very far from that level of usability. Only three tools officially support UML/MARTE modeling according to OMG,[10] namely Magic Draw,[11] IBM Rational Software Architect (RSA),[12] and Papyrus.[13] Our experience in using these tools reflects that of recent empirical studies raising concerns on their user-friendliness [60]. In particular, while we found Papyrus to be the tool with the better support of UML/MARTE, it still requires the user to be familiar with the UML/MARTE metamodel. Furthermore, as already reported in the literature [61], we found the dependability of Papyrus to be questionable. However, the alternative UML/MARTE implementations, i.e., those for Magic Draw and RSA, are aligned with old tool versions, and do not seem to be actively maintained.

Furthermore, the small amount of significant reported evidence on the benefits of applying UML/MARTE considerably hinders the industrial adoption of the profile. This aspect, combined with the small amount of training material available and the lack of proper tool support, makes practitioners see applying UML/MARTE as an investment with a high cost, and potentially low returns. To effectively model and analyze RTESs, defining customized domain-specific languages (DSLs) from scratch, with ad hoc notation and semantics, is seen as such an easier alternative, that is worth the cost of dropping the alignment with widely recognized standards such as UML. While for better tool support a significant conjunct effort from the community, vendors, and organizations is needed, in this article we attempt at mitigating the lack of methodologies on how to use UML/MARTE, and reported industrial applications.

[10] http://www.omgmarte.org/node/31.

[11] http://www.nomagic.com/products/magicdraw.

[12] http://www.ibm.com/developerworks/.

[13] http://eclipse.org/papyrus/.

7.1 Issues encountered and potential modeling alternatives

Currently, engineers in Kongsberg Maritime spend several days simulating the behavior of the FMS and monitoring its performance requirements. We expect that, by following the systematic strategy proposed in this paper, they can both (1) properly configure the delay times in the FMS I/O drivers, and (2) stress test them under the worst operating conditions, so that no safety risks are posed by violating performance requirements at runtime. While engineers in KM acknowledged the usefulness of our approach, we cannot report results on whether the identified configurations and stress test cases lead to actual satisfactions or violations of performance requirements at runtime. Even though this is a clear threat to the validity of our industrial validation, we argue that the work presented in this article is a basis to exploit in future work, both to further assess the applicability of UML/MARTE in industrial contexts, and to define UML/MARTE-compliant DSLs via the proposed pruned metamodel.

An important issue we faced concerns the way values of type *NFP_Duration* are specified within UML/MARTE. Recall from Sect. 5 that we use values of the *NFP_Duration* supertype *NFP_Real* to represent the real-time properties related to software tasks (Table 1). In particular, values of type *NFP_Real* are specified according to the Value Specification Language (VSL), which is also defined in the UML/MARTE specification. In VSL, the value of a *NFP_Real* is represented by tuple expressions of the form (v, u), where v represents the value and u represents the unit, e.g., (10, ms). In particular, *NFP_Duration* contains several attributes of type *NFP_Real*, such as *best*, *worst* and *value*, which we use in the mapping of our conceptual model to UML/MARTE. The values of the *NFP_Duration* attributes should be specified in such a way that the values are explicitly linked to the attributed they refer to. For example, the expression $\big(best = (15, \text{ms}), value = (20, \text{ms}), \text{worst} = (25, \text{ms})\big)$ represents an *NFP_Duration* whose *best*, *value*, and *worst* attributes are 15, 20, and 25 ms, respectively. However, VSL allows label-less expressions where tuples (v, u) are not explicitly linked to their attributes. For example, an *NFP_Duration* can also be specified with the expression $\big[(15, \text{ms}), (20, \text{ms}), (25, \text{ms})\big]$. In this label-less expression, it is hard to relate *NFP_Real* values to their attributes, because UML/MARTE does not specify a default order for such attributes. Note that, along with several others, this is still an open issue which will hopefully be fixed in the next

iteration of the specification.[14] Therefore, we recommend to avoid label-less expressions for subtypes of *NFP_Real*, because they can often lead to confusion. This also important in order to ensure that the model-to-text transformation from UML/MARTE to OPL is able to correctly parse VSL expressions.

We also noted that a number of semantic inconsistencies with UML, also reported by others [56], can render models potentially hard to understand. Consider for instance the stereotype «*HwProcessor*» from the Hardware Resource Modeling package, which we use to model the processing unit of the RTES hardware platform. This stereotype is typically used on a UML Class, which represents a hardware component in a class diagram. However, an association of an «*HwProcessor*» class with another class which is not stereotyped is potentially ambiguous. This is because UML is typically used to model software, and, without any stereotype applied, an UML class models a software entity by default. For this reason, an association between a class modeling a hardware component and a traditional class modeling a software entity should be given a specific meaning, such as the deployment of the software to its hardware platform. As it is often the case when devising methodologies to apply UML/MARTE, these potential inconsistencies are addressed in the modeling guidelines we propose (Sect. 5.2).

Furthermore, we found that there exist several abstractions in UML/MARTE that could possibly be mapped to the concepts presented in our conceptual model. The most notable example in this respect is the stereotype «*RtFeature*» from the High-Level Application Modeling (HLAM) package, which is used to annotate UML elements with real-time properties, which are in turn specified in UML comments stereotyped «*RtSpecification*». While HLAM does not provide the same support as the SAM package for schedulability analysis, «*RtSpecification*» allows to model arrival patterns for real-time events, such as triggering of tasks, at a finer-grained level than «*SaStep*». In particular, the attribute *occKind* of «*RtSpecification*» allows to specify events arrival patterns through the type *ArrivalPattern*, including periodic and aperiodic events. This means that, to achieve a more powerful description of events arrival patterns, we could have used the stereotypes «*RtFeature*» and «*RtSpecification*» alongside «*SaStep*». Even though it is possible in UML to apply multiple stereotypes to the same entity, we deemed this to be potentially confusing for practitioners using our UML/MARTE approach. In addition, recall from Sect. 4 that, to effectively generate configurations and stress test cases, we only need the tasks period, and bounds for interarrival times. These properties can be easily represented through the *value*, *best*, and *worst* attributes of the *interOccT* property of «*SaStep*», as shown in Table 1.

[14] http://issues.omg.org/issues/spec/MARTE.

7.2 Limitations, generalizability, and scalability of our approach

The approach presented in this article originates from the interaction with Kongsberg Maritime over the years, and hence draws on context factors (Sect. 4) that need to be ascertained prior to successful application. While we have found these factors to be commonplace in many industry sectors relying on RTESs, it is likely that there are more complex scenarios for which our methodology would have to be extended. For example, in this article we only consider a fixed-priority preemptive scheduling policy. However, the stereotype «*Scheduler*», which we use to model the RTOS scheduler, allows the specification of scheduling policies other than fixed priority. Extending the approach to consider different policies would require the definition of a set of constraints in the COP which would be used in place of those modeling the preemptive scheduler behavior (Sect. 4.2). Note that the other constraints of the COP would not need to be modified.

In our methodology, we do not consider constraints on memory and network usage. This is because, in the FMS, the only computational resource contented by tasks which is also constrained by performance requirements is the CPU. Therefore, we do not provide guidelines on modeling abstractions concerning memory or network, such as RAM, hard drives, or buses. In order to do so, we would have to first analyze how memory and network usage affect the way tasks are scheduled. This would lead to extending our conceptual model in order to include concepts related to memory network usage, which would in turn be mapped to entities in UML/MARTE such as «*HW_RAM*», «*HW_Drive*», and «*HW_Bus*». These entities, and their relative attributes, would have to be mapped to constant values in the OPL model, which would in turn be used to define additional constraints in our COP. While this work would undeniably require a considerable effort, we argue that the building blocks of our methodology, namely the conceptual model and the COP, could be extended without the need of being overhauled. In particular, the conceptual model would contain additional entities, but adding them would not require modifying the existing entities and associations. In a similar way, the COP would contain additional constants, variables, and constraints which could be added without the need of modifying the existing ones.

We also note that the scenarios presented in Sect. 2 are characterized by a relatively small number of tasks and interactions. In practice, more complex behavioral scenarios are represented across different sequence diagrams, which nest and cross-reference UML *Interaction* entities. In order to successfully generate the COPs data for these large scenarios, we would have to retrieve values for the stereotype properties across different interactions. This would require modify-

ing the model-to-text transformation from UML/MARTE to OPL data files (Sect. 5.4) by adding the capability to navigate nested interactions.

Recall from Sect. 4 that the ultimate goal of the approach presented in this paper is to support the generation of configurations and stress test cases in RTESs. For this purpose, UML/MARTE is used to conveniently organize the input data for the COP in a standard notation. However, it may be argued whether the approach we propose can effectively be used in contexts where UML models require a high degree of formalization, for example including rules specified in the Object Constraint Language (OCL). In this article, we do not propose guidelines on particular OCL constraints to use in order to ensure well-formedness with respect to our modeling guidelines, and full compliance to UML and UML/MARTE. This is because, provided that the input data for the COP is consistently represented in the stereotype properties in Table 1, our approach is able to generate configurations and stress test cases regardless of the full conformance of the input models to OMG standards.

Finally, it may also be argued whether the inherent complexity of performance tuning and stress testing in large and complex RTESs effectively limits the practical effectiveness of our approach. Indeed, it is likely not possible to exhaustively assess whether a target RTES meets its performance requirement by tuning performance-related parameters or identifying worst-case scenarios. However, previous work in the field shows that doing so is arguably a useful technique to gain confidence in mitigating the risks associated with improper system configurations and unforeseen task interactions [17,54].

8 Conclusions and future work

Performance tuning and stress testing in safety-critical Real-Time Embedded Systems (RTESs) are tasks complicated by several context factors. In particular, the software components of complex RTESs often communicate with a large number of external devices via software drivers smoothing the data transfer between the hardware and software components of the system. Drivers are usually designed as concurrent applications, whose tasks timing can be configured to correctly operate with the specific devices connected. Nonetheless, safety-critical device drivers are often subject to requirements on task deadlines, response time, and CPU usage, which render tuning their timing properties a challenging task, often performed manually based on engineers expertise. The satisfaction of these requirements not only depends on configurable parameters that regulate the tasks timing, but also on unpredictable environmental conditions that trigger the system tasks at runtime. Therefore, it is both necessary to carefully tune the performance-related parameters, and to stress test the system to ensure appropriate responses with respect to external outputs.

In this paper, we presented a methodology, based on UML/MARTE, that models RTESs to support the generation of configurations and stress test cases characterized by RTESs timing properties. In particular, the key idea behind our work is to (1) identify scenarios where tasks are as far as possible from their deadlines, and exhibit low response time and CPU usage (*performance tuning*), and (2) identify scenarios where tasks are as close as possible from their deadlines, and exhibit high response time and CPU usage (*stress testing*). Such scenarios are determined by the way tasks are scheduled to execute at runtime, which in turn depend on the value of timing parameters, and external events triggering the system tasks. Therefore, we enable the definition and the implementation in model-driven engineering development processes of a constrained optimization problem (COP) that finds combinations of timing properties maximizing the satisfaction/violation of performance requirements on deadline misses, response time, and CPU usage. Specifically, we first abstract the problem of generating system configurations and stress test cases by devising a conceptual model that captures, independently from any modeling notations, the key entities needed for our analysis. Then, we map our conceptual model to stereotypes and stereotype properties in the standard UML/MARTE profile for modeling and analyzing the performance of RTESs. We prune from such metamodel the entities that are not needed for our analysis, obtaining a significantly smaller metamodel. The pruned metamodel enables the definition of a mapping between UML/MARTE and the Optimization Programming Language (OPL), which is used to define the data for the COP that generates system configurations and stress test cases. The mapping between UML/MARTE and OPL is implemented as a model-to-text transformation in Acceleo, an open-source implementation of OMG MOF model-to-text language (MTL) standard.

We validate our approach on a RTES from the maritime and energy domain concerning safety-critical device drivers, showing that our approach can be applied with reasonable overhead in an industrial setting, and is able to effectively identify Pareto-optimal delay times with respect to CPU usage and response time in less than half hour, and scenarios predicted to violate performance requirements in a few minutes. While we note that our methodology draws on context factors (Sect. 4) that need to be ascertained prior to successful application, we have found the factors to be commonplace in many industry sectors relying on RTESs. Furthermore, we argue that the model transformation between the UML/MARTE entities required for our analysis and the OPL constructs encapsulating the COP data is a significant step toward the full automation of our approach. Achieving this full automation would require building a framework which encapsulates both the model transforma-

tion from UML/MARTE to OPL, and the resolution of the constraint model via ILOG CPLEX CP Optimizer. These two steps currently have to be performed separately. Nevertheless, we envision that future work on implementing such a framework is not prone to facing significant technology limitations, due to the open-source nature of Acceleo and the interoperability of ILOG CPLEX CP Optimizer with several languages, such as C++ and Java.

References

1. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: FM 2006: Formal Methods, pp. 1–15. Springer, Berlin (2006)
2. Lee, E.A., Seshia, S.A.: Introduction to embedded systems, a cyber-physical systems approach (2011). http://LeeSeshia.org
3. Lala, J.H., Harper, R.E.: Architectural principles for safety-critical real-time applications. Proc. IEEE **82**(1), 25–40 (1994)
4. Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications. Springer, Berlin (2011)
5. Storey, N.R.: Safety Critical Computer Systems. Addison-Wesley Longman Publishing Co. Inc., Redwood City (1996)
6. Gomaa, H.: Designing concurrent, distributed, and real-time applications with UML. In: Proceedings of the 28th International Conference on Software Engineering. ACM, pp. 1059–1060 (2006)
7. Bell, R.: Introduction to IEC 61508. In: Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software-Volume 55, pp. 3–12. Australian Computer Society, Inc., Darlinghurst (2006)
8. Beizer, B.: Software Testing Techniques, 2nd edn. Van Nostrand Reinhold, New York (1990)
9. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: Future of Software Engineering, 2007. FOSE'07. IEEE, pp. 171–187 (2007)
10. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: a survey. IEEE Trans. Softw. Eng. **30**(5), 295–310 (2004)
11. Demathieu, S., Thomas, F., André, C., Gérard, S., Terrier, F.: First experiments using the UML profile for MARTE. In: 2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pp. 50–57 (2008)
12. Ali, S., Briand, L.C., Hemmati, H.: Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. Softw. Syst. Model. **11**(4), 633–670 (2012)
13. Iqbal, M., Ali, S., Yue, T., Briand, L.: Experiences of applying UML/MARTE on three industrial projects. In: Model Driven Engineering Languages and Systems, pp. 642–658. Springer, Berlin (2012)
14. David, A., Illum, J., Larsen, K.G., Skou, A.: Model-based framework for schedulability analysis using UPPAAL 4.1. Model-based Des. Embed. Syst. **1**(1), 93–119 (2009)
15. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Tools for Practical Software Verification, pp. 1–30. Springer, Berlin (2012)
16. Nejati, S., Adedjouma, M., Briand, L.C., Hellebaut, J., Begey, J., Clement, Y.: Minimizing CPU time shortage risks in integrated embedded software. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 529–539 (2013)
17. Briand, L.C., Labiche, Y., Shousha, M.: Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Genet. Program. Evolvable Mach. **7**(2), 145–170 (2006)
18. Di Alesio, S., Nejati, S., Briand, L., Gotlieb, A.: Stress testing of task deadlines: a constraint programming approach. In: IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), pp. 158–167 (2013)
19. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play-in/play-out approach. Software and Systems Modeling **2**(2), 82–107 (2003)
20. Di Alesio, S.: Optimal performance tuning in real-time systems using multi-objective constrained optimization. In: 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016) (2016)
21. Di Alesio, S., Nejati, S., Briand, L., Gotlieb, A.: Worst-case scheduling of software tasks—a constraint optimization model to support performance testing. In: Principles and Practice of Constraint Programming (CP 2014) (2014)
22. Sen, S., Moha, N., Baudry, B., Jézéquel, J.-M.: Meta-model pruning. In: Model Driven Engineering Languages and Systems, pp. 32–46. Springer, Berlin (2009)
23. Steel, J., Jézéquel, J.-M.: On model typing. Softw. Syst. Model. **6**(4), 401–413 (2007)
24. Sen, S., Moha, N., Mahé, V., Barais, O., Baudry, B., Jézéquel, J.-M.: Reusable model transformations. Softw. Syst. Model. **11**(1), 111–125 (2012)
25. Van Hentenryck, P.: The OPL optimization programming language. MIT Press, Cambridge (1999)
26. Di Alesio, S., Gotlieb, A., Nejati, S., Briand, L.: Testing deadline misses for real-time systems using constraint optimization techniques. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), pp. 764–769 (2012)
27. Nejati, S., Di Alesio, S., Sabetzadeh, M., Briand, L.: Modeling and analysis of CPU usage in safety-critical embedded systems to support stress testing. In: Model Driven Engineering Languages and Systems, pp. 759–775, Springer, Berlin (2012)
28. Di Alesio, S., Briand, L., Nejati, S., Gotlieb, A.: Combining genetic algorithms and constraint programming to support stress testing of task deadlines. ACM Trans Soft Eng Methodol. **25**, 4:1–4:37
29. Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications, vol. 24. Springer, Berlin (2011)
30. Shin, K.G., Ramanathan, P.: Real-time computing: a new discipline of computer science and engineering. Proc. IEEE **82**(1), 6–24 (1994)
31. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocess. Microprogram. **40**(2), 117–134 (1994)
32. Baker, T.P.: An analysis of fixed-priority schedulability on a multiprocessor. Real Time Syst. **32**(1–2), 49–71 (2006)
33. Mikučionis, M., Larsen, K.G., Rasmussen, J.I., Nielsen, B., Skou, A., Palm, S.U., Pedersen, J.S., Hougaard, P.: Schedulability analysis using UPPAAL: Herschel-Planck case study. In: Leveraging Applications of Formal Methods, Verification, and Validation, pp. 175–190. Springer, Berlin (2010)
34. Di Marco, V.C.A., Inverardi, P.: Model-Based Software Performance Analysis. Springer, Berlin (2011)
35. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Prentice-Hall Inc., Englewood Cliffs (1984)
36. Kartson, D., Balbo, G., Donatelli, S., Franceschinis, G., Conte, G.: Modelling with Generalized Stochastic Petri Nets. Wiley, New York (1994)
37. Plateau, B., Atif, K.: Stochastic automata network of modeling parallel systems. IEEE Trans. Softw. Eng. **17**(10), 1093–1108 (1991)
38. Petriu, D.C.: Software model-based performance analysis. In: Babau, J.P., Blay-Fornarino, M., Champeau, J., Robert, S., Sabetta, A. (eds.) Model Driven Engineering for distributed Real-Time

Systems: MARTE Modelling, Model Transformations and Their Usages. ISTE Ltd and Wiley, New York (2010)
39. Shousha, M., Briand, L., Labiche, Y.: A UML/SPT model analysis methodology for concurrent systems based on genetic algorithms. In: Model Driven Engineering Languages and Systems, pp. 475–489. Springer, Berlin (2008)
40. Mraidha, C., Tucci-Piergiovanni, S., Gerard, S.: Optimum: a marte-based methodology for schedulability analysis at early design stages. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011)
41. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 1990. LICS'90, pp. 414–425 (1990)
42. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Formal methods for the design of real-time systems, pp. 200–236. Springer, Berlin (2004)
43. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems, vol. 39. Springer, Berlin (2001)
44. Singh, A.: Identifying Malicious Code Through Reverse Engineering. Springer, Berlin (2009)
45. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al.: The worst-case execution-time problemoverview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. (TECS) **7**(3), 36 (2008)
46. Selic, B.: Using UML for modeling complex real-time systems. In: Mueller, F., Bestavros, A. (eds.) Languages, Compilers, and Tools for Embedded Systems, pp. 250–260. Springer, Berlin (1998)
47. OMG: UML profile for MARTE: modeling and analysis of real-time embedded systems. OMG, Technical Report OMG Document Number: formal/2011-06-02 (2011)
48. Sprunt, B., Sha, L., Lehoczky, J.: Aperiodic task scheduling for hard-real-time systems. Real Time Syst. **1**(1), 27–60 (1989)
49. Schmidt, D.C.: Model-driven engineering. IEEE Comput. Soc. **39**(2), 25 (2006)
50. Committee, I.S. et al.: 754-2008 IEEE standard for floating-point arithmetic. IEEE Computer Society Std (2008)
51. Bruce, K., Vanderwaart, J.: Semantics-driven language design-statically type-safe virtual types in object-oriented languages. Electron. Notes Theor. Comput. Sci. **20**(1), 1–26 (2004)
52. Hwang, C.-L., Masud, A.S.M.: Multiple objective decision making-methods and applications: a state-of-the-art survey. Lecture Notes in Economics and Mathematical Systems, vol. 164 (1979)
53. Pardalos, P., Migdalas, A., Pitsoulis, L.: Pareto Optimality, Game Theory and Equilibria, vol. 17. Springer, New York (2008)
54. Nejati, S., Briand, L.C.: Identifying optimal trade-offs between CPU time usage and temporal constraints using search. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 351–361. ACM (2014)
55. Atamtürk, A., Savelsbergh, M.W.: Integer-programming software systems. Ann. Oper. Res. **140**(1), 67–124 (2005)
56. Iqbal, M.Z., Ali, S., Yue, T., Briand, L.: Applying UML/MARTE on industrial projects: challenges, experiences, and guidelines. Softw. Syst. Model. **14**(4), 1367–1385 (2015)
57. Grossman, M., Aronson, J.E., McCarthy, R.V.: Does UML make the grade? Insights from the software development community. Inf. Softw. Technol. **47**(6), 383–397 (2005)
58. Espinoza, H., Richter, K., Gérard, S.: Evaluating MARTE in an industry-driven environment: TIMMO's challenges for AUTOSAR timing modeling. Modeling and Analysis of Real-Time and Embedded Systems with the MARTE UML Profile (2008)
59. Selic, B., Gérard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. Elsevier, Amsterdam (2013)
60. Safdar, S.A., Iqbal, M.Z., Khan, M.U.: Empirical evaluation of UML modeling tools—a controlled experiment. In: Taentzer, G., Bordeleau, F. (eds.) Modelling Foundations and Applications. Springer, Berlin, pp. 33–44 (2015)
61. Middleton, S.E., Servin, A., Zlatev, Z., Nasser, B., Papay, J., Boniface, M.: Experiences using the UML profile for MARTE to stochastically model post-production interactive applications. In: eChallenges, 2010, pp. 1–8 (2010)

Dr. Stefano Di Alesio is a Postdoctoral Fellow in the Software Engineering Department at Simula Research Laboratory. He received his Ph.D. (Computer Science, March 2015) from the University of Luxembourg. His research interests revolve around developing efficient, effective, and scalable methodologies to support verification and validation of large industrial software systems. While carrying out his research, Dr. Di Alesio built expertise in the areas of model-driven, search-based, and reverse software engineering. He has published papers on these topics on widely recognized conferences and journals, including MODELS, ISSRE, ACM TOSEM, SANER and ASE. Dr. Di Alesio has also been a reviewer of several acknowledged software engineering journals, such as RESS, SoSyM, and EMSE.

Dr. Sagar Sen is a research scientist at Simula Research Laboratory and visiting scientist at the Cancer Registry of Norway. Dr. Sen's interests are in verification and validation (V&V) of socio-technical systems. His research is based on industrial cases from the Norwegian Customs and the Cancer Registry of Norway. He has published scientific articles spanning subjects such as testing data-intensive systems, testing self-adaptive systems, domain-specific modelling, graph transformations, social/human computing, gamification, and applications of lightweight formal methods in software engineering. He supervises several masters students and a Ph.D. student. Dr. Sen co-founded Sweetzpot AS in 2015 to develop sensors and cyber-physical systems for endurance sports. He holds a M.Sc. (2006) in Computer Science from McGill University, Canada and a Ph.D. (2010) from Université de Rennes 1 while conducting research in INRIA, Rennes, in France. He has been a postdoc at INRIA Sophia-Antipolis and École des Mines, Nantes.