Enabling Performance Modeling for the Masses: Initial Experiences

Abel Gómez[1][0000−0003−1344−8472], Connie U. Smith[2], Amy Spellmann[2], and Jordi Cabot[1,3][0000−0003−2418−2489]

[1] Internet Interdisciplinary Institute (IN3)
Universitat Oberta de Catalunya (UOC), Spain
agomezlla@uoc.edu
[2] L&S Computer Technology, Inc, USA
{cusmith|amy}@spe-ed.com
[3] ICREA, Spain
jordi.cabot@icrea.cat

Abstract. Performance problems such as sluggish response time or low throughput are especially annoying, frustrating and noticeable to users. Fixing performance problems after they occur results in unplanned expenses and time. Our vision is an MDE-intensive software development paradigm for complex systems in which *software designers* can evaluate performance early in development, when the analysis can have the greatest impact. We seek to empower designers to do the analysis themselves by automating the creation of performance models out of standard design models. Such performance models can be automatically solved, providing results meaningful to them. In our vision, this automation can be enabled by using model-to-model transformations: First, designers create UML design models embellished with the *Modeling and Analysis of Real Time and Embedded systems* (MARTE) design specifications; and secondly, such models are transformed to automatically solvable performance models by using QVT. This paper reports on our first experiences when implementing these two initial activities.

Keywords: Experience, Performance Engineering, UML, MARTE, QVT

1 Introduction

Poor performance of cyber-physical systems (CPS) is exemplified by: (*i*) noticeably sluggish response time that becomes frustrating and unacceptable to users; (*ii*) low throughput that, in the worst case, cannot keep pace with the arrival and processing of new information; (*iii*) jitter such as flickering of displays, pixelation, irregular unpredictable responses, pauses while the system catches up, etc.; (*iv*) lack of response to user inputs because the system is busy with the previous request; or (*v*) timeouts and error messages.

Performance problems are obvious to users, and especially annoying and frustrating. Performance has become a competitive edge. The consequences of poor performance range from complaints or rejection of new products to a system failure that, in the worse case, may involve loss of life [18]. Social media and online product reviews expose these performance problems in a way not previously possible, so product failures are much more visible. Extreme cases raise the potential of a business failure.

However, more often than not, performance problems are tackled after they occur, resulting in unplanned expense and time for refactoring. Instead, we advocate for a Model-driven Engineering (MDE) [15] approach to CPS systems development in which stakeholders evaluate the performance of systems early in development when the analysis can have the greatest impact [34]. We seek to move system performance analysis from an isolated set of tools, that require experts to do laborious manual transfers of data among design and analysis tools, to an integrated framework in which independent tools share information automatically and seamlessly [33].

Our vision is a framework which takes advantage of ① ***Model Interchange Formats (MIF)***, which are a common representation for data required by performance modeling tools. The MIFs we use were originally proposed in 1995 [35,37] and have been broadened in scope over the years to incorporate performance-determining factors found in most performance modeling tools and techniques [19,32]. Using a MIF, tools in the framework may exchange models by implementing an import/export mechanism and need not be adapted to interact with every other tool in the framework. In fact, tools need not know of the existence of other tools thus facilitating the addition of new tools in the framework. This framework exploits model-to-model (M2M) transformations from design models to a MIF and thus provides an automated capability for analyzing the performance of CPS architectures and designs, enabling stakeholders to obtain decision support information – quickly and economically – during the early stages of development.

Our envisioned framework is ② ***design-driven*** rather than *measurement-driven*. *Measurement-driven* approaches use metrics of performance behavior and find ways to reduce resource usage to improve performance. On the contrary, our *design-driven* approach ties the performance metrics to the aspects of the design that cause excessive demands, so it is also possible to change the way the software implements functions. This often leads to more dramatic improvements than those achievable solely with measurement-driven approaches. The design-driven approach also leads to the resource-usage-reduction improvements so a combination of both types of improvements are attainable.

Finally, we envision an approach exploiting ③ ***design specifications*** as the source for the performance models, as opposed to *performance specifications* (see Section 4 for a description, and Section 8 for a comparison with similar previous approaches). Developing *performance modeling annotations* to designs – such as those in the *Modeling and Analysis of Real Time and Embedded Systems / Performance Analysis Model* (MARTE/PAM) [25] – requires expertise in performance engineering, and we seek to enable system designers to evaluate the performance of their designs without requiring performance-modeling experts. *System designers* – software architects, designers, modelers, developers, etc. – are *the masses* to whom our approach is targeted[a]. We do not envision eliminating performance specialists altogether: experts should be used when system designers find high performance risk or serious problems requiring performance expertise, when successful project completion is vital, and other high-profile concerns exist. This makes effective use of scarce performance-expertise resources.

This paper reports on our experience on implementing the first version of the design to MIF transformation for our MDE-based performance modeling framework supporting

[a] From this point on, we will use the generic term *system designers* to refer to any stakeholder taking advantage of our approach.

our vision to bring performance assessment closer to system designers. The starting point of this experience is the *Implementation of a Prototype UML to S-PMIF+ model-to-model transformation (UML to S-PMIF+)* project. This UML to S-PMIF+ transformation is a core element of our approach and key to study the feasibility of the approach. This *L&S Computer Technology* project, supported by the MDE experts at the *Universitat Oberta de Catalunya*, aims at implementing the transformation to generate performance models – that can be automatically verified – from design models. As the title of the project specifies, design models are specified using UML [27], which are enriched with design modeling stereotypes from the *Modeling and Analysis of Real-time Embedded Systems* (MARTE) [25] standard. These design models are transformed to the *Software Performance Model Interchange Format*+ (S-PMIF+) [32]. S-PMIF+ is an XML-based, MOF-compliant interchange format that can be fed into performance engineering analysis tools such as RTES/Analyzer [20].

The rest of the paper is structured as follows. Section 2 introduces a classical – i.e., not automated – performance analysis process, which serves as the basis to our automated proposal presented in Section 3. Section 4 presents our approach to model performance by using UML and MARTE, and Section 5 presents how such UML/MARTE models can be transformed to an automatically solvable performance model by using the QVT transformation language. Section 6 exemplifies how the concepts introduced in the two previous Sections are put in practice. Section 7 discusses our findings and lessons learned during this experience. Section 8 discusses related work and Section 9 presents our conclusions.

2 *Software Performance Engineering* in a Nutshell

Software Performance Engineering (SPE) [34] is a systematic, quantitative approach to the cost-effective development of software systems to meet performance requirements. Presented more than 25 years ago, it is a clear example of a classical and well established performance analysis process. The SPE process focuses on the system's use cases and the scenarios that describe them. From a development perspective, use cases and their scenarios provide a means of understanding and documenting the system's requirements, architecture, and design. From a performance perspective, use cases allow the identification of workloads that are significant from a performance point of view, that is, the collections of requests made by the users of the system. Traditionally, SPE processes have been conducted by performance analysts – assisted by software architects or developers – who use existing functional models of the system as their starting point. Fig. 1 describes typical steps in a simplified SPE process from the modeling point of view:

1. A performance analyst *identifies the critical use cases*, which are those that are important to the operation of the system, or to responsiveness as seen by the user.
2. The analyst *selects the key performance scenarios* – i.e. **UML Sequence Diagrams** in design models. The key performance scenarios are those that are executed frequently, or those that are critical to the perceived performance of the system.
3. The analyst *establishes the performance requirements*, i.e., identifies and defines the performance requirements – expressed in terms of response time, throughput, or resource usage – and workload intensities for each scenario selected in step 2.

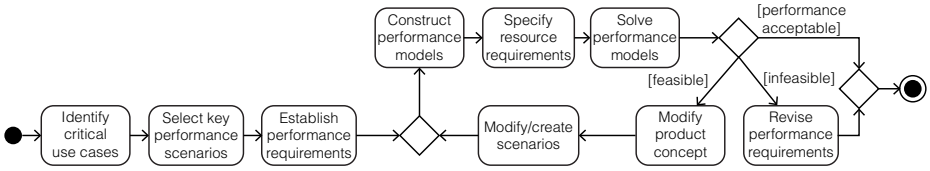


Fig. 1: SPE Process (adapted from [34])

4. The performance analyst *constructs the performance models* by translating the sequence diagrams of the key performance scenarios into execution graphs [34].
5. The analyst *specifies the resource requirements*, i.e., the amount of service that is required from key devices in the execution environment. This is typically done in two separate steps: first, by specifying the *software resource requirements*, i.e., the computational needs that are meaningful from a software perspective; and secondly, by mapping those software resource requirements onto the *computer resource requirements*. Software resource requirements can be extracted from the functional description of the system (e.g., from **UML Class Ciagrams**), while computer resource requirements depend on the environment in which the software executes (e.g., typically specified in **UML Deployment Diagrams** and other documentation).
6. Finally, the analyst *solves the performance models*. Solving the execution graph characterizes the resource requirements of the proposed software in isolation. If this solution indicates that there are no problems, the analyst proceeds to solve the system execution model. If the model solution indicates that there are problems, there are two alternatives: (*i*) *revise performance requirements*, modifying them to reflect this new reality – in this case, all stakeholders should decide if the new requirements are acceptable – or, (*ii*) *modify the product concept*, looking for feasible, cost-effective alternatives for satisfying this use case instance. This latter option may require modifying existing scenarios or creating new ones – again, involving other stakeholders such as software architects or developers.

These steps describe the SPE process for one phase of the development cycle, and the steps repeat throughout the development process. At each phase, the analyst refines the performance models based on increased knowledge of details in the design.

Despite many successes applying the SPE methods, there are key barriers to its widespread adoption and use: (*i*) it requires considerable experience and knowledge of performance modeling and there is a small pool of these experts; (*ii*) it is time consuming to manually develop performance models of a large, complex system; and (*iii*) a substantial amount of time is required to keep performance models in sync with evolving design models.

3 Towards Automated *Software Performance Engineering*

As mentioned above, a traditional SPE process is a labor-intensive approach requiring considerable expertise and effort: performance engineers work side by side with system designers to understand their design, and then create performance models of the design

using a performance modeling tool, such as SPE·ED [20]; when problems are detected, performance engineers recommend solutions to system designers who do a refactoring in order to improve performance. Clearly, automating the production of performance models would make early design assessment viable and enable system designers to conduct many of their own analyses without requiring extensive performance expertise.

Fortunately, as electronic systems have become more and more complex and software intensive [36], new engineering practices have been introduced to advance productivity and quality of these cyber-physical systems [16]. Model-Driven Engineering (MDE) [15] is a powerful development paradigm based on software models which enables automation, and promises many potential benefits such as increased productivity, portability, maintainability, and interoperability [10].

Although SPE relies on some design models, it does not exploit all their potential. Thus, our vision for SPE is a MDE-intensive software development paradigm based on MDA standards such as UML [27], MARTE [25], QVT [23] and MOF [24]. In this paradigm, automatic model transformation plays a key role in the development process, allowing system designers to evaluate the performance of systems early in development, when the analysis can have the greatest impact. Thus, we seek to empower system designers to do the analysis themselves by automating the creation of performance models, invoking the model solver, and getting the analysis results in a format meaningful to them. This quantifies the performance of system design options and identifies performance risks.

Achieving such empowerment, however, presents two important challenges:

C1 — We need to provide system designers with **model specifications** which allow them to express performance-determining design elements such as communication, constrained resources, etc. in their design models

C2 — We need to provide system designers with automatic tools able to **transform system design models** into analyzable performance models.

Thus, resolving these challenges accomplishes the objective of providing performance predictions without the performance expertise previously required.

Informally, our proposed renovated process can be seen as an evolution of SPE in which we introduce automation as shown in Fig. 2[b]. An important aspect to be noted (Fig. 2) with respect to a classical SPE process (Fig. 1) is that the main actors involved are now *system designers* as opposed to *performance engineers*. Our process consists of the following Activities:

1. As in traditional SPE, a performance assessment process starts by *identifying the critical use cases*.
2. System designers *define the key performance scenarios*. As opposed to Fig. 1, in which the key performance scenarios were selected by performance engineers from design models, here system designers use UML modeling tools to directly create them as we will describe later in Section 4.
3. System designers *define performance requirements* directly in the design models. This can be done by enriching the UML functional models with non-functional properties, using MARTE stereotypes, as we also describe in Section 4. Here,

[b] We have indicated with a gray background the activities that are different from those in Fig. 1

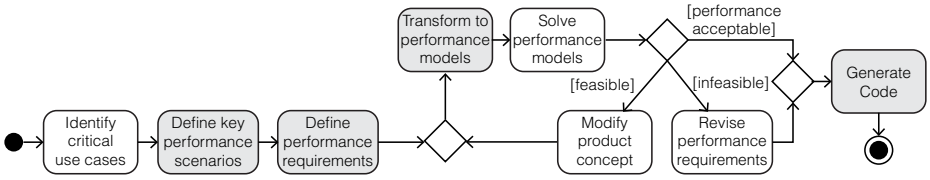


Fig. 2: An automated SPE Process

performance requirements are thus part of system models, and not a separate artifact as in traditional SPE.

4. Design models are *transformed to performance models*. As opposed to traditional SPE, where performance models were manually created by performance engineers, here performance models are automatically generated by executing a model-to-model transformation specified in QVT, as we outline in Section 5.
5. From here on, the process is similar to traditional SPE: performance models are solved, and after obtaining the analysis results three possibilities arise: (*i*) the results are acceptable; (*ii*) the results are unacceptable but the performance requirements are infeasible; and (*iii*) the results are unacceptable but the performance requirements are feasible. In the first and the second case the process continues as in traditional SPE. In the third case, system designers may modify the product concept (i.e., the models) and regenerate/reevaluate the performance models without intermediate steps.
6. Finally, as in any MDE process, system designers may automatically generate the application code for the system models. This latter step is out of the scope of this paper.

4 Defining Performance Scenarios and Requirements with UML/MARTE

In Section 2 we informally introduced some of the different UML diagrams that are useful from the SPE point of view. However, since traditional SPE design models do not need to be machine readable, no specific design rules are enforced in that approach. Our approach aims to achieve automation, and thus, it advocates for – and enforces – the use of four different UML diagrams to specify design models including performance characteristics.

An important aspect of UML is that customization is possible by using profiles. *Modeling and Analysis of Real-time Embedded Systems* (MARTE) [25] is an OMG standard defining foundations for model-based descriptions and analysis of real time and embedded systems. To facilitate its adoption, MARTE has been defined as a UML profile.

We advocate for the use of MARTE to include performance information and requirements in design models. Thus, system designers can make use of tools they are familiar with, without requiring performance engineers to manually create performance models. Although the use of MARTE stereotypes to enable the generation of performance models is not novel (see Section 8), the use of *design specifications* in favor of *performance modeling annotations* is. Thus, we propose the use of design modeling annotations – such as those from the *Generic Resource Modeling (GRM)*, *Software Resource Modeling*

(SRM), *Hardware Resource Modeling (HRM)* or *Allocation modeling (Alloc)* MARTE [25] subprofiles – as opposed to *performance modeling annotations* – such as those from the *Generic Quantitative Analysis Modeling (GQAM)*, *Performance Analysis Modeling (PAM)* or *Schedulability Analysis Modeling (SAM)* MARTE subprofiles.

Below we specify the UML diagrams to be used in our automated approach and their purpose, which are later exemplified in the case study in Section 6.

Structural View – Deployment Diagrams (DD) specify elements defining the execution architecture of systems. In our modeling approach, DDs specify hardware elements of the system, i.e., those capable of providing any kind of processing service.

Structural View – Class Diagrams (CD) specify the main logical entities participating in a system. In our modeling approach, CDs are used to define software elements of the system, as well as other communication and synchronization entities.

Examples of MARTE stereotypes that can be applied on class diagrams are those applicable to *Operations*, such as `MARTE::MARTE_Foundations::GRM::Acquire` and `MARTE::MARTE_Foundations::GRM::Release`. Such stereotypes can be used to specify that the stereotyped operations acquire or release a *mutex*, respectively.

Structural View – Composite Structure Diagrams (CSD) allow modeling the internal structure of a given *Classifier*. In our modeling approach CSDs are used to represent how the specific instances participating in a system – modeled as *Properties* – relate to each other from a static point of view. Such participants instantiate the classifiers representing either hardware or software elements (specified in a DD or a CD respectively). CSDs specify resources and their allocations for performance analysis. Typical stereotypes used in CSDs are (non exhaustive): `MARTE::MARTE_DesignModel::SRM::SW_Concurrency::SwSchedulableResource`, to annotate software elements generating a workload, and which execute concurrently with other software elements; `MARTE::MARTE_DesignModel::HRM::HwLogical::HwComputing::HwComputingResource`, to annotate active hardware execution resources such as CPUs or FPGAs; `MARTE::MARTE_Foundations::Alloc::Allocate`, typically applied on *Abstractions*[c] between software resources and hardware resources; or `MARTE::MARTE_Foundations::Alloc::Allocated`, typically applied to software and hardware elements related by an `Allocated` *Abstraction*.

Behavioral View – Sequence Diagrams (SD) allow describing precise inter-process communication by specifying execution traces. In our proposal, *Lifelines* in a SD represent elements declared in a CSD. SDs are the main means to specify *key performance scenarios* in our modeling approach. SDs typically also include fine grained resource usage information by using the `MARTE::MARTE_Foundations::GRM::ResourceUsage` stereotype. This stereotype may be applied to a *Message* or to an *ExecutionSpecification* to indicate that a given operation effectively requires the usage of the resource represented by the *Lifeline* – either receiving the *Message* or covered by the *ExecutionSpecification*, respectively – for a specific amount of time.

5 Automatic Transformation to Performance Models

With the aim of automating the transformation of software design models into performance models, we have implemented a transformation in a M2M transformation language. As Section 4 describes, the source models to be transformed are UML design models enriched with MARTE annotations.

We have chosen the *Software Performance Model Interchange Format+* (S-PMIF+) as the target representation for our performance models. S-PMIF+ is a Model Interchange Format (MIF) to exchange Real-Time and Embedded Systems (RTES) and Internet of Things (IoT) performance models among modeling tools proposed by Smith et al. [32]. S-PMIF+ is an extension of the S-PMIF, which is MOF-compliant since 2010 [22].

We have chosen MOF 2.0 Query/View/Transformation (QVT) [23], and specifically its *Operational* language (QVTo)[d], to encode the transformation rules between UML/MARTE and S-PMIF+. While a plethora of other existing transformation languages could have been chosen to implement this project, we chose QVTo for the following reasons:

Consistency — Almost all the languages in this work are OMG standards (UML [27], MARTE [25], MOF [24]). Using QVT allows us to stay inside the OMG stack.

[c] `Allocate` can only be applied to *Abstractions*, which are a specific kind of UML *Dependency*.

[d] In fact, the QVT specification defines three transformation languages: *Core*, *Operational* and *Relations*, being the main difference among them their declarative or imperative nature.

Table 1: High-level transformation mappings

SOURCE ELEMENT	MARTE STEREOTYPE	TARGET ELEMENT
`Property` represented by a `Lifeline`	`SwSchedulableResource` (`isActive=true`)	`PerformanceScenario`
`Property` represented by a `Lifeline`	`SwSchedulableResource`	`ExecutionGraph` (contained within the corresponding `Scenario`)
`Property` represented by a `Lifeline`	`TimingResource`	`PassiveEntity` (`type=timer`)
`ExecutionSpecification` whose covered `Lifeline` does not receive neither sync nor async `Messages`	—	`BasicNode`
`MessageOccurrenceSpecification`	`SwSchedulableResource` (applied on the `Property` represented by the `Lifeline` receiving the message); and `ResourceUsage` (applied on the `ExecutionSpecification` whose start event is the receive event of the current message)	`ActiveService`
`MessageOccurrenceSpecification` of a self-message	`SwSchedulableResource` (applied on the `Property` represented by the `Lifeline` receiving the message); and `ResourceUsage` (applied on the `Message` whose start event is the current `MessageOccurrenceSpecification`)	`ActiveService`
`MessageOccurrenceSpecification` whose corresponding `Message` is invoking a method called `'start'`	`TimingResource` (applied on the `Property` represented by the `Lifeline` receiving the message)	`PassiveService` (`command=start`)
`MessageOccurrenceSpecification` whose corresponding `Message` is invoking a method called `'stop'`	`TimingResource` (applied on the `Property` represented by the `Lifeline` receiving the message)	`PassiveService` (`command=stop`)
`Property` (receiving an `Abstraction`)	`Allocated` and `HwComputingResource` or `DeviceResource`. Additionally, the `Abstraction` pointing to the `Property` must have `Allocate`.	`Server`

Standardization — QVT has a normative document describing the semantics of the language, alleviating any future *vendor lock-in* problem.

Availability — Eclipse provides an interpreter of this language. Eclipse is the ideal platform to implement this transformation, since it provides (open source) tools to cover all the modeling steps of our proposed process.

Adequacy to the problem — The transformation from UML to S-PMIF+ involves sequence diagrams, where ordering is an important property. Managing ordering with declarative languages is hard, thus an imperative language such as QVTo provides a better control of the transformation logic (however, at the expense of abstraction).

Table 1 shows the subset of the transformation rules of the UML to S-PMIF+ transformation that are relevant for the case study presented in Section 6. The first column indicates the UML elements (see [27]) involved in the rule; the second column the MARTE stereotypes (see [25]) that have to be applied so that the rule matches; and the third column indicates the S-PMIF+ element (see [22, 32] for a full reference) that should be generated.

The UML to S-PMIF+ transformation follows a top-down approach. Starting from the UML top-level element – i.e., the *Interaction* corresponding to the SD – traverses the containment tree processing the contained elements. In this navigation, one of the most rel-

Listing 1: Excerpt of the UML/MARTE to S-PMIF+ QVTo transformation

```
mapping UML::ExecutionSpecification::executionSpecification2Node() : SPMIF::ProcessingNode
disjuncts UML::ExecutionSpecification::executionSpecification2BasicNode,
  UML::ExecutionSpecification::executionSpecification2ReplyNode,
  UML::ExecutionSpecification::executionSpecification2NoReplyNode;

abstract mapping UML::ExecutionSpecification::executionSpecification2abstractNode() : SPMIF::Node {
  var index : Index = new Index();
  self.events()->forEach(s) {
 serviceReq += s[UML::ExecutionSpecification]
                .map executionSpecification2ServiceSpec(index);
 serviceReq += s[UML::MessageOccurrenceSpecification]
                .map messageOccurrenceSpecification2PassiveService(index);
 serviceReq += s[UML::MessageOccurrenceSpecification]
                .map messageOccurrenceSpecification2ActiveService(index);
  }
}

mapping UML::ExecutionSpecification::executionSpecification2BasicNode() : SPMIF::BasicNode
inherits UML::ExecutionSpecification::executionSpecification2abstractNode
when { -- Generate Basic Node when the Lifeline does not receive neither sync nor async messages
  self.events()[UML::MessageOccurrenceSpecification].message[ --> Select messages that:
 receiveEvent.covered() = self.covered() --> Are received by this Lifeline
   and receiveEvent.covered() <> sendEvent.covered() --> Are not self-messages
   and (messageSort = UML::MessageSort::synchCall
        or messageSort = UML::MessageSort::asynchCall) --> Are sync or async messages
  ]->isEmpty()
}{
  name := self.name;
}

helper UML::ExecutionSpecification::events() : OrderedSet(UML::InteractionFragment) {
  var start : Integer = self.covered().events()->indexOf(self.start);
  var finish : Integer = self.covered().events()->indexOf(self.finish);
  assert fatal (start < finish)
    with log ('Malformed input model in ExecutionSpecification "{1}": its "start" event ({2})
          appears after its finish ent ({3}).'._format(self, self.start, self.finish));
  return self.covered().events()->subOrderedSet(start, finish);
}
```

evant properties of *Interaction* is *fragment*, which contains – in the order they occur – all the events happening in the *Interaction*. Simplifying, once an interesting event – i.e., an event that should be transformed – is found, the corresponding transformation rule is applied.

Listing 1 shows the *QVT mappings* implementing the rule specified in the fourth row of Table 1. Rule `executionSpecification2Node` (lines 1–4) is a *mapping* that is called when an *ExecutionSpecification* contained within an *Interaction* is found. This mapping is indeed a *disjunction* of three other mappings: `executionSpecification2BasicNode` (lines 18–29), `executionSpecification2ReplyNode` (not shown) and `executionSpecification2NoReplyNode` (not shown). A disjunction indicates that only the first *mapping* whose *when* clause holds will be executed. As it can be observed, `executionSpecification2BasicNode` inherits from the *abstract mapping* `executionSpecification2abstractNode` (lines 6-16). This abstract mapping cannot be executed by itself (in fact, `SPMIF::Node` is an abstract class, which prevents its execution), but can specify transformation actions that can be reused and extended by other mappings (such as the `executionSpecification2Node` disjoint mappings). In this case, the *abstract mapping* is executed before the instruction in line 28, and triggers the execution of the mappings between lines 9–14 for the events returned by the helper `events()`. This helper is declared in the context of `ExecutionSpecification` so that it can be used as shown in line 8. It returns the list of events that occurr in the *Lifeline* covered by the *ExecutionSpecification* while it is active. As it can be observed, we rely on the order of the events to determine whether an event occurrs during the execution. Lines 34–36 show an interesting feature of QVTo: the possibility to specify assertions. This is a specially useful feature as we will discuss in Section 7. Finally, the *when* clause between lines 20–27 specifies that the mapping will only be executed when the Lifeline covered by the *ExecutionSpecification* does not receive neither synchronous nor asynchronous messages while the *ExecutionSpecification* is active.

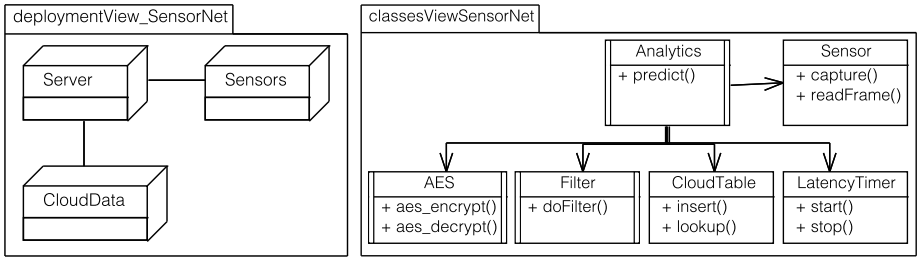
This Listing is only a small demonstration of what our M2M transformation – of nearly 2000 lines of code (LOC) – looks like. In Section 7 we provide more information about its characteristics and numbers.

6 An Illustrative Case Study: Cyber Physical Systems Analysis

We illustrate our approach by analyzing an existing *data acquisition system* (*SensorNet*) and predicting its performance when encryption is added. Encryption is critical to ensure that data is securely transferred from *servers* to a *data store* in the cloud. We chose this case study to show how both security and performance can be analyzed before implementation.

Our *SensoreNet* case study involves both hardware and software elements as shown in Figure 3. Figure 3a shows the DD with the processors used in execution: *Servers* are hardware elements, with computing and communication capabilities, that read information from simple hardware *Sensors* – 2700 in our case study – and send this information via a communication media to the cloud (represented by *CloudData*). Figure 3b depicts the software elements in a CD: *Analytics* reads information from a *Sensor*[e], later processes it by using the *Advanced Encryption Standard (AES)* and *Filter*

[e] *Sensor* here represents the software element used to access hardware *Sensors*.



(a) (b)

Fig. 3: Deployment Diagram (a) and Class Diagram (b)

software artifacts; and finally sends it to a *CloudTable*. Additionally, *Analytics* makes use of a *LatencyTimer*, which tracks the beginning and the end of this process.

Figure 4 shows the actual instances of these hardware and software elements of our *SensorNet* case study in a CSD: *cloudData*, *server* and *sensors* are instances of the *Nodes* specified in Figure 3a; while *filter*, *aes*, *analytics*, *sensor* and *latencyTimer* are instances of the *Classes* specified in Figure 3b. As it can be observed, we used MARTE stereotypes to specify additional data that is needed to build the performance model[f]: `SwSchedulableResource` specifies workload in *analytics* by using the VSL [25] expression `closed(population=10, extDelay=(500,ms))`, i.e., 10 requests in an interval of 500ms; `HwComputingResource` designates the processors for the *Servers*, i.e., 80 instances; `DeviceResource` represents a server that does not model contention delays (a so-called *delay server* in the performance model); and the `TimingResource` designates the *latency timer*. The *Allocate* shows how processes are allocated to the

[f] We obtained processing times and data/network transfer bytes specified in Figures 4 and 5 from the analysis of benchmark data.

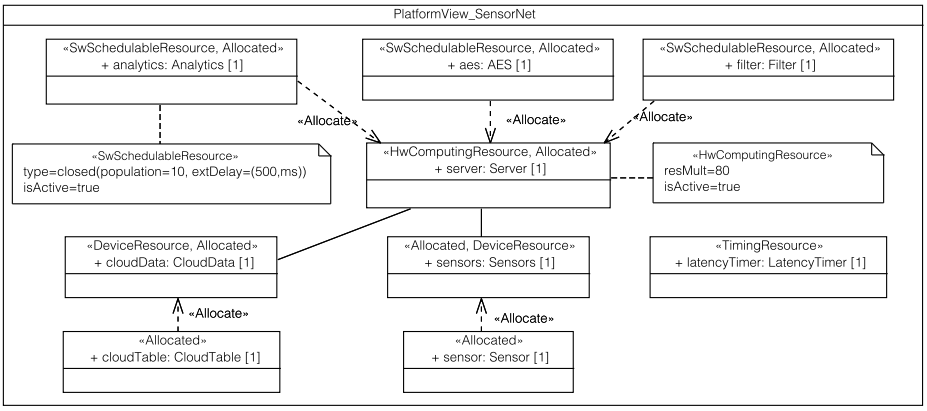


Fig. 4: Composite Structure Diagram

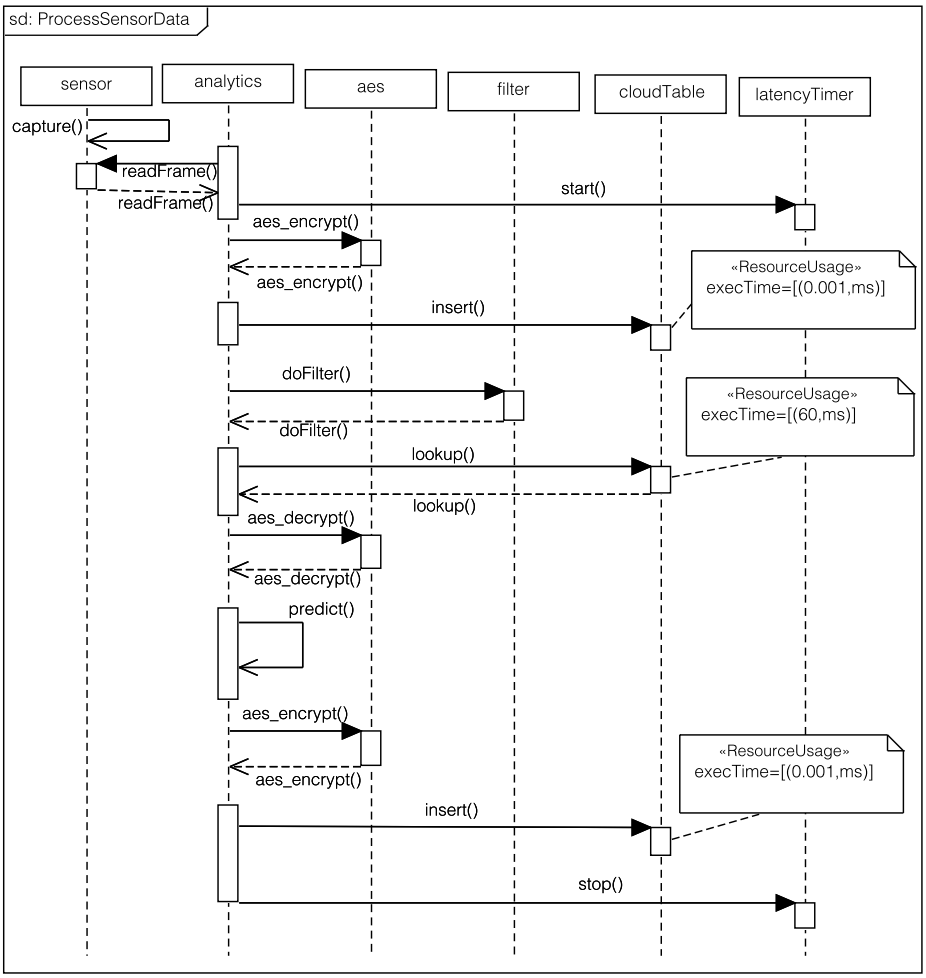


Fig. 5: Sequence Diagram

processors: *cloudTable* is hosted on *cloudData*; *filter*, *aes* and *analytics* tasks are executed on a *server*; and the software representation of *sensors* lie on hardware *sensors*.

Finally, we modeled two scenarios: the first adds security/encryption using basic *sensors*, where the *encryption* and *filtering* happen on the servers; and the second evaluates replacing the *basic sensors* with *smart sensors*, capable of doing the encryption on the sensor itself. In both cases, we use the *CloudTable* database for storing data.

The sequence diagram for the first scenario is shown in Figure 5: *analytics* reads a frame of captured data from a specific basic *sensor*, starts the *latencyTimer*, *encrypts*[g] the frame, and *inserts* it into the *cloudTable*. Then, it *filters* the data, does a *lookup*

[g] We based the encryption and decryption on an open source version of the Advanced Encryption Standard (AES) [8].

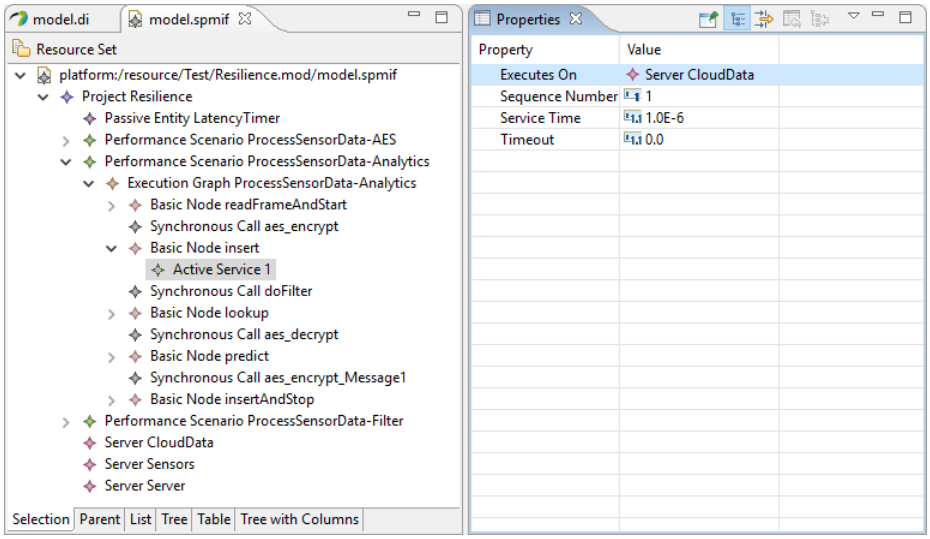


Fig. 6: Generated S-PMIF+ model

from the *cloudTable* to get recent activity discovered by the sensor, *decrypts* it, and makes *predictions* of future behavior. Results are finally *encrypted* and *inserted* into the *cloudTable*, and the *latencyTimer* is stopped. The figure shows the MARTE annotations for the execution time required for some – not all for readability purposes – steps.

We do not show the sequence diagrams for the second scenario (i.e., using the smart sensors) for the sake of conciseness. In summary, this second sequence diagram lacks the encryption and filter steps, and has a lower value in the specification for the data rate. All the other structure diagrams remain unchanged.

Once the scenarios are modeled, our prototype is able to transform them to the corresponding S-PMIF+ specifications by applying the rules introduced in Section 5. Figure 6 shows the resulting S-PMIF+ model for the first scenario. A *Performance Scenario* – with its corresponding *Execution Graph* – is generated for the *analytics* property, which was stereotyped as `SwSchedulableResource`. Additionally, a *Basic Node* is generated from the *ExecutionSpecification* sending the *insert* message to *cloudTable*. This message, in turn, generates an *Active Service* which executes on the *CloudData Server* with a *service time* of $1.0 \cdot 10^{-6}$ (seconds). All the other elements are generated according to the transformation rules listed in Table 1.

The S-PMIF+ models are sent to the RTES/Analyzer solver. RTES/Analyzer is the tool allowing the developer to study the performance of the modeled system with different parameter settings for the *data rate*, *number of processors*, *time for encryption*, and *time for CloudTable processing*.

From our experiments using RTES/Analyzer, we obtained that the first scenario using the *basic sensors* requires 80 CPUs to meet the performance requirement (for the 2700 sensors of the case study); while the second scenario using *smart sensors* requires only 50 CPUs. Additional valuable information from the RTES/Analyzer model shows that

we need more processors in the cloud to speed up the *insert* and *lookup* tasks. For the case study, we used a single instance in the cloud. There are many other options for both platforms and designs that can be explored with the model, such as: (*i*) reducing the time required for encryption by tuning the algorithm to the application; (*ii*) using asynchronous *cloudTable* inserts; (*iii*) using a pipeline architecture; or (*iv*) using cloud vs. on-premises storage. In any case, the evaluation process is the same: the design model is revised, transformed and solved.

7 Discussion

This section reports on the some of the lessons learned during the realization of this work, mainly linked to the realization of the technology transfer project between *L&S Computer Technology* and the *Universitat Oberta de Catalunya*.

While the specific goal of the project was to "simply" write a transformation between UML/MARTE and S-PMIF+ (i.e., implementing Activity 4 of Section 3), we quickly realized that clearly defining the inputs and outputs of such a transformation indeed impacted the whole process. This led us to redraw the initial scope of the transformation, having a wider vision of the project, and coming out with a set of modeling guidelines (which in turn support Activities 2–3 in Section 3) that, together with the transformation itself, make up the core of the framework. A transformation project is, in the end, a software development project (where *the software* is the transformation) and, as such, it is not without similar challenges.

In the following, we provide some facts about this project, and reflect about the decisions taken and the experience we gained. We believe this could be useful to other teams developing projects involving industry-level transformations. This is the first take-away for anybody starting a transformation project.

Project size and effort — The project lasted for 2 months and was lead by two main technical contacts, one with nearly 30 years of experience in performance engineering, and the other with more than 13 years of experience in MDE and OMG standards.

The set of conceptual correspondences between UML/MARTE and S-PMIF were identified in a several-months previous study, and were provided in an Excel sheet at the start of the project. Including attributes, the spreadsheets documented up to 200 correspondences, including 40 MARTE stereotypes with their corresponding S-PMIF+ counterparts.

To complete the transformation code itself, the project required 8 meetings and over 150 emails exchanged; and the final deliverable included a 118-pages report.

Barrier to entry: the modeling languages — While UML and MARTE indeed allow stakeholders to provide the design specifications without having to learn complex performance modeling languages, there is still a lot to do to lower the barrier to entry.

Especially regarding MARTE, although there exists a reference book [31], there are very few online documents providing systematic modeling guidelines and we had to rely on online tutorials [9, 21] to determine the right recommendations for users of our approach. We based our specifications on the design methods of Selic et al. [31] because they are a big step forward on *how* to specify typical design characteristics, particularly

those that impact performance such as communication, synchronization, etc. This work provides performance feedback on the desirability of design options. This and other work that provides design-assessment feedback makes UML/MARTE more attractive going forward. If another, more promising MDE design language emerges it should be straightforward to adapt our approach to transform it to our MIFs to provide performance predictions.

As a consequence of the scarce documentation available, 34 pages out of the 118 of the report mentioned above were dedicated to explain our recommended use of UML and MARTE to support Activities 2–3 of our approach. This is necessary to resolve some of the language ambiguities (e.g., regarding the specification of VSL expressions[h]).

Our thoughts on using QVTo — Although imperative transformation languages do not have an especially good reputation, in this special case QVTo was a very good choice for our project thanks to the following features of the language:

- Its imperative character facilitated the processing of ordered elements (required for the transformation of sequence diagrams) in a very natural way.
- Its logging facilities and support for assertions are specially useful to control ill-formed models produced by the tools (more on this below).
- It has explicit support to organize transformations in libraries which helps when developing complex transformations and facilitates reusability.
- Helpers can be used to add new operations to meta-elements dynamically, without changing the metamodels (similar to the concept of extension functions in Kotlin [14] and other languages). Again, this simplifies the writing of complex transformations.
- QVTo allows the definition of intermediate classes, which only live within the transformation execution scope. This is very useful to reify VSL expressions – Strings – in their corresponding in-memory complex datatypes (the so-called *NFP types* [25]).

We have been pleasantly surprised with QVTo, especially after previous bad experiences with its declarative counterpart. QVTo is definitely an option to be considered when choosing the transformation language for your project, particularly if you require some of the complex requirements above.

Repetitive Transformation code — The transformation was spread out in 4 files for a total of 2027 lines of code (LOC) excluding empty lines. These LOC were distributed as follows:

- 243 LOC dedicated to check the presence/absence of MARTE stereotypes (58 helpers were written to deal with MARTE stereotypes);
- 272 LOC (in 30 helpers) devoted to string manipulations;
- 448 LOC to deal with VSL expressions and NFP types (21 helpers to deal with them);
- 305 LOC in UML helpers (47 helpers to deal with UML elements);
- 759 LOC for the actual implementation of the transformation mappings

As you see, more than 60% of the transformation code dealt with auxiliary tasks. This must be taken into account when estimating the effort required to implement

[h] See `http://issues.omg.org/issues/MARTE12-4`

transformations. Too often we based that estimation on the analysis of the mappings forgetting that this will be only a small part of the total LOC.

Nevertheless, this repetitive code could be simplified by importing external libraries (a clear example would be a QVTo library for String manipulation). These ready-made libraries do not exist at this time, but we believe it is in the best interest of the community to develop and share them.

Limitations of the modeling tools — Within the Eclipse ecosystem, Papyrus is the most popular tool for UML modeling. Still, it also has known limitations when it comes to SDs and this had a negative impact on our project. Ordering of events is crucial in SDs (see Section 5), however Papyrus is not always able to maintain it correctly in the underlying model as soon as the user moves messages around. Papyrus models get corrupt very easily, and *ExecutionSpecifications* – among other primitives – lose their start and finish events easily. Garbage elements are also commonly left around.

Limitations of this approach — The design specifications follow the methods in [31] for specifying communication, synchronization and other coordination, resource constraints, etc. using the rules in Section 5. These guidelines must be followed for the resulting performance model to represent the intended behavior of the system. Likewise, the performance models only contain features that are expressed in the design models; developers should be aware that early predictions tend to be optimistic, and only represent details that have been specified in the design models. This follows the SPE method of adding features as the software evolves: early models may not represent all aspects of performance (best-case models); details are added as the software specifications evolve to get a more precise prediction of performance.

All the previous facts and issues, beyond delaying the project, also forced us to write additional *sanity check code* to ensure the correctness of the input models before actually transforming them.

On the positive side, the interpreter of QVTo provided all the expected facilities of a modern IDE: content-assist, line-by-line debugging, and watch expressions, which helped us in detecting the above issues.

8 Related work

The assessment of non-functional requirements, such as performance, of software systems is a well-established discipline in the software engineering field [1, 4, 7, 34]; however, different formalisms, techniques, and levels of automation have been achieved.

Other design-based approaches can also be found in the literature. *Performance by Unified Model Analysis* (PUMA) is a framework for transforming data from a UML-based software design to performance tools [28, 39]. It uses the *Core Scenario Model* (CSM) as the intermediate representation. CSM was originally based on the UML profile for *Schedulability, Performance, and Time* (SPTP) [26] and later adapted to MARTE/PAM both of which closely correspond to the information requirements of the *Layered Queueing Model* (LQN) tool. This simplifies the M2M transformations, but because the MARTE/PAM input specifications so closely resemble the performance model itself,

it requires performance expertise to create those specifications. Our work uses MIFs that were originally proposed for a model interchange paradigm in 1995 [35, 37]. They have been updated and generalized [19, 32] to include performance modeling features found in a variety of performance modeling tools and techniques that have proven to be useful over the years, including those in LQN. Another key difference is that we do not require the performance-specific annotations in MARTE/PAM; we use the MARTE design specifications provided by developers instead. Nevertheless, these approaches are similar in concept, and useful insights on the challenges of developing transformations are also described in [38].

Palladio [2] is an example that also uses MDE techniques. Its simulation tool is implemented using the same technologies as the prototype presented in this work (e.g., Eclipse, Eclipse Modeling Framework, etc.). Unlike our proposal, Palladio provides a domain specific modeling language, the so-called *Palladio Component Model* (PCM), to specify component-based software architectures. Nevertheless, it is worth mentioning that PCM resembles UML in some parts (e.g., component, activity and deployment diagrams).

Kounev et al. [17] propose a model-based approach to designing self-aware IT systems using the *Descartes Modeling Language* (DML). DML is a domain-specific architecture-level language that allows specifying adaptation points to reconfigure the application architecture at runtime. The Descartes approach is fully automated, it is also based on Eclipse, and enables on-line performance predictions and model-based adaptation. DML has been applied to several industrial case studies [13].

These and other approaches differ in that they transform to one specific tool rather than to a MIF. E.g., both PCM and DML transform to Queueing Petri Nets (QPN) to solve their models using the QPME tool; while our prototype transforms our UML/MARTE models to S-PMIF+, which serves as a pivot language for different formalisms and tools.

On the other hand, these tools still require an expert in the use of that performance analysis tool: e.g., Palladio and DML require learning a new performance model specification language. While this is not a problem for performance modeling experts, it is a barrier to system developers who wish to evaluate their own design with minimal extra work. It is also noteworthy that the contents of these meta-models (PCM and DML) were considered and incorporated when possible in the development of the MIFs used in our approach.

The DICE framework [6] is an MDE-based solution using UML specifically designed to focus on the specific challenges of quality-assurance for data-intensive applications using big data technologies. Its DICE Simulation component [3] is also built using Eclipse Papyrus, and is able to transform annotated UML models to both performance and reliability models to stochastic Petri nets using QVTo. The main difference with respect to the work presented here is that, in order to fully support the specificities of data-intensive applications, DICE provides its own profile – the so-called *DICE Profile*. This profile provides performance modeling annotations – as opposed to the design specifications of our approach – which extend and reuse constructs from the GQAM, PAM and SAM MARTE subprofiles, as well as from the DAM [4] profile.

Process mining techniques are a clear example of measurement-based approaches (as described earlier) and several tools are available (e.g., [5, 11, 12, 29, 30]). These approaches try to bridge the gap between the resulting performance metrics and the design itself, however, this is still a challenging task requiring significant expertise.

Our approach is design-based, and uses M2M transformations to bridge such a gap by automatically generating performance models from UML diagrams, which are compliant with the standard OMG MARTE [25] profile.

9 Conclusions

This experience has proved the viability of automating SPE processes based on MDE techniques and MIFs. The heart of this automated approach, the transformation from UML/MARTE, shows that a renovated SPE process can be based on the models produced by system designers without requiring extensive knowledge and experience in performance engineering. By automating the transformation of software designs to performance models, we eliminate the need for laborious and error-prone manual translation of software design information into performance models, and the effort in keeping the design and performance models in sync throughout development and operation. The results are also presented in a format that can be easily evaluated by system designers. Automation and usability are key if system designers are to use the technology.

The prototypes we created demonstrated that the end-to-end process is clearly viable even if we learned a few hard lessons along the way. We developed screens that make the transformation of designs to performance models, automated solution of experiments, and the conversion of tool output into a results format that is easy to comprehend, highlights potential problems, allows evaluation of tradeoff in design parameters, and allows user customization of results and formats.

The focus of this effort was on performance analysis of CPS systems; however, as further work, we plan to *plug in* other tools to support additional types of design analysis, such as safety, reliability/availability, fault tolerance and others.

References

1. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Trans. Softw. Eng. 30(5), 295–310 (May 2004)
2. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for Model-driven Performance Prediction. J. Syst. Softw. 82(1), 3–22 (Jan 2009)
3. Bernardi, S., Domínguez, J.L., Gómez, A., Joubert, C., Merseguer, J., Perez-Palacin, D., Requeno, J.I., Romeu, A.: A systematic approach for performance assessment using process mining. Empirical Software Engineering (Mar 2018), DOI: 10.1007/s10664-018-9606-9
4. Bernardi, S., Merseguer, J., Petriu, D.C.: Dependability Modeling and Analysis of Software Systems Specified with UML. ACM Comput. Surv. 45(1), 1–48 (Dec 2012)
5. Celonis PI (2011), URL: https://www.celonis.com, accessed June 2018
6. Consortium, D.: Getting Started with DICE: Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements (2018), URL: http://www.dice-h2020.eu/getting-started/, accessed June 2018
7. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis. Springer Publishing Company, Incorporated, 1st edn. (2011)
8. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)

9. Demathieu, S.: MARTE Tutorial: An OMG UML profile to develop Real-Time and Embedded systems, http://www.uml-sysml.org/documentation/marte-tutorial-713-ko/at_download/file, accessed June 2018
10. Di Ruscio, D., Paige, R.F., Pierantonio, A.: Guest editorial to the special issue on success stories in model driven engineering. Sci. Comput. Program. 89(PB), 69–70 (Sep 2014), DOI: 10.1016/j.scico.2013.12.006
11. Diwan, A., Hauswirth, M., Mytkowicz, T., Sweeney, P.F.: TraceAnalyzer: A system for processing performance traces. Software: Practice and Experience 41(3), 267–282 (2011)
12. Günther, C.W., Rozinat, A.: Disco: Discover Your Processes. BPM (Demos) 940, 40–44 (2012)
13. Huber, N., Brosig, F., Spinner, S., Kounev, S., Bähr, M.: Model-Based Self-Aware Performance and Resource Management Using the Descartes Modeling Language. IEEE Transactions on Software Engineering 43(5), 432–452 (May 2017)
14. JetBrains: Extensions – Kotlin Programming Language, https://kotlinlang.org/docs/reference/extensions.html, accessed June 2018
15. Kent, S.: Model driven engineering. In: Proceedings of the Third International Conference on Integrated Formal Methods. pp. 286–298. IFM '02, Springer-Verlag, London, UK, UK (2002)
16. Khaitan, S.K., McCalley, J.D.: Design Techniques and Applications of Cyberphysical Systems: A Survey. IEEE Systems Journal 9(2), 350–365 (2015)
17. Kounev, S., Huber, N., Brosig, F., Zhu, X.: A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. IEEE Computer 49(7), 53–61 (July 2016), DOI: 10.1109/MC.2016.198
18. Leveson, N.G.: Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology. Addison-Wesley (1995)
19. Lladó, C.M., Smith, C.U.: Pmif+: Extensions to broaden the scope of supported models. In: Balsamo, M.S., Knottenbelt, W.J., Marin, A. (eds.) Computer Performance Engineering. pp. 134–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), DOI: 10.1007/978-3-642-40725-3_11
20. L&S Computer Technology, Inc.: SPE·ED+, http://spe-ed.com/, accessed June 2018
21. Medina, J.: The UML Profile for MARTE: modelling predictable real-time systems with UML, http://www.artist-embedded.org/docs/Events/2011/Models_for_SA/01-MARTE-SAM-Julio_Medina.pdf, accessed June 2018
22. Moreno, G.A., Smith, C.U.: Performance analysis of real-time component architectures: An enhanced model interchange approach. Performance Evaluation 67(8), 612 – 633 (2010), special Issue on Software and Performance
23. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3, http://www.omg.org/spec/QVT/1.3/
24. OMG: Meta Object Facility (MOF), Ver. 2.5.1, http://www.omg.org/spec/MOF/2.5.1/
25. OMG: Modeling and Analysis of Real-time Embedded Systems (MARTE), Ver. 1.1, http://www.omg.org/spec/MARTE/1.1/
26. OMG: UML Profile for Schedulability, Performance, & Time (SPTP), Ver. 1.1, http://www.omg.org/spec/SPTP/1.1/
27. OMG: Unified Modeling Language (UML), Ver. 2.5, http://www.omg.org/spec/UML/2.5/
28. Petriu, D.B., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from uml designs. Software & Systems Modeling 6(2), 163–184 (Jun 2007), DOI: 10.1007/s10270-006-0026-8
29. ProM Tools (2017), URL: http://www.promtools.org/doku.php, accessed June 2018
30. QPR Process Analyzer (2011), URL: https://www.qpr.com, accessed June 2018
31. Selic, B., Gérard, S.: Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2013)

32. Smith, C.U., Lladó, C.M.: SPE for the Internet of Things and Other Real-Time Embedded Systems. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. pp. 227–232. ACM, New York, USA (2017), DOI: 10.1145/3053600.3053652
33. Smith, C.U., Lladó, C.M., Puigjaner, R.: Model interchange format specifications for experiments, output and results. The Computer Journal 54(5), 674–690 (2011), DOI: 10.1093/comjnl/bxq065
34. Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison Wesley Longman Publishing Co., Inc. (2002)
35. Smith, C., Williams, L.: A performance model interchange format. Journal of Systems and Software 49(1), 63 – 80 (1999), DOI: 10.1016/S0164-1212(99)00067-9
36. Wallin, P., Johnsson, S., Axelsson, J.: Issues Related to Development of E/E Product Line Architectures in Heavy Vehicles. In: 42nd Hawaii Int. Conf. on System Sciences (2009)
37. Williams, L.G., Smith, C.U.: Information requirements for software performance engineering. In: Beilner, H., Bause, F. (eds.) Quantitative Evaluation of Computing and Communication Systems. pp. 86–101. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
38. Woodside, M., Petriu, D.C., Merseguer, J., Petriu, D.B., Alhaj, M.: Transformation challenges: from software models to performance models. Software & Systems Modeling 13(4), 1529–1552 (Oct 2014), DOI: 10.1007/s10270-013-0385-x
39. Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (puma). In: Proceedings of the 5th International Workshop on Software and Performance. pp. 1–12. WOSP '05, ACM, New York, NY, USA (2005), DOI: 10.1145/1071021.1071022