Early Model-Driven Timing Validation of IoT-Compliant Use Cases

Padma Iyenghar, Arne Noyer, Elke Pulvermueller
Software Engineering Research Group, University of Osnabrueck, Germany
{piyengha | anoyer | elke.pulvermueller}@uos.de

***Abstract*—Several IoT platforms, with support for a rich set of device libraries, facilitate rapid development of embedded IoT applications. But, none of the approaches deal with the analysis of performance characteristics. An example is the early timing validation of IoT use cases in specialized timing analysis tools.**

Addressing this gap, a generic workflow for a quick, early model-driven, system-level timing validation of IoT compliant hand-written code in specialized timing analysis tools, such as SymTA/S, is proposed in this paper. A prototype implementation of the workflow and its experimental evaluation in a novel, IoT-compliant real-time emission monitoring use case is presented. The main benefit of the proposed workflow is the early feedback regarding performance characteristics of the embedded IoT application, under varying timing constraints.

***Index Terms*—Embedded IoT application; Reverse Engineering; Early timing validation; Unified Modeling Language (UML);**

I. Introduction

In the recent years one can witness an exponential growth of a myriad of smart physical objects equipped with sensing, computing and communication capabilities which may connect together and to a network, to achieve a common goal (e.g. a *smart* system). These aspects may be collectively regarded as the emerging paradigm of the Internet of Things (IoT). The IoT applications are gaining inroads in variegated domains, for instance, from real-time critical aerospace and automotive to insurance and supply chain environment. Thus, the application platforms for IoT-enabled use cases may be coarsely classified into enterprise and embedded software development. The above grouping is primarily based on the distinctive requirements for embedded application development such as the requisiteness to meet critical real-time constraints and limited computing resources (e.g. memory).

In general, IoT applications use sensors and actuators embedded in the (IoT-enabled) environment. They collect volumes of data, such as distance, temperature, emission and energy consumption, to name a few. The primary goal of the application is to avoid operational failures that may have a tangible impact on the environment. For instance, an integrated future vehicle may comprise of a swarm of sensors collectively interacting as IoT-enabled applications across several Electronic Control Units (ECUs), with critical hard and soft real-time constraints, networked via gateways. In such cases, it becomes imperative to address not only the functional properties, but also satisfy the Non-Functional Properties (NFPs) such as the timing requirements of the underlying IoT use case, during application development.

Motivation: Application development for IoT use cases often employ *mashups*, which permit visual, interactive modeling of the message flow between devices. On the other hand, model-driven approaches which are also being employed for IoT application development, in general, permit expressive modeling of the systems, possibly with code generation. However, a majority of IoT applications for embedded devices, often employ hand-written code in programming languages such as C/C++ (e.g. ARM-*mbed* environment [3])

Despite being an emerging field, none of the existing approaches for IoT application development deal with the analysis of performance characteristics of the IoT-enabled software system. For instance, existing approaches do not address the aspect of early timing validation of IoT based use cases, in specialized timing validation tools such as SymTA/S [13] and Timing Architects [16].

On the other hand, reverse engineering the hand-written IoT-application software system (e.g. in C++), can yield a design model at a higher abstraction level, for instance as Unified Modeling Language (UML) diagrams. A quick, system-level timing validation of this design model in a specialized timing validation tool such as SymTA/S, would provide valuable feedback to the developer regarding the actual system behavior, early in the development stages of the IoT use case. It is intuitive to perceive that, such a feedback is a boon to the embedded software engineer who may in turn tune the underlying IoT-based software system to achieve desired performance characteristics. These aspects serve as the main motivation of this paper.

Novel contributions: A general approach for NFP analysis [9] is extended towards model-based timing analysis of UML design model, in [6] and reliability analysis of UML design model, in [7]. However, a workflow for a quick, early model-driven, system-level timing validation of IoT compliant hand-written code in specialized timing analysis tools such as SymTA/S is missing. Addressing the aforementioned gap and extending the work in [6], the following novel contributions are presented in this paper:

- A generic workflow for early model-driven timing validation of IoT-compliant use cases
- Prototype implementation of the workflow using a light-weight interfacing tool framework (with plug-ins) devel-

oped using the Eclipse Modeling Framework (EMF)[1] and the Eclipse plugin development environment.

- Experimental evaluation of the importer/exporter framework and the proposed workflow in a novel, IoT-compliant, real-time emission-quality monitoring system.

The remaining of this paper is organized as follows. Next to this introduction, related work is discussed in section II. The proposed workflow is elaborated in section III. An experimental evaluation of the proposed workflow in a novel, IoT-compliant, emission monitoring use case is presented in section IV. Section V concludes this paper.

II. Related work

Application development for IoT-compliant use cases may be carried out using methods such as (i) traditional hand written code-development, (ii) model-driven development and (iii) mashups. While mashup tools provide a quick prototyping environment, modeling approaches permit very expressive modeling of the systems, possibly with code generation [11]. Several development toolkits, which combine both mashup and model-driven development paradigm, were recently introduced [10]. However, such tools support prototype code generation in the Java programming language only, which is not a commonly used programming language for embedded software development, though involving IoT aspects.

On the other hand, among the Model Driven Development (MDD) approaches for IoT-compliant embedded software development, ThingML [15] is a practical model-driven software engineering tool-chain, which targets resource constrained embedded systems such as low-power sensor and microcontroller based devices. In [4], an UML profile for the IoT, with targeting cyber-physical manufacturing systems, is introduced. The approach in [4] automates the generation process of the IoT-compliant layer for legacy and new mechatronic components alike, to exploit the IoT connectivity. However, support for evaluation of NFPs integrated with the overall development approach is not available in [4], [15].

The UML4IoT profile introduced in [4] is also claimed to be applicable to legacy code, after Reverse Engineering (RE) and annotating the resulting UML diagrams. RE applied to software can be considered as a process of analyzing the subject system (e.g. hand-written code) to create representations of the system at a higher level of abstraction (e.g. UML models). RE is often employed as a process of examination only, wherein the subject software system is not modified. Similarly, in the workflow proposed in this paper, RE is used to primarily extract models at a higher level of abstraction (e.g. UML diagrams) from the embedded software system (e.g. hand-written C++ code) for IoT specific use cases. The UML models, thus obtained, are annotated with the NFPs for performance evaluation, as proposed in [6], [9].

In the UML domain, profiles such as Modeling and Analysis of Real Time and Embedded Systems (MARTE) [14] are specially developed to address such application areas. Despite the fact that many timing requirements could be validated and analyzed on model level, most of the UML tools do not have the capability to handle this. Specialized timing validation tools such as SymTA/S and Timing Architects address the aforementioned aspect. Therefore, an interfacing framework between UML and such timing validation tools is needed. Addressing this aspect, a model-based workflow for validation of timing requirements, employing the MARTE profile, is presented in [8]. However, a specific case study, involving application of the approach proposed in [8] to embedded IoT-compliant use cases, is not available. Similarly, a NFP model for IoT applications presented in [12] may be considered as a starting point for the development of IoT privacy-aware solutions. However, specifics pertaining to addressing the IoT timing-aware solution are not available.

Further, at this juncture, a majority of the embedded IoT applications are developed as hand-written code (e.g. rapid prototyping). For example, the *mbed* platform is a light-weight C/C++ microcontroller development environment that facilitates the user to quickly write programs, compile and download them to various platforms. A quick and early model-driven system-level timing analysis of the code, now available as UML diagrams (after RE of hand-written code), in specialized timing analysis tools would prove beneficial, nevertheless is still missing. Addressing this gap, a model-driven workflow, encompassing a bottom-up approach, for timing validation of IoT-compliant, hand-written code in specialized timing validation tools such as SymTA/S is presented in the following.

III. Proposed workflow

The MDD paradigm with its rich set of features to specify, extract, translate and transform models is gaining significant inroads in automation projects and industrial engineering (e.g. Industry 4.0). However, at this juncture, a majority of the embedded IoT applications are quick hand-written code in programming languages such as C++. In order to subject such a software system to a timing validation at a higher abstraction level, RE may be employed on the hand-written code for the IoT use case. For instance, hand-written code for IoT use cases in the programming language C++ may be reverse engineered (using UML tools) to obtain the design model representation as UML diagrams. Now, with the design model of the IoT use case in hand, a model-driven timing analysis approach may be applied for a quick and early validation of the performance characteristics.

The proposed workflow for an early model-driven timing analysis of IoT use cases is illustrated in Fig. 1. The workflow comprises of the following steps:

1) *Reverse engineering of IoT code* to obtain a design model at a higher level of abstraction
2) *Specifying timing requirements* in the design model
3) *Automatic extraction* of timing requirements and synthesis of an analysis model
4) *Timing validation* and feedback to the design model

In the following the above steps are elaborated in detail. In section IV, a prototype implementation of the proposed

[1]https://eclipse.org/emf/

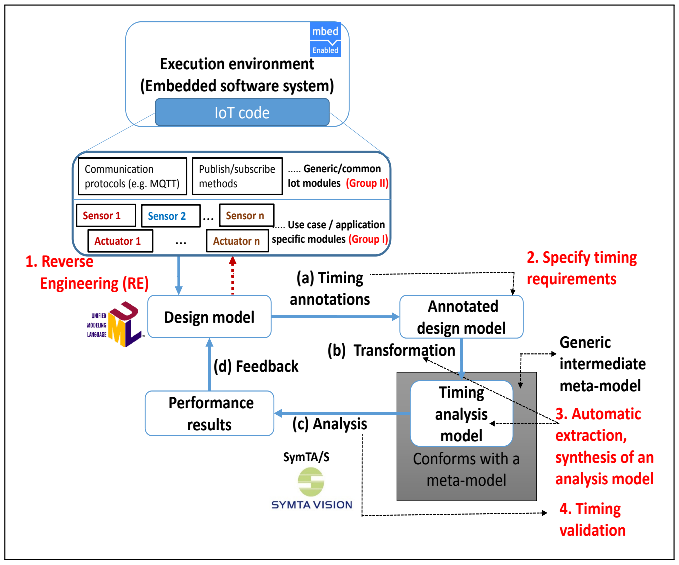

mbed Enabled
Execution environment (Embedded software system)
IoT code
Communication protocols (e.g. MQTT)
Publish/subscribe methods
..... Generic/common Iot modules (Group II)
Sensor 1
Sensor 2
Sensor n
Actuator 1
Actuator n
.... Use case / application specific modules (Group I)
1. Reverse Engineering (RE)
(a) Timing annotations
2. Specify timing requirements
Design model
Annotated design model
(b) Transformation
Generic intermediate meta-model
(d) Feedback
Timing analysis model
Conforms with a meta-model
3. Automatic extraction, synthesis of an analysis model
Performance results
(c) Analysis
SymTA/S
SYMTA VISION
4. Timing validation

Fig. 1. Workflow for model-driven timing analysis of IoT use cases

workflow is elaborated by evaluating it using a novel emission IoT use case.

A. Reverse engineering of IoT application code

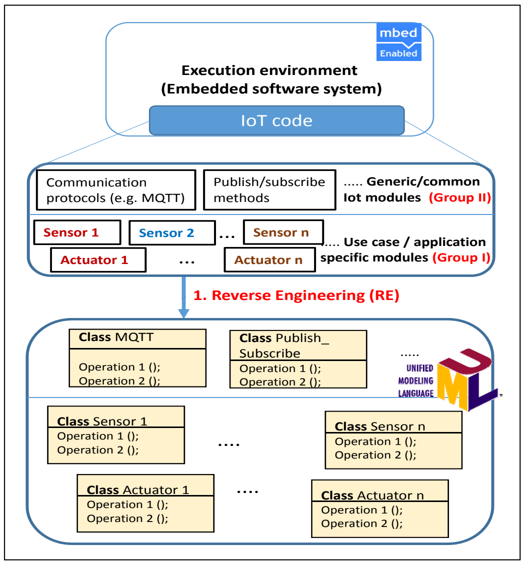


Fig. 2. Reverse engineering IoT application code

As seen in Fig. 2 (before step 1), the IoT application code may comprise of both the application-specific modules and common IoT modules. These are indicated as Group I and Group II respectively in Fig. 2. The group I part of the IoT application code comprises of constructs which are particular to the use case. For instance, this may comprise of the implementation (read/write operations) specific for sensors and actuators belonging to a particular IoT use case. The second group, as indicated in Fig. 2, may comprise of modules which are coarsely independent of the IoT application. For example, this may be the communication aspects implementation (e.g. MQTT and Ethernet protocols) for the IoT use cases.

Such a grouping is introduced here to understand the structure and components of the embedded IoT application code. Further, during timing analysis, the resulting UML diagrams from Group I code (e.g. real-time critical read, write operations of sensors) may be mapped to tasks which are higher priority. The Group II code (less real-time critical operations) may be mapped to tasks with lower priority (in relation to Group I).

In this paper, embedded applications of IoT use cases developed using the programming language C++ are taken into consideration. The object-oriented C++ code is then subject to RE using an UML tool. RE is supported by various UML tools, both proprietary and open source alike. In step 1 of the proposed workflow, when the object-oriented, hand-written C++ code specific for IoT application is subject to reverse engineering, the code structures are mapped into their UML representations: for example, a C++ Class is mapped into a UML Class element, with the variables being defined as attributes, methods modeled as operations, and interactions between the C++ Classes represented by the appropriate connectors (cf. Table I).

TABLE I
MAPPING OF IoT APPLICATION CODE STRUCTURES TO UML REPRESENTATIONS

Code	*UML element*	*Timing analysis*
C++ Class	UML class	Task
Methods	Operations	Runnables
Sequence of method calls	Sequence diagram	Execution path

Class diagrams may be regarded as an invaluable help for software engineers, both developers and maintainers, to understand programs architectures, behaviours, design choices, and implementations. Hence, the class diagrams are used as the primary construct, also for annotating the timing properties in the next step of the proposed workflow in Fig. 1. In addition, a sequence of operations in an execution path, comprising of interactions among the read/write operations of sensors and actuators (in the IoT application code), may be mapped to a sequence diagram.

Thus, as a result of the reverse engineering (from step 1 in Fig. 1), a higher level representation, primarily comprising of UML class diagrams, of the application code for the IoT use case is now available. With the UML class diagrams in hand, the next steps towards timing analysis, namely annotating the class diagrams with timing requirements, can be carried out.

B. Specifying timing requirements in the design model

A general assumption that, for a basic schedulability analysis in a specialized timing validation tool such as SymTA/S [13], a set of tasks with timing properties such as *priority*, *execution time* and *period* are required, is taken into consideration in this work. Hence, the components in the UML design model need to be mapped as a set of tasks for a schedulability analysis in a timing analysis tool (i.e., step 2 in Fig. 1). For the above general assumption, the mapping between the

IoT application code elements, the UML design model and the timing aspects (semantics required for system-level timing analysis in SymTA/S) are shown in Table I.

Please note that only a simplified, yet sufficient, set of timing parameters for a first-hand timing analysis are discussed in this paper, due to space limitations. The proposed approach can be extended for annotation of further timing parameters and/or configurations required for timing analysis in the timing validation tool under consideration. Further, please note that the timing parameters (e.g. execution times for each operation) can be obtained from methods such as Worst Case Execution Time (WCET) analysis, tracing on the target or in a simulator and from budgeting with prior experience. In the examples discussed in this paper, the timing parameters are specified based on budgeting,

In the UML domain, the MARTE profile [14] is a standard by the OMG for specifying timing behavior of real time and embedded systems. For instance, it enhances the UML to support value units and modeling of a platform on which a software application is executed. However, MARTE is an exhaustive profile with hundreds of stereotypes for annotating aspects pertaining to real-time and embedded software/hardware. Therefore an alternative option is to make use of a custom-defined profile with only a few elements (cf. Table I) required for a first-hand and quick timing analysis in the timing analysis tool under consideration.

Once the timing properties are annotated in the UML diagrams, the next step (step 3 in Fig. 1) is to synthesise a timing analysis model from the design model. The timing analysis model would comprise of only the timing properties of the design model.

C. Synthesis of an analysis model

As mentioned above, the timing requirements in the annotated design model need to be extracted and translated to a timing analysis model, in order to carry out a timing analysis (i.e., steps 3, (b) in Fig. 1). Model-To-Model (M2M) transformations can be used to synthesize a timing analysis model based on the design model. Such model transformations aid in bridging the large semantic gap between the source and target model. In our case, the source model is the UML design model and the target model is the timing analysis model.

While employing the M2M transformations, both the source and the target models must conform with their respective meta-models. The source model (i.e., UML design model) conforms by default, with the UML meta-model. However, a target meta-model for the timing analysis needs to be defined to carry out the M2M for synthesis of a timing analysis model corresponding to the UML design model. This meta-model adheres with the semantics of generic timing analysis concepts and also the timing validation tool under consideration (i.e., SymTA/S). A very simplified view of the timing analysis meta-model, created using the EMF is shown in Fig. 3.

From Fig. 3, it is seen that the timing analysis meta-model comprises of a package with several elements required for a basic timing evaluation of a software system, in a hierarchy. The package comprises of *ECUs*, which in turn may encompass *cores*. Each core may have *task*(s) and each task could comprise of *runnables* (e.g. an operation). A task may or may not have a *trigger*, depending on its activation. Apart from these basic elements, there may be several *execution paths* in a software system, which encompass path elements. The path element may be a *task* or a *runnable* as seen in Fig. 3. In addition, each *task* and *runnable* comprises of a *tag* element, which may be used to store additional useful information (e.g. as a string value). Each task and runnable comprise of an attribute to store the execution time (CET). This is used as an input for timing analysis. A result of timing validation, namely the *response time* is an attribute for tasks and execution paths. Please note that only a simplified (yet sufficient) view of the timing analysis model is presented here, because of space limitations.

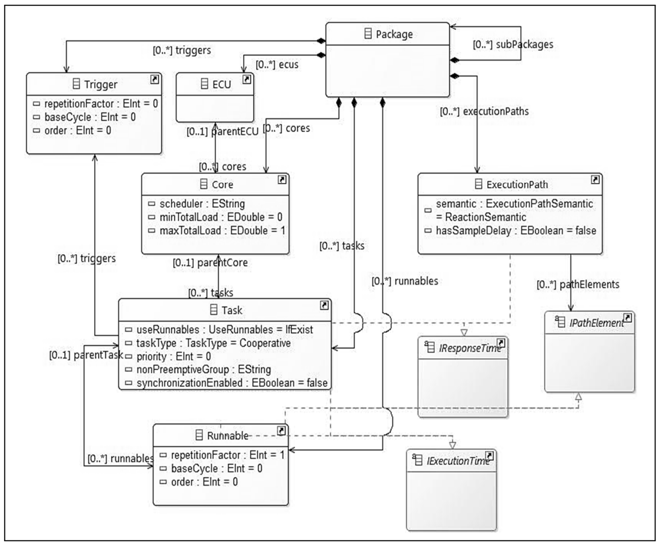


Fig. 3. Timing analysis meta-model created based on the semantics of SymTA/S timing analysis tool (created using EMF)

Thus, at this juncture, for M2M transformations the source model (UML design model, annotated with timing properties) and its meta-model (UML meta-model) are available. The target meta-model (i.e., the timing analysis meta-model in Fig. 3) is also available. M2M transformations can now be implemented in languages such as the Atlas Transformation Language (ATL) for creation of an instance of the timing analysis model, corresponding to the annotated UML design model. Once the timing analysis model is synthesized (i.e., step 3 is complete), it may be exported to the timing validation tool (i.e., SymTA/S) for a schedulability analysis.

IV. Experimental evaluation

In the previous section, the main generic steps in the proposed workflow were outlined. An experimental evaluation of the workflow is elaborated in this section. First, a novel IoT-compliant emission monitoring use case for automotive systems is introduced. Next, a prototype implementation of the various steps in the proposed workflow (in Fig. 1) is elaborated together with examples from the IoT-compliant use case.

A. *IoT use case for emission monitoring*

Around 90% of city residents across the world are exposed to air pollution[2]. Hence vehicle emissions are being strictly monitored and emission norms are being revised regularly to ensure a ”greener” and pollution free environment. In general, the auto exhausts are sensed and analyzed by a machine and a pollution certificate is issued. This certificate, when issued, comprises of an expiry date, hence necessitating regular and periodic checks (e.g. after a gap of 2-3 years). Vehicles are checked for their emissions and if the exhaust is within the specified limit, a certificate might be issued for the vehicle, which may come with an expiry date.

However, this system has its own drawbacks. For example, the system only checks for the emissions on the date of the test and not in between the two test dates. Also, the data is not under load conditions, which might change according to the age and condition of the vehicle. Besides, many factors like engine-tuning and adulteration of fuel might increase the pollution and may move it to an unacceptable level. An appropriate example in relation to the above drawbacks is the laboratory testing of Volkswagen (VW) exhaust, exposing a major scandal[3], thereby drawing global attention on emission standards and tests.

To overcome such drawbacks, the Real Driving Emissions Test (RDE) is expected to become mandatory in the near future for emission checks. Portable Emissions Measurement System (PEMS) is one among the alternatives, to capture real driving emissions. In this paper, an IoT-compliant use case which may be coupled with the aforementioned PEMS system (in place) is used. Such an IoT-compliant system would signal the real-time pollution-level data of the vehicle to the owner. A rather strict pollution check regime may be enforced when such an IoT use case is integrated with the PEMS system.

The use case comprises of an array of sensors to detect the amount of various components of the exhaust. These sensors are implemented to provide real-time information of the exhaust components such as Carbon monoxide (CO) and Nitrous Oxide (NO). Among various choices for an underlying IoT hardware system, the Ublox C027 [2] is chosen for the case study. This hardware unit is chosen for the use case, as it comes with an in-built GPS module and GSM module and is powered by an ARM Cortex M3 processor. Further, the ARM mbed platform is among the most popular IoT platforms with a rich set of device libraries, facilitating rapid IoT application development.

Components of the use case: In the prototype, the use case is built with two main sensors, which detect CO and NO levels. In Fig. 4, the components of the use case are illustrated. The left part of the figure shows the sensors interfaced to the U-blox board (on the right side). The U-blox base board is connected on top with an application shield including a GPS module and LCD display. A GPS module is employed in the use case to derive the latitude and longitude information of the vehicle. The speed of the vehicle may also be calculated using the GPS module. This aspect makes the use case independent of the On-Board-Diagnostic (OBD) data to obtain the vehicle speed. The GSM shield, shown in Fig. 4, is used to send live data captured to the cloud platform. In this use case, the IBM Watson platform is employed for data analytics, based on the data obtained from the use case. Thus, with this simple set up, live emission data (from sensors) is captured against the time and speed (load) of the vehicle and visualized in the IBM Watson platform.

[2]Air quality in Europe, 2016 report https://www.eea.europa.eu//publications/air-quality-in-europe-2016

[3]Official Emission Violation letter to Volkswagen AG from EPA: https://www.epa.gov/sites/production/files/2015-10/documents/vw-nov-caa-09-18-15.pdf

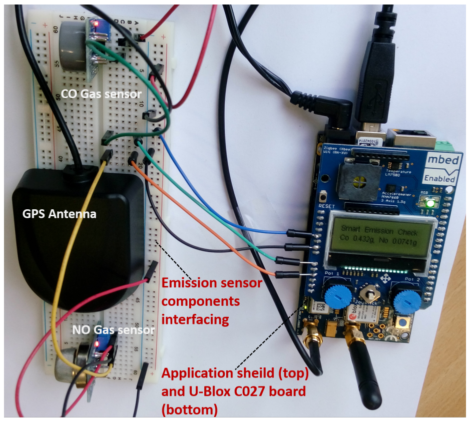

CO Gas sensor
GPS Antenna
Emission sensor components interfacing
NO Gas sensor
Application sheild (top) and U-Blox C027 board (bottom)

Fig. 4. Left: Emission sensor components interfacing, Right: Application shield and U-blox base board with NXP ARM Cortex M3 processor

B. *Prototype implementation of the workflow*

The aforementioned use case is implemented as object oriented C++ code, using the ARM-mbed microcontroller development environment. The use case specific implementation corresponds to interfacing the sensors (for CO and NO gases), the GPS module (e.g. for speed calculation) and the cellular module (for sending data to the cloud). Ready-to-use libraries for the u-Blox hardware unit in the mbed platform facilitates quick application development.

The code thus developed is logically grouped (e.g. group 1 and group 2 in Fig. 2), for ease of understanding the use-case specific and the generic hardware support libraries as seen in Fig. 5-(a). For instance, the modules that may require mapping to higher priority tasks during the timing analysis may be grouped together.

Export of source code and reverse engineering: The source code thus developed, may be compiled and tested on the hardware unit. In parallel, the source code is exported and provided as input for reverse engineering. RE is supported by several model-driven development tools such as IBM Rhapsody Developer [1], which is employed in this experimental evaluation. The corresponding representations for the source code, as UML diagrams, are created by the reverse engineering

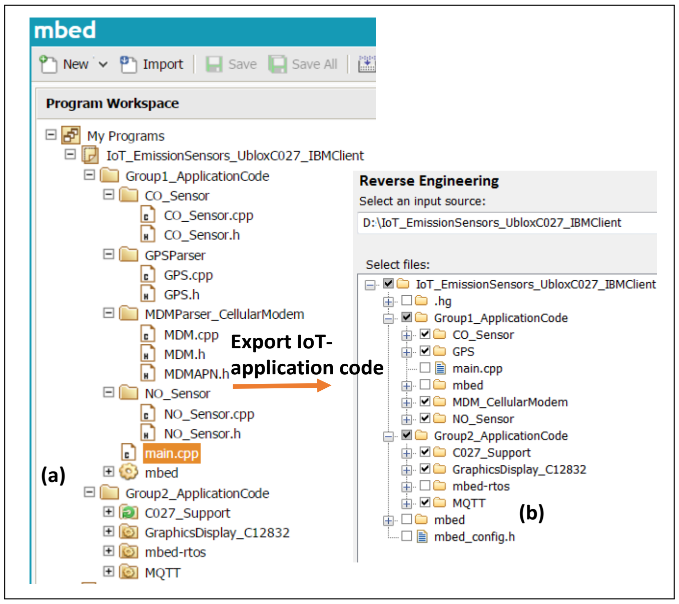


Fig. 5. (a) Application code for IoT-compliant emission sensor use case and (b) Reverse engineering in UML tool [5] after exporting the code

tool. An excerpt of the main components of the IoT application code (in Fig. 5), available as UML diagrams, is shown in Fig. 6. Thus, a design model representation of the IoT use case is now available as UML diagrams, which completes the step 1 in the proposed workflow.

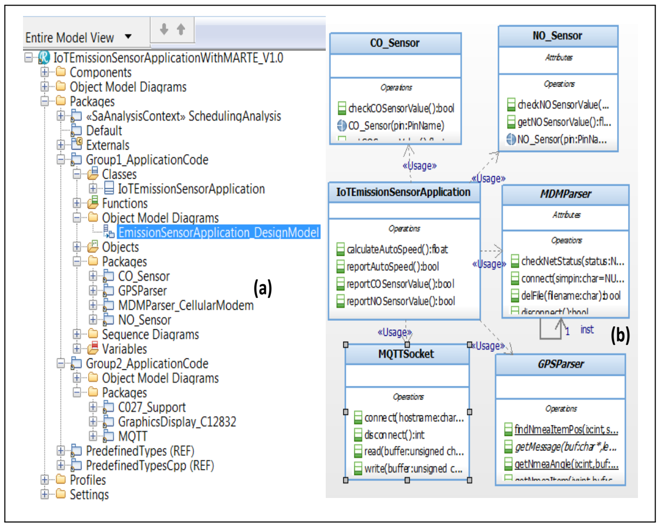


Fig. 6. (a) Reverse engineered IoT-application code structure in UML tool [5] and (b) An excerpt of the main classes constructed as a result of RE

Annotating the design model with timing properties: As seen in step 2 of the workflow, the design model needs to be annotated with timing properties. In this experimental evaluation, the stereotypes available in the MARTE profile for timing analysis of tasks (i.e., a schedulable resource) are employed. For timing analysis, elements such as tasks (*SchedulableResource*), CPU cores (*SaExecHosts*) and a scheduling mechanism are defined using these stereotypes.

The elements are placed in a UML package, in which the stereotype *SaAnalysisContext* is applied to indicate that they are relevant for scheduling analyses. The tagged values *execTime* and *deadline* are used to specify the (maximum) execution time (without interruption). Fig. 7 shows the classes in the design model mapped to tasks and also annotated with timing properties.

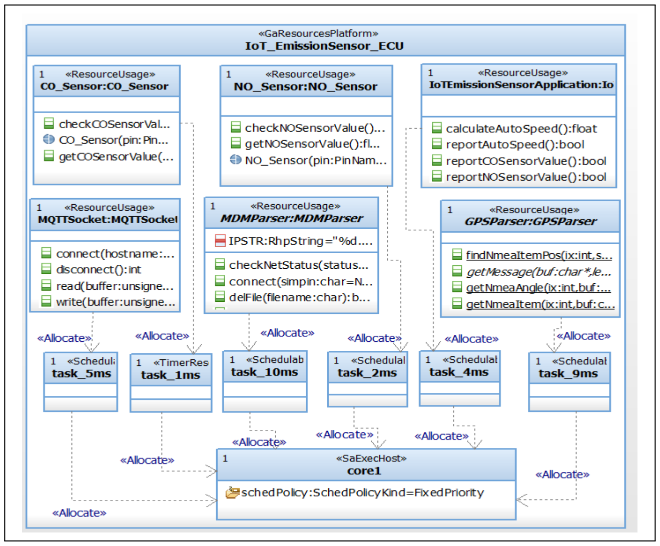

«GaResourcesPlatform»
IoT_EmissionSensor_ECU
«ResourceUsage» CO_Sensor:CO_Sensor
checkCOSensorVal...
CO_Sensor(pin:Pin...
getCOSensorValue(...
«ResourceUsage» NO_Sensor:NO_Sensor
checkNOSensorValue()...
getNOSensorValue():fl...
NO_Sensor(pin:PinNam...
«ResourceUsage» IoTEmissionSensorApplication:Io
calculateAutoSpeed():float
reportAutoSpeed():bool
reportCOSensorValue():bool
reportNOSensorValue():bool
«ResourceUsage» MQTTSocket:MQTTSocket
connect(hostname:...
disconnect():int
read(buffer:unsigne...
write(buffer:unsigne...
«ResourceUsage» MDMParser:MDMParser
IPSTR:RhpString="%d....
checkNetStatus(status...
connect(simpin:char=N...
delFile(filename:char):b...
«ResourceUsage» GPSParser:GPSParser
findNmeaItemPos(ix:int,s...
getMessage(buf:char*,le...
getNmeaAngle(ix:int,buf:...
getNmeaItem(ix:int,buf:c...
«Allocate»
task_5ms
task_1ms
task_10ms
task_2ms
task_4ms
task_9ms
«SaExecHost» core1
schedPolicy:SchedPolicyKind=FixedPriority

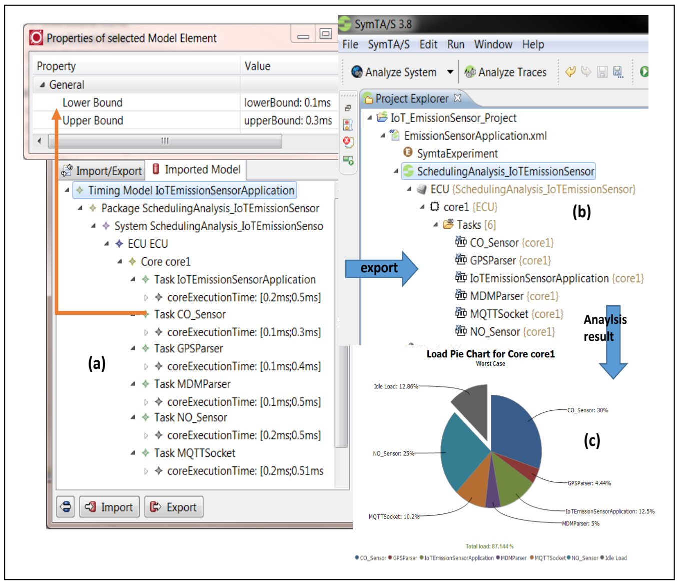
Fig. 7. Emission monitoring IoT application-design model annotated with timing properties

Automatic extraction and synthesis of an analysis model: The timing requirements in the annotated design model are extracted (i.e., step 3 in Fig. 1) using model transformations implemented in the Atlas Transformation Language (ATL). These transformations are invoked as part of a prototype implementation of an importer/exporter plug-in environment. The source and target for these transformations are the UML design model (in Fig. 7) and the timing analysis meta-model (Fig. 3) respectively. The output result of the ATL transformations, is an instance of the timing analysis model as seen in Fig. 8-(a).

The intermediate timing analysis model comprises of only the timing properties specified in the design model in Fig. 7. The synthesized analysis model in Fig. 8-(a) conforms with the meta-model in Fig. 3.

Export and validation of timing properties: The timing analysis model (in Fig. 8-(a)) is persisted in XMI format and exported to SymTA/S, using plug-ins implemented in the Java programming language and EMF. The timing analysis model corresponding to the design model in Fig. 8-(a) is now ready for timing validation in SymTA/S, as seen in Fig. 8-(b).

Extensive analysis of the timing properties, of the embedded IoT application, may be obtained as a result of the validation in SymTA/S, as it is a specialized timing validation tool. For instance, the worst-case load per task for the emission sensor IoT-use case is shown in Fig. 8-(c). Such analysis results provide early feedback regarding the performance characteristics (load, schedulability) of the IoT application software.


Properties of selected Model Element
SymTA/S 3.8
File SymTA/S Edit Run Window Help
Analyze System
Analyze Traces
Project Explorer
IoT_EmissionSensor_Project
EmissionSensorApplication.xml
SymtaExperiment
SchedulingAnalysis_IoTEmissionSensor
ECU {SchedulingAnalysis_IoTEmissionSensor}
core1 {ECU}
(b)
Tasks {6}
CO_Sensor {core1}
GPSParser {core1}
IoTEmissionSensorApplication {core1}
MDMParser {core1}
MQTTSocket {core1}
NO_Sensor {core1}
export
Anaylsis result
Import/Export
Imported Model
Timing Model IoTEmissionSensorApplication
Package SchedulingAnalysis_IoTEmissionSensor
System SchedulingAnalysis_IoTEmissionSenso
ECU ECU
Core core1
Task IoTEmissionSensorApplication
coreExecutionTime: [0.2ms;0.5ms]
Task CO_Sensor
coreExecutionTime: [0.1ms;0.3ms]
Task GPSParser
coreExecutionTime: [0.1ms;0.4ms]
Task MDMParser
coreExecutionTime: [0.1ms;0.5ms]
Task NO_Sensor
coreExecutionTime: [0.2ms;0.5ms]
Task MQTTSocket
coreExecutionTime: [0.2ms;0.51ms]
(a)
Import
Export
Load Pie Chart for Core core1
Worst Case
Idle Load: 12.86%
CO_Sensor: 30%
NO_Sensor: 25%
GPSParser: 4.44%
IoTEmissionSensorApplication: 12.5%
MDMParser: 5%
MQTTSocket: 10.2%
(c)
Total load: 87.144 %
● CO_Sensor ● GPSParser ● IoTEmissionSensorApplication ● MDMParser ● MQTTSocket ● NO_Sensor ● Idle Load

Property	Value
General	
Lower Bound	lowerBound: 0.1ms
Upper Bound	upperBound: 0.3ms

Fig. 8. (a) Imported timing analysis model (b) timing analysis model exported to SymTA/S for validation and (c) Analysis result-load of the core

V. Conclusion

The generic workflow proposed in this paper, makes use of reverse engineered hand-written IoT code, to carry out an early, model-driven timing validation of the IoT application in specialized timing analysis tools. This addresses the aspect of a quick and early timing validation, coupled together with the existing support for rapid prototyping of embedded IoT application software development. Such an early and quick feedback, is a significant advantage to the IoT application developer to understand the performance characteristics of the application under varying timing constraints. An analysis of performance trade-offs among energy and timing characteristics for IoT-compliant use cases, is an item for future work.

Acknowledgment

This work is part of a project supported by a grant (id: KF2312004KM4) from BMWi-ZIM co-operation, Germany. This project is carried out in close cooperation with Willert Software Tools GmbH and SymtaVision GmbH. The authors thank Mr. Sakthivel. M. Sundharam (FSTC/LASSY, Uni. Luxembourg) for his co-operation in building the IoT use case.

References

[1] IBM Watson IoT platform. http://www.ibm.com/internet-of-things/iot-solutions/watson-iot-platform/, Last accessed: 01.03.2017.
[2] mbed enabled Internet of Things kit. https://developer.mbed.org/platforms/u-blox-C027/, Last accessed: 01.03.2017.
[3] ARM mbed development platform for IoT. https://developer.mbed.org/, Last accessed: 01.03.2017.
[4] F. Christoulakis and K. Thramboulidis. IoT-based integration of IEC 61131 industrial automation systems: The case of UML4IoT. In *25th IEEE International Symposium on Industrial Electronics (ISIE)*, 2016.
[5] IBM Rational Rhapsody Developer, Ver 8.2. http://www.ibm.com, 2017.
[6] P. Iyenghar, A. Noyer, J. Engelhardt, E. Pulvermueller, and C. Westerkamp. End-to-end path delay estimation in embedded software involving heterogeneous models. In *11th IEEE Symposium on Industrial Embedded Systems (SIES 16)*.
[7] P. Iyenghar, S. Wessels, A. Noyer, E. Pulvermueller, and C. Westerkamp. A novel approach towards model-driven reliability analysis of simulink models. In *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*.
[8] A. Noyer, P. Iyenghar, J. Engelhardt, E. Pulvermueller, and G. Bikker. A model-based framework encompassing a complete workflow from specification until validation of timing requirements in embedded software systems. *Software Quality Journal*, pages 1–31, 2016.
[9] D. C. Petriu. *Software Model-based Performance Analysis*, pages 139–166. John Wiley & Sons, Inc., 2013.
[10] F. Pramudianto et al. Iot link: An internet of things prototyping toolkit. In *IEEE 11th Intl Conf on Ubiquitous Intelligence and Computing*, pages 1–9, 2014.
[11] C. Prehofer and L. Chiarabini. From internet of things mashups to model-based development. In *39th Annual Computer Software and Applications Conference (COMPSAC), IEEE*, volume 3, pages 499–504, 2015.
[12] S. Sicari, A. Rizzardi, A. Coen-Porisini, and C. Cappiello. A NFP Model for Internet of Things applications. In *IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2014.
[13] SymTA/S: Scheduling Analysis & Timing Verification Tool. https://www.symtavision.com/products/symtas-traceanalyzer/, Last accessed: 01.03.2017.
[14] The UML profile for Modeling And Analysis of Real-Time and Embedded Systems (MARTE). http://www.omgmarte.org/, Last accessed: 01.03.2017.
[15] ThingML modeling language for embedded distributed systems. http://thingml.org/, Last accessed: 01.03.2017.
[16] Timing Architects: Multi-Core Development Tool suite. https://www.timingarchitects.com/tatoolsuite/, Last accessed: 01.03.2017.