ITEA-Project:	COMPACT
ITEA Reference Number:	16018
Funding Organizations:	Austrian Research Promotion Agency FFG (project number 863828 and 864769) Finnish funding agency for innovation Tekes (Diary Number 3098/31/2017) German ministry of education and research (BMBF) (reference number: 01IS17028)
Project Duration:	01.09.2017 until 31.08.2020
Tasks:	T1.1: Use-cases from industrial partners T1.2: Requirements and constraints T1.3: Measures, metrics and their application
Deliverable	**D1.1**
Title	**Requirements and Evaluation Criteria Definition**
Deliverable Type:	Report
Dissemination Level:	Public
Due Date:	30.04.2018
Date of Creation:	30.04.2018
Involved Partners:	ABIX GmbH (ABI) Eberhard-Karls-Universität Tübingen (EKUT) FZI Forschungszentrum Informatik (FZI) Infineon Technologies AG (IFX) Kasper & Oswald GmbH (KAOS) Noiseless Imaging Oy (NI) OFFIS (OFF) Robert Bosch GmbH (RB) SparxSystems Software GmbH (SSCE) Tampere University of Technology (TUT) Technische Universität München (TUM) Universität Paderborn (UPB) Visy Oy (VIS)
Deliverable Lead Partner	SparxSystems Software GmbH (SSCE)

Contents

CONTENTS **2**

INTRODUCTION [SSCE] 3

OBJECTIVES 4

Use Cases [NI] *4*
Requirements Management [SSCE] *4*
Evaluation Criteria definition [UPB] *4*

STATE-OF-THE-ART ANALYSIS 5

Requirements Management [SSCE] *5*
Evaluation Criteria Definition [UPB] *7*

COMPACT USE CASES 8

Overview/Approach [NI] *8*
COMPACT Use Case: Embedded Car Type Recognizer [VIS] *8*
COMPACT Use Case: Device Authentication [KAOS] *10*
COMPACT Use Case: IoT Actor [IFX] *11*
COMPACT Use Case: Bosch [RB] *16*

COMPACT REQUIREMENTS ANALYSIS, ELICITATION & HANDLING [SSCE] 19

Requirements & Constraints derived from WP1-T1.1 .... *19*
Requirements & Constraints derived from WP1-T1.5 & WP5 .... *19*
COMPACT SW & FW Requirements & Constraints *20*
COMPACT HW Requirements & Constraints *21*
Requirements Syntax & Patterns *21*
Handling Requirements in the IoT-PML Modelling Tool *22*

COMPACT EVALUATION CRITERIA [UPB] 23

Overview *23*
Metrics and Measurements *23*

REFERENCES 26

Introduction [SSCE]

This document, entitled “Deliverable D1.1: Requirements and Evaluation Criteria Definition,” serves to provide an overview of the work performed in the first three ITEA3 COMPACT tasks of Work Package WP1:

Task T1.1: “Use-cases from industrial partners,” Task Leader: Noiseless Imaging (NI)

Task T1.2: “Requirements and constraints,” Task Leader: SparxSystems CE (SSCE)

Task T1.3: “Measures, metrics and their application,” Task Leader: University of Paderborn (UPB)

Objectives

Use Cases [NI]

The industrial partners' Use Cases represent the first work results for the COMPACT project and serve as the basis for Requirements elicitation. The objective was to provide real-life implementation examples by partners in their own fields of experience and, as such, with a realistic perception of resource application. The structure has been updated during the first few project months.

Requirements Management [SSCE]

The overall objective for Requirements Management within the scope of this project is to utilize the ongoing development of the COMPACT Toolchain itself to propose, test, and verify a useful, intuitive Requirements Management methodology and specification that adequately handles complexity while providing durable Traceability throughout for end-users of the COMPACT Toolchain.

Requirements Elicitation & Handling

The objective regarding the gathering of requirements is to establish best practices within the overall Requirements Management methodology to ensure that elicitation efforts can provide the information needed to produce high-quality requirements and constraints. Part of this objective is to further ensure the desired quality using pre-defined syntax and patterns that provide for ubiquity and explicitness in requirements descriptions, as well as importing or modeling requirements directly in the IoT-PML modeling tool.

Requirements Traceability

The importance of Traceability has grown within the scope of the COMPACT project so that a workable solution for ensuring traceability of Requirements information from the model through generated, debugged and compiled code and back again has been established as a project goal. The current objective is to determine and implement a Requirements ID format as well as any additional information that can withstand the debugging and compilation processes without putting an unnecessarily burden on what should be 'lightweight' code.

Evaluation Criteria definition [UPB]

The overall objective for evaluation criteria definition within the scope of this project is to define measurements and metrics to determine the progress, success and effectiveness of the methodology and tools developed within the project. Measurements are evaluations based on metrics, which take use cases and demonstrators to indicate and track quantifiable progress during the project. As the main outputs of the COMPACT project are the methodology and tools, their effectiveness shall be evaluated by quantifiable measurements. Additional measurements will determine the level of quality and maturity of the tool-generated software by evaluating memory footprint, performance, power consumption etc.

State-of-the-Art Analysis

Requirements Management [SSCE]

Requirements Handling

There are many ways to write requirements. They could be in the form of a requirement or functional specification document as continuous text, like an essay. Requirements can also be formulated according to certain guidelines to already adhere to a formalism when specifying requirements, offering certain advantages later during the engineering of the system. Using a template, the most important advantages to be gained are:

- Explicitness of requirements
- Completeness of requirements
- Testability of requirements (simpler test case setup)
- Every requirement consists of only one sentence (no chain requirements and therefore better assignment of requirements to test cases and architectural building blocks)
- Easier subsequent formalization (e.g. creation of formal behaviour models) due to the template's semiformal character

An architecture emerges from requirements as a solution, and as such can in turn yield new requirements. A requirements management concept, therefore, must have a high enough degree of flexibility to adequately maintain order and traceability when new requirements are introduced [1].

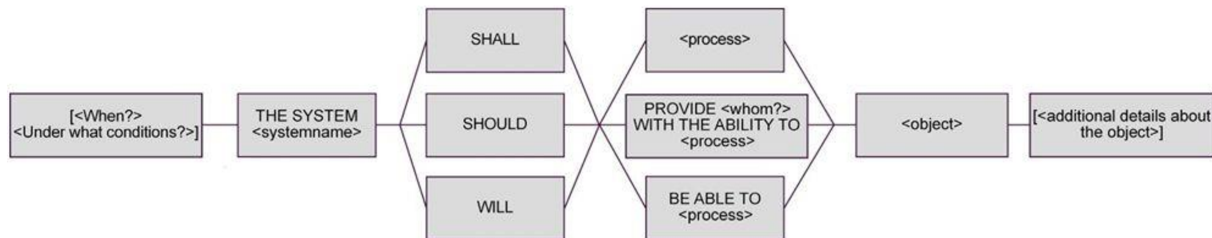

Figure 1: Requirements Template

Using the above Requirements Template [2], a COMPACT Requirement extracted from, for example, the RB Use Case:

> *"The software has to have a low memory consumption"*

might read as follows:

> *"Under conditions of low memory consumption, the software shall be able to function at full capacity."*

Graph-based languages have gained rising attention in the past years. Graph-based means that the nodes of a graph are used as an abstract placeholder for real objects and that, instead of the design object, a graph representation is processed by machines (i.e., computers). Central approaches use visual, graph-based design languages formulated in UML (Unified Modelling Language), which allow both the creation of new engineering models and the reuse of pre-existing engineering models and know-how. The use of the UML as an abstract representation of the future product to be designed allows for the consistent integration of various disciplinary domain models [3].

Survey of Requirements Management approaches [SSCE]

Text Document (e.g. MS Word)

Word is virtually universally available, documents are easy to exchange, the review mode aids in collaboration and sometimes using other tools is simply not an option for different reasons. Hence, most people do use Word to create requirements documents. No reliable automatic numbering or

cross-referencing (traceability) of requirements is directly available in MS Word, though workarounds that make it a more viable Requirements Engineering solution for small, short-term requirements projects can be implemented via macros and the INCLUDETEXT field type [4]. Most commonly, however, such workarounds are not made use of. Text-document-based requirements are hardly traceable, the document's structure must remain basic enough to avoid over-complexity and versioning of requirements is nearly impossible.

For this project and for future customer projects utilizing the COMPACT Toolchain, a text-based Requirements Management is suitable for Use Case description and initial elicitation of Business Requirements, but not for managing Requirements due to the reasons stated above.

Tabular/Spreadsheet (e.g. MS Excel)

MS Excel is highly available and the second-most-popular Requirements Management tool. In 2012, Betsy Stockdale listed the "5 'Joys' of using Excel as a Requirements Management Tool" [5]:

- Many versions of the same document
- Reconciling the various requirement document versions
- Tracking changes made to requirements
- Traceability of requirements is a manual process
- Visibility of most up-to-date document version

Any one of these issues causes an increase in time & effort and/or decrease safety. Solving some of these challenges is possible with more advanced tool features (and/or extensions, integrations) but successful, efficient management of large project Requirements using a tabular spreadsheet solution like MS Excel exceptionally difficult. However, Excel is a viable solution for handling Requirements in small, short-term projects.

A tabular/spreadsheet-based Requirements Management approach can be useful within the scope of a COMPACT Toolchain project for those who wish to organize their Requirements first before uploading them into the Requirements Management tool of their choice (as has been done in the current ITEA3 COMPACT project).

Content Management tool (e.g. Confluence, SharePoint)

Some clear advantages for companies implementing SharePoint or Confluence are availability, usability and approval. When used for Requirements management, such a platform offers a refreshingly dynamic and secure way to handle Requirements content when compared to the methods above; changes to documents are logged, all are aware of its status as the "single source of truth", and traceability can be implemented relatively effectively – in the case of Confluence when used with Jira, and in the case of SharePoint using a combination of functionalities – though with considerable effort and size limitations. A CMS, wiki or document management system is not built around a formal language such as UML (although Confluence plugins exist for modelling in UML [6], Visio diagrams can be displayed in SharePoint [7], etc.). Requirements content can be adequately managed and maintained in the same place as other company content, but that is also where it will remain.

While a framework for the integration of any 'standard' Requirements tool into the COMPACT Toolchain is part of the final project result, and while this might apply to Confluence/Jira in some cases, the model-to-code-focused nature of the COMPACT Toolchain would likely require that the COMPACT Toolchain not take a subordinate role to a Confluence/Jira implementation without significant customer-specific integration effort.

Requirements Tool (e.g. JAMA, IBM DOORS)

Specialized, professional tools for the management of requirements are a major step up from the approaches mentioned above. For example, IBM DOORS (Dynamic Object-Oriented Requirements System) and JAMA are powerful tools that provide real-time traceability including links to other requirements and test cases, collaboration among team members, change management, access via web browser and communication with outside systems via ReqIF (Requirements Interchange Format).

While a Requirements tool is not used in the ITEA3 project itself, it is intended that the COMPACT Toolchain provides connection and communication via ReqIF. Using a Requirements tool throughout the development of the COMPACT Toolchain is not constructive as the RM tool would have to be subordinated to the modelling tool. Also, implementing an additional extraneous tool for handling requirements increases dependency on unknown factors.

Modeling Tool (e.g. Sparx Systems Enterprise Architect)

A UML modelling tool cannot compete with a Requirements Management tool when it comes to intuitive and function-rich handling of requirements. The main benefits of using a modelling tool are that Requirements elements, their dependencies and relationships can be entered and managed directly in the same tool used in the next stage of system development, ensuring better integration into and evolution with that system.

As an additional layer of communication (namely between a Requirements Management tool and the modelling tool), one goal of the COMPACT project is to develop a COMPACT Requirements Management structure as part of the Toolchain for the end customer to best ensure consistent traceability, documentation and reuse within a single source of truth.

Evaluation Criteria Definition [UPB]

The COMPACT project addresses the generation of software for IoT devices. As such, evaluations are based on hardware platforms, which can either be a combination of hardware models or an application of available physical hardware settings. Hardware models can be defined at different levels of abstraction. The COMPACT project mainly plans on Virtual Prototypes (VPs) of microcontroller platforms. A detailed state-of-the-art investigation for VP analysis platforms can be found in Deliverable 1.3. in the Section *IoT analysis*.

COMPACT Use Cases

Overview/Approach [NI]

The “Use Cases from Industrial Partners” Task 1.1 was undertaken to define use cases for testing and evaluating the solutions developed in the project. The use cases have also served as the basis for defining requirements in Task T1.2 with the intention of including all IoT aspects to be covered in COMPACT. Compared to the demonstrators, the use cases have focused on very specific aspects allowing for a tailored analysis and subsequent implementations.

COMPACT Use Case: Embedded Car Type Recognizer [VIS]

Description

In this use case, Visy will develop surveillance video analytics application software that recognizes the types of cars that appear in a surveillance video. As the application is to be developed for a self-contained low-cost surveillance camera, the software description must be lean both in terms of memory footprint and computation resource usage.

The case study application will be provided to the rest of the COMPACT consortium as an application example for validating and testing the COMPACT software synthesis framework. The work invested in this work package will also benefit the demonstrator in WP5 that is built on the same basis.

The surveillance camera’s processing platform model is unique in the COMPACT project in the sense that it contains data parallel image processing hardware (graphics processing unit, GPU) and represents an important category within processing platforms.

The software will be implemented using C++ and OpenCL for speeding up the critical parts of the computation.

Objectives

The development objectives are four-fold:

(1) productivity goals

(2) hardware design goals

(3) application-specific goals and

(4) software design goals.

The productivity related objectives are the following:

- Level of system generation (amount of automatically generated OpenCL code) shall be 80%.
- The objective of automatic OpenCL code generation is to reduce development effort by 90%. The baseline design effort of the manually programmed car type classifier OpenCL implementation was approximately 20 working days. Therefore, with automatic generation we target 2 working days of engineering.
- Performance overhead of automatically generated code should be < 10%.

The hardware related objectives are the following:

- The hardware platform shall be single board.
- The hardware platform shall have passive cooling.
- The camera shall be tightly integrated to the computing device, and both should fit inside standard camera housing.

The application specific goals are the following:

- The recognition should be real-time at a framerate of 25 fps (40 ms average processing time / frame) and standard PAL camera resolution (768x576).
- The software shall detect the presence of a car in front of the camera.
- The accuracy of the car detection shall be over 90% with the dataset collected during the project.
- The software shall recognize the type of the car in the picture with 4 categories: truck, bus, van and small car.
- The accuracy of car type recognition shall be over 80% on the benchmark dataset of the publication [8].

The software related objectives are the following:

- The implementation shall be a standalone library, and shall not have dependencies to other than necessary hardware drivers, etc.
- The source code must be target-agnostic, and maximally portable between hardware platforms.

COMPACT Related Challenges

The use case will exploit the results of WP2-WP4 in the following ways:

- The IoT-PML must support OpenCL and C++ (COMPACT WP2).
- The target code shall be executable on Windows command line, Linux command line and support heterogeneous CPU and GPU targets (COMPACT WP3).
- The compiler must automatically adjust the workload to all OpenCL device models (COMPACT WP4).

COMPACT Related Features and Requirements

Features

- The self-contained smart camera platform represents a very typical IoT device
- The use case involves a significant software emphasis

Requirements

- The software description needs to be target device agnostic
- Code size (lines of code) and performance requirements are indicated

Success Criteria

The baseline for the application use case is the current implementation, which is running on an Intel Xeon based server platform. The input data for the current implementation consists of individual frames triggered from the video stream using physical devices, such as inductive ground loops, photocell or laser scanner (LIDAR). Thus, the implementation is not time-critical, as the amount of data is minimal.

The success criteria for the software implementation are the following:

- The existing neural network may have a memory footprint as large as 500 MB. We target at reducing this to less than 5 MB.
- The number of lines of code in the implementation shall be no more than 10.000 lines of code.

Moreover, the following criteria define the success indicators for the overall system performance:

- The system does not depend on physical triggering (ground loops etc.) but can detect the vehicle from the video stream with 95% accuracy.

- The triggering can be performed in real time (less than 40ms per frame on average).
- The car type recognition can be performed in less than 1s per frame on average.

COMPACT Use Case: Device Authentication [KAOS]

Description

With the expected increase in IoT devices over the next few years, security functionalities are critical for the stability and proper functionality of the overall IoT network.

One of the main concerns in the IoT domain is how to authenticate and identify devices. For this use case, KAOS will develop a set of protocols that allow in-factory personalization of devices so that they can, after being deployed, reliably authenticate themselves against other parties (and vice versa).

The use case will have two sub-use cases:

Personalization of the device.

This use case is executed by the manufacturer in a secure environment. As part of the production process, a functionality within the device is triggered which establishes a unique identity and secret material (e.g. private key) in the device for authentication later. A public portion (e.g. public key) of the generated material is extracted from the device and stored for further use.

If mutual authentication is desired, some authentication data (e.g. another public key) may also be stored on the device.

Authentication towards a third party.

This use case is executed after a device has been deployed and can be triggered by its user. Before a critical operation (e.g. access of restricted of data from some server) can be executed by a device, it needs to prove its identity towards the party it wants to access. By being in possession of the device's public key and executing an interactive protocol, the third party can securely establish its identity—or deny any claims.

If the device is in possession of authentication data for the third party, the authentication protocol can be executed in such a way that both parties mutually authenticate each other.

As described before, the full use case has three actors: A personalization agent (PA), a user (U) and an authentication target (AT). The PA executes the first sub-use case and U triggers the second sub-use case while the AT provides stimuli to the device to prove its identity.

The implementation of this use case will make use of a cryptographic library (encapsulating cryptographic functionality such as key generation, key agreement, response computation, etc.) provided by KAOS for the COMPACT project. This library needs to be highly optimized and thus will be written in C and possibly assembler. As most of the performance-critical code is part of the security library, the „glue" for the implementation of the full use case can be implemented in C++.

Objectives

At the core of the implementation of this use case is a cryptographic library, for which the following objectives hold:

- Implementation of state-of-the-art cryptography to easily allow interaction with non-COMPACT third parties.
- HW-specific optimization of cryptographic code to maximize performance.
- Protection of resulting code against timing side-channels to thwart remote attackers.

In terms of the performance of the cryptographic functions, the two use cases have somewhat different requirements:

- Execution of the personalization step must happen within a reasonable timeframe (10-20s) to minimize impact on the production pipeline.

- Execution of the authentication protocol must be as fast as possible (100-200ms) to be imperceptible to any potential user waiting for it.

COMPACT-Related Challenges

There are two definite challenges regarding implementation of the use case within the COMPACT framework:

1. To leverage the benefits of generating entire IoT platforms from models, security needs to be modelled on a highly abstract level. Sequence diagrams seem to be the best type of models to model communication and add security attributes to individual communications. However, as of now, the tools used in this project do not support the creation of sequence diagrams.
2. To provide a secure implementation of cryptography, a platform to evaluate the actual timing behaviour of the executed code needs to be provided. One of the intermediate results of this project needs to be such a platform, where timings can be obtained with sufficient accuracy and in sufficient quantity.

COMPACT Related Features and Requirements

To allow execution of both sub-use cases, HW and SW means to communicate with the device must be available:

- To trigger the personalization step and exchange public key data, a channel to communicate with the device *must* be found (this could be a JTAG or serial/USB port).
- For the authentication step, the implementation of an exemplary authentication target is necessary, against which device authentication can be performed. To allow implementation of this, the generated firmware (on the OS level) running on the device *must* provide a communication stack which allows to establish TCP/IP connections to external parties.

Furthermore, to allow optimization and evaluation of the cryptographic library, the generated HW needs to meet these requirements:

- The more details about the HW are known at compile time (e.g. register width, CPU frequency, availability of dedicated multipliers, etc.), the better the cryptographic library can be optimized. The IoT-PML *should* provide a way to model relevant characteristics and these *should* be passed to the optimizer.
- To perform statistical analysis of the relevant part of the generated machine code, a simulator or emulator of the targeted RISC-V processor *must* be available. It *must* be possible to measure the execution time of individual instructions with sufficient accuracy and quickly enough to gather statistically relevant data for evaluation.
- The code of the cryptographic library will be written in C, the COMPACT toolchain *must* support the C programming language. An even higher degree of optimization of code is possible only by using hand-written assembly code. The toolchain *may* further allow to use assembly code, if highest performance is a general requirement.
- The PML *must* allow modeling of security requirements for specific use cases in the form of sequence diagrams (or a suitable alternative).

Success Criteria

It is not possible to “measure” security in a quantifiable way. As a result, success is demonstrated by being able to execute the two sub-use cases. The third party (for authentication) can be either simulated—so the use case is purely standalone—but a preferable demonstration is achieved by integrating this use case into a that of a partner.

COMPACT Use Case: IoT Actor [IFX]

The Infineon demonstrator is a light control application as depicted in Figure 2.

It shall have in the one hand side a communication interface to control the light(s). From today's perspective a DLX interface shall be used here.

It shall have on the other side an interface to control LEDs. From today's perspective, it will be one or more PWM signals controlling power transistors which in turn allow turning on and off as well as dimming of power LEDs.

This demonstrator is a smart example for an IoT edge device consisting of a communication interface and actors. Therefore, it is a good demonstration and evaluation vehicle to collect requirements and to derive feedback from and forward it to the technical work.

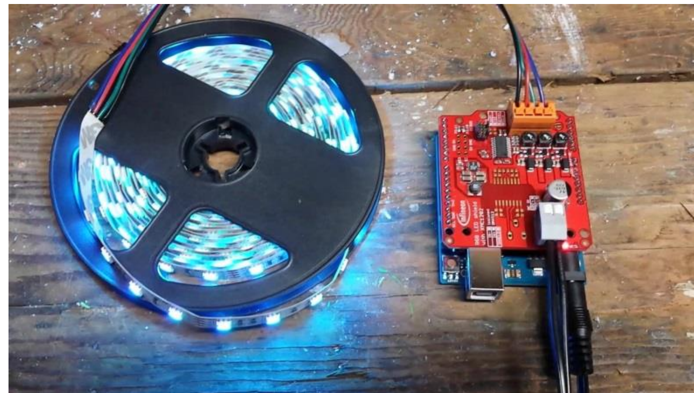

Figure 2: A light control application

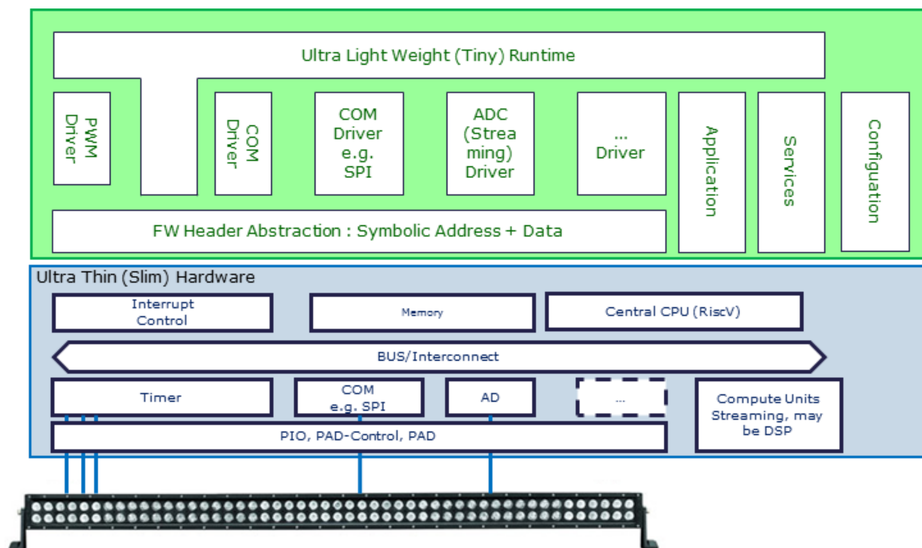

Figure 3: Hardware and Software Platform

At a first glance, there is no technical challenge. As the picture shows there are technical solutions around. The first challenge comes from the underlying hardware architecture, which is depicted in Figure 3.

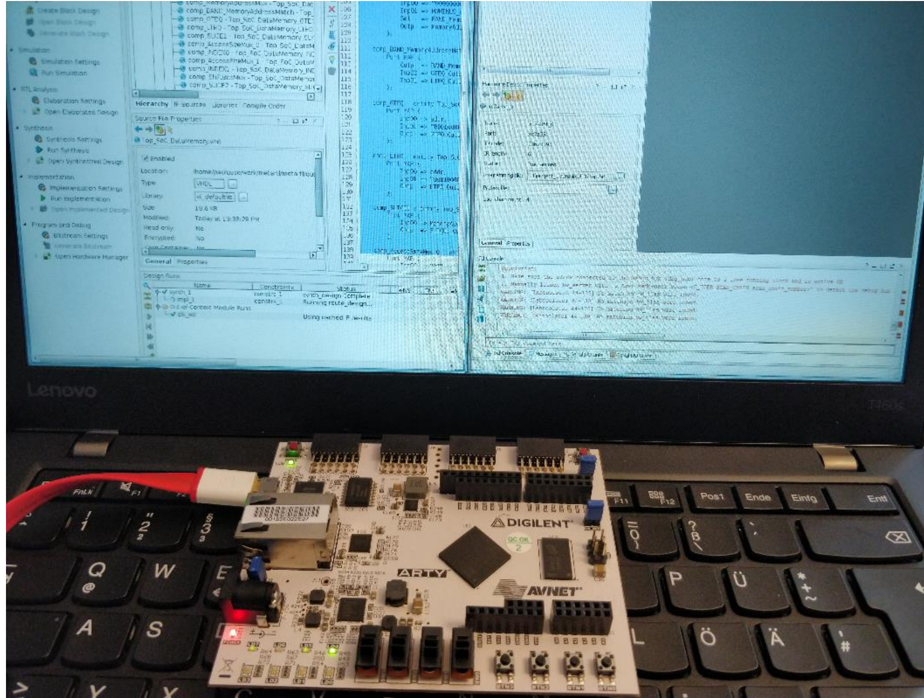

Figure 4: ARTY 35T board configured with RISC-V Subsystem

The hardware platform consists of digital components, subsystem- and architecture descriptions that are highly configurable. Generators are used to build the design, the subsystems and the modules. In this way an application tailored solution can be provided. This yields a minimum die size and fulfils the IoT requirement of smart and cost-efficient solutions.

The generators follow an Infineon in-house developed code generation solution that follows the Model-Driven-Architecture vision of OMG [9]. The approach is built in Python and uses a class diagram like notation. Generation starts with structured specifications (e.g. Framemaker, Dita, RISC-V ISA). Intermediate results are stored in XML. The XML follows the meta-model; however, an extra (generated) XML schema is available as well.

The tool chain generates RTL files and tool scripts for synthesis. For demonstration, Xilinx Vivado is used to synthesize the generated code. A DIGILENT ARTY 35T board with a Xilinx Artix-7 XC7A35T-L1CSG324I FPGA is used as target hardware platform [10] (Figure 4). The software can be loaded via a programmer interface using an SPI interface.

The software can be coded in C and RISC-V assembler – or a mix of both - and compiled and linked with gcc. An ELF based tooling is used to do some adoptions. State-of-the-art firmware header file generation from IP-XACT like Models is used to consistently provide bit-field and register structures to the firmware.

The firmware – shown in Figure 3 – must be flexible to adopt to the selected hardware configuration and itself offer several alternatives. It shall provide the complete interface to the hardware and a light weight runtime system.

1.1.1. Objectives

The – hand coded application – shall be executed on an automatically generated firmware. Only firmware generation will allow automated HW adoption and automated generation of FW alternatives needed for a HW/SW trade-off analysis

1.1.2. COMPACT Related Challenges

Starting backward from WP4 and going via WP3 to WP2, the following challenges must be addressed in the COMPACT work packages.

1. Many different HW and SW components, each with plenty of independent configurations shall be combined to system alternatives.
    a. There is no state-of-the art build process for such an approach
    b. There is no smart handling of all configurations and the models generated with the configurations.
2. Building generators must be efficient. Otherwise too much effort must be spent upfront, that does not pay off through design generation
3. There must be a way to influence the generated code to support different coding styles. This may lead to other challenges
    a. Highly configurable code generator
    b. Transformations on a kind of PSM, e.g. a transformation and generation friendly concrete syntax tree of the target language. A meta-model is required for that. This may be similar, derived, or combined with LLVM/CLANG data structures.
4. Generation and transformation must be combinable in any order. Thus, challenge 1.a. gets even more complicated
5. A code must be provided that is almost as good as hand coded
    a. Efficient translation and transformation utilities are needed
6. Dynamic and static evaluation methods are needed to evaluate the effect of generation options and transformations
7. There must be a meta-model for configuration firmware (part of boot SW)
8. There must be a meta-model and a mechanism for transformation and translation configuration
9. There must be a meta-model defining functions at an abstract level
    a. Meta-model for interfaces
    b. Meta-model for functionality
    c. Bridging hardware (IP-XACT) to run time/OS (e.g. UML activity charts)
    d. Handling various types, also special ones as event
10. It must be possible to relate meta-models to each other's, e.g. by having a way of defining common core features of models
11. An agreement on coding styles must be established to be able to combine results
12. Hand coded and generated code must be usable together

1.1.3. COMPACT Related Features & Requirements

This section contains high-level requirements and features of the methods/tools to be developed/enhanced. They are structured according to WPs and Tasks. If several are touched, the most important one is used for structuring.

1. WP Requirements and Concepts
    1. Task Use-cases from Industrial Partners
        i. The Infineon use case shall contain the following components: timer, interrupt controller, communication device (SPI, I2C, UART, USB), streaming device (may be accessed via special instructions).
    2. Task Requirements and Constraints
    3. Task Measures, Metrics and their Application
    4. Task Technical Project Baseline and ITEA Living Roadmap
    5. Task Selection and Alignment of SW Implementation Patterns and Styles
        i. *If software shall be commonly used, there shall be a coding style defined that guarantees interoperability*
2. WP IoT Platform Modelling Languages
    1. Task Overall Modelling Approach
        i. *A family of meta-models is needed to define the models holding the information to do code generation*
        ii. *A joint semantic/grammar for the family of meta-models is needed to allow interoperability*
        iii. *There must be some basic models for types, objects, assignments, etc.*
        iv. *The meta-models must cover targeted code generation (C,ASM) and the configuration of generation*
        v. *The meta-models must support code generation to generate drivers for at least all devices named in 1.1.i*

2. Task Meta-Models for Non-Functional Properties
    i. No Infineon requirement
3. Task Meta-Models for Specific Functional Behaviour
    i. *The meta-models must provide abstractions beyond implementation languages*
    ii. *The meta-models must enable generation of code variants*
    iii. *The meta-models must support partial generation of run time environments*
    iv. *The meta-models must support functionality that uses several hardware modules beyond CPU and memories, especially interrupt controllers, timers, and communication devices*
4. Task Meta-Models for Firmware Configuration
    i. *There must be a concept for meta-models configuring translators and transformers*
    ii. *There shall be a meta-model for specifying the (initial) configurations*

3. WP Tooling, Tooling Framework and Automation
    1. Task Generators and Libraries
        i. *There must be generators for all HW devices named in 1.1.i*
        ii. *Code alternatives must be generated*
        iii. *Different coding styles must be supported, especially those defined in 1.5.i*
        iv. *Generators must be configurable with meta-models according to 2.4.i*
    2. Task Plugins for Optimiser
        i. *Code generation and transformation/optimization must be enabled by the infrastructure. Especially data for trade-off analysis must be managed*
        ii. *Interfaces for generation/transformation utilities and basic libraries for generation/transformation utilities must be provided*
    3. Task Tools for Highly Automated IoT Software Development Process
        i. *Measures must be provided to keep the overhead for building the generators low*
        ii. *Code generation must be also applied to generator development*
    4. Task Framework Integration of Tools and Artefacts
        i. *All configuration alternatives, generated code snippets, and backend tooling (compiler, synthesis) must work fully automated to support trade-off analysis*
        ii. *All generated alternatives must be stored and handled efficiently*
4. WP Analysis and Optimisation
    1. Task Analysis
        i. *Some activities are needed, not clear yet*
    2. Task Optimisation
        i. *Optimization shall be done by generation of alternatives, controlling the alternative generation, evaluating the properties of the generated software and by selecting the best one(s)*

Success Criteria

Infineon's overall goal of the project is to get the standard work automated to be able to focus on implementation of distinctive features. This means that standard HW- and SW components must be fully automated.

To measure the contribution of COMPACT to this goal, the following SMART measurement criteria are defined

1. <u>Level of System Generation:</u> This is computed by the fraction of generated code and overall used code (characteristic LoC). Code includes all involved code, especially HW and SW.

   Goal is to reach a generation fraction of 80%.
2. <u>Building System Generation:</u> This follows the same computation scheme; however, code of the generation framework is considered (characteristic LoC). Having a high generation level is important to avoid automation cost eating up the automation benefit.

   Goal is to reach a generation fraction of 80%.
3. <u>FW Overhead:</u> Finally, automatically generated code shall not be less efficient - in terms of memory footprint and execution time than hand-coded implementations. An overhead of up

to 10% is seen as acceptable. The number shall be gained from experiments comparing existing hand-written code with generated code.

4. Tiny OS Generation: At least 50% of the code of the run time system shall be generated. This fraction is derived from the lines of hand written code and generated code. Characteristic of the code is defined in LoC.
5. There must be a HW proof of the generated SW: The hand coded and generated SW must run correctly on an FPGA board (e.g. ARTY 35T) configured with a RISC-V sub-system as depicted in Section 1.

COMPACT Use Case: Bosch [RB]

Description

RB-Use-Case #1

The first use case is the software development for sensor platforms that contain a sensor signal processing ASIC with one or more on-chip processors. Since the memory consumption of on-chip software directly influences production cost and because fast response times are crucial for sensor applications, the software must have a low memory consumption and a low computation resource consumption.

The sensor platform application will be used in COMPACT to evaluate and demonstrate the developed software generation methods.

The special characteristic of sensor processing ASICs is that they run lean software that uses very small-sized operating systems on small processors and operates close to the hardware/software interface of the ASIC. This requires software optimization on low abstraction levels.

RB-Use-Case #2

In this uses case Bosch will develop a toolbox of different software components to be used on smart sensor hubs. This toolbox will provide resource-optimized software components to be used by the different applications running on smart sensors.

Since sensor nodes will always have resource limitations the embedded application has to be highly optimized with respect to the footprint, memory consumption, runtime, and power consumption, since many nodes are powered by a small battery. Another aspect are security topics. This also applies to the components coming from the toolbox. Therefore, measures will be taken to evaluate the footprint, memory consumption, runtime, and -- in combination with the application -- also the power consumption. These measures will be integrated in the continued integration environment, so it will be verified automatically.

A second major point during embedded software development is the development time. One measure to speed it up is to have a pre-developed software library as described above and another one is to integrate it into the development tool chain to be able to easily integrate the pre-developed software modules into an embedded application, and to be able to easily configure necessary parts to fit the application's purpose. Therefore, part of this use case is to develop a framework for managing the software components and integrate it to the applications so an application developer can simply "use" the applications.

The developed toolbox as well as the embedded applications will be used to develop a framework to manage and integrate software components to an embedded application and generate code. The software will be implemented using C.

Objectives

The development objectives are four-fold: (1) productivity goals (2) hardware design goals (3) application- specific goals, and (4) software design goals.

Productivity related objectives are the following:

RB-Use-Case #1

- The development effort for software modules that can be generated should be reduced by at least 50%.

RB-Use-Case #2

- The development effort for embedded applications should be reduced by using pre-developed and tested components.
- Test effort for the embedded applications should be reduced because of the reuse of pre-tested modules.

The hardware related objectives are the following:

RB-Use-Case #1

- Both processor and memories are on-chip.
- Processors have a RISC or DSP architecture.
- Both general purpose processors and DSPs shall be targeted.

RB-Use-Case #2

- The footprint of the software shall be as small as possible.
- The memory consumption shall be as small as possible.

The application specific goals are the following:

RB-Use-Case #1

- Typical real-time goals of signal processing and safety-critical systems shall be met.
- The generated software should contain mechanisms for reducing and balancing power consumption.
- The code size should not exceed the code size of handwritten code.
- The generated code should fulfill the requirements of safety standards.
- It should be possible to generate code for communication protocols and direct interaction with the hardware/software interface.

RB-Use-Case #2

- It shall be easy to reuse qualified software components to quickly generate an application.
- It shall be possible to use the framework with other code generation frameworks.
- It shall be possible to add application specific code without worrying about the integrity of the other modules.
- The application specific requirements related to power consumption shall be met.
- The application specific requirements related to security shall be met.

The software related objectives are the following:

RB-Use-Case #1

- The generated code shall be human readable.
- Mechanisms for extending the generated code by handwritten code shall be available.

RB-Use-Case #2

- The requirements related to software quality like reliability, maintainability, readability and so on shall be met.
- The software shall consider all aspects related to defensive and secure coding.
- If relevant, the software shall meet state-of-the-art security requirements.

COMPACT Related Challenges

The use cases will exploit the results of WP2-WP4 in the following ways:

- The modeling languages should support both DSP features as well as constructs for describing communication protocols and direct interactions with the hardware/software interface (COMPACT WP2).
- The toolchain should support safety critical applications (COMPACT WP3).
- It should be possible to integrate the developed analysis methods into UVM-based test-benches (COMPACT WP4)
- The framework to manage and integrate software components must be as lightweight as possible and fit the application development process.
- State of the art security aspects must be considered.
- The software architecture must be defined in a way that fits almost all applications.

COMPACT Related Features & Requirements

Features

The sensor platform represents an example for small signal processing and safety critical systems with a considerable amount of software. The software components and the framework to manage and integrate them together with code generation represents a concept of a secure and fast software development process.

Requirements

Specified code size, real-time and power consumption objectives must be met.

Success Criteria

RB-Use-Case #1

The solutions developed in COMPACT will be evaluated by using a virtual demonstrator platform containing a real-world example of a sensor system. For the demonstrator, generic software modules will be modeled, and the generated code will be evaluated with respect to the defined objectives and requirements.

RB-Use-Case #2

The software modules, the framework and the concepts of how to integrate them to the development process will be used during software development. The average development time will be reduced without losses related to security, code efficiency, and performance.

COMPACT Requirements Analysis, Elicitation & Handling [SSCE]

Requirements & Constraints derived from WP1-T1.1

Requirements Elicitation

Initially, COMPACT Requirements (Work Package 1, Task 2) have been elicited primarily from the various Use Cases created in Task 1. The three main problems in Requirements Elicitation were not adequately solved, namely the problems of Scope, Understanding and Volatility [11].

Handling of Requirements

Requirements Handling in the COMPACT project has evolved since the start of the project to include the handling of requirements directly in the IoT-PML modelling tool itself. This is to better ensure the handing of future complexities as well as traceability (described below).

Requirements Traceability

Requirements traceability is concerned with documenting the life of a requirement [12].

According to the IEEE Systems and Software Engineering Vocabulary, Traceability is defined as follows:

1. The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another;
2. The identification and documentation of derivation paths (upward) and allocation or flow down paths (downward) of work products in the work product hierarchy;
3. The degree to which each element in a software development product establishes its reason for existing; and
4. Discernible association among two or more logical entities, such as requirements, system elements, verifications, or tasks [13].

Currently, some test cases can be derived from COMPACT Use Case descriptions. The need for traceability also fundamentally determines the syntax and patterns used to label and describe Requirements and associated model elements and influences subsequent code implementations and iterations in regard to potential loss of traceability through debugging and compilation processes. Best practices for handling Requirements directly in the IoT-PML modelling tool further establish 'full traceability' and raise the likelihood that traceability durability can be ensured throughout the COMPACT toolchain.

Requirements & Constraints derived from WP1-T1.5 & WP5

[TUM]

Source code representation:

Generated source code should be human-readable and include basic comments. It should follow embedded coding guidelines as will be defined during the COMPACT project in Task T1.5.

Target platforms:

Generated code can be platform-specific and does not have to support all platforms. Target platforms are Linux, Windows and various more lightweight or real-time operating systems or OS-less platforms. It should be possible to target heterogeneous OpenCL platforms as well.

Memory use:

The source code should follow guidelines to reduce the static and dynamic memory footprint.

Traceability:

Locations in the source code should be traceable back to the respective model location.

Security:

Code should follow security guidelines to minimize attack surface.

[UPB]

UPB provides basic contributions for the definition and review of technical requirements and constraints.

COMPACT SW & FW Requirements & Constraints

[SSCE]

Traceability:

It must be possible to track from the code to the design elements. This is to be achieved by a combination of code and model. It must be possible to trace from the code to the model and from the model to the code.

Timeliness of the model:

The model is intended to represent an executable description of the code in a metalanguage that is current in each project phase.

Debugging:

Debugging should be facilitated by a direct connection of code and model. It must be possible for a user to go directly to the point in the model during code debugging.

[EKUT]

EKUT accompanies the requirement analysis by definition of platform-dependent aspects of power management, hardware-specific timing models, constraints of timing and power and trade-off in resulting design corridor, memory-sizes and other platform-dependent constraints.

[FZI]

FZI accompanies the requirement analysis process to understand the industrial needs and to derive more suited concepts. One intended contribution of the FZI is to the development of code generators by considering hardware platform characteristics, to support a generation of interfaces to the hardware. The IFX use case raised that issue in the context of auto-generated firmware to support different HW alternatives and a fluent HW/SW trade-off. In addition, the KAOS use case requires the consideration of HW-specific optimizations during code generation, e.g. to optimize SW-libraries for specific HW platforms. The Visy use case requires having no dependencies to other than necessary hardware drivers. To consider these aspects, the IoT-PML modelling language must model hardware platform characteristics that are considered by the code generators.

Another aspect that the use cases highlight is the consideration of different variants. The IFX and KAOS use cases both consider changing HW platforms, while the KAOS use case also considers different application scenarios for the IoT-device, to adjust the SW-libraries accordingly. This scenario requires modelling scenarios and product variants to derive efficient code and library configurations.

Most of the presented use cases motivate the need of a low memory footprint. The KAOS use case additionally highlights the requirement for considering timing constraints. Based on this, we see the requirement to consider memory and timing analysis during the generation process to assess different realization variants and find a suitable realization.

[IFX]

IFX also contributes to the requirement collection to be met by the generated software. These requirements will be refined in direction of modeling and generation methods.

[OFF]

OFF: (in Collaboration with Bosch) Participation in definition of a) Requirements for the code generation, b) Requirements for the size of the binaries, c) Evaluation criteria and trade-off analysis/DSE, d) Use-Case: MATLAB /Simulink Model → Analysis of Generated Code → Matching with DSP LLVM Size optimizable input code → Definition of source to source translation rules e) Test environment (for WP4 and WP5)

COMPACT HW Requirements & Constraints

[RB]

Bosch defines requirements for on- and off-chip processors to ensure the applicability of the generated software and work on a security concept for smart sensor nodes. Bosch will also develop a method for modeling requirements in the context of the system.

[MP]

MP provides requirements and constraints for image analysis use case application.

Requirements Syntax & Patterns

The need for stricter Requirements syntax & patterns for the COMPACT project was introduced early on with the intention of better ensuring Traceability throughout what will become the COMPACT Toolchain. On the one hand, requirements and constraints specified by the project partners on the COMPACT Toolchain itself shall be adequately ubiquitous, with the level of explicitness to be refined as the COMPACT project moves forward. Currently, requirements are defied as high-level requirements in textual form (Table 1)

Alias	Author	Requirement Title	Version	Summary of the Requirement
BR46	IFX	SW - Software in C	1	The software can be coded in C

Table 1: Requirements are defied as high-level requirements in textual form

whereby the following attributes are recorded:

- **Alias**: The Requirement ID, in this case #46 in the list of Business Requirements (BR)
- **Author**: The COMPACT Partner ID designations are used, in this case “Infineon”
- **Requirement Title**: Includes an indication of area of application (SW, FW or HW), is kept short while roughly describing its object
- **Version**: It is expected that existing Requirements & Constraints will be updated
- **Requirement Summary**: This is the clear Requirement description itself

On the other hand is the question of how Requirements will be handled by developers using the COMPACT Toolchain in the future for its intended purpose. As an example of what a stricter format for Requirements syntax and patterns could look like, the EARS method for formal requirements definition was proposed (Figure 5):

Pattern Name	Pattern
Ubiquitous	The <system name> shall <system response>
Event-Driven	WHEN <trigger> <optional precondition> the <system name> shall <system response>
Unwanted Behavior	IF <unwanted condition or event>, THEN the <system name> shall <system response>
State-Driven	WHILE <system state>, the <system name> shall <system response>
Optional Feature	WHERE <feature is included>, the <system name> shall <system response>
Complex	(combinations of the above patterns)

Figure 5: EARS method for formal requirements definition

Handling Requirements in the IoT-PML Modelling Tool

Adequately-elicited Requirements and Constraints can be imported or modelled directly into the IoT-PML Modelling Tool following the established best practices including syntax and patterns. In this case, requirements derived from COMPACT Use Cases have been imported as a CSV-formatted spreadsheet into the Enterprise Architect UML/SysML modelling tool by SparxSystems.

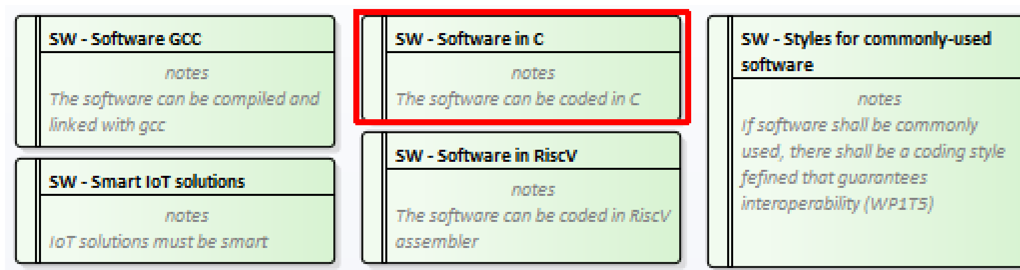

Figure 6: Requirements elements

Double-clicking on the Requirements element (Figure 6) opens the Properties dialog (Figure 7) where the desired set of identifiers and description can be viewed, edited and expanded, while automatically maintaining traceability.

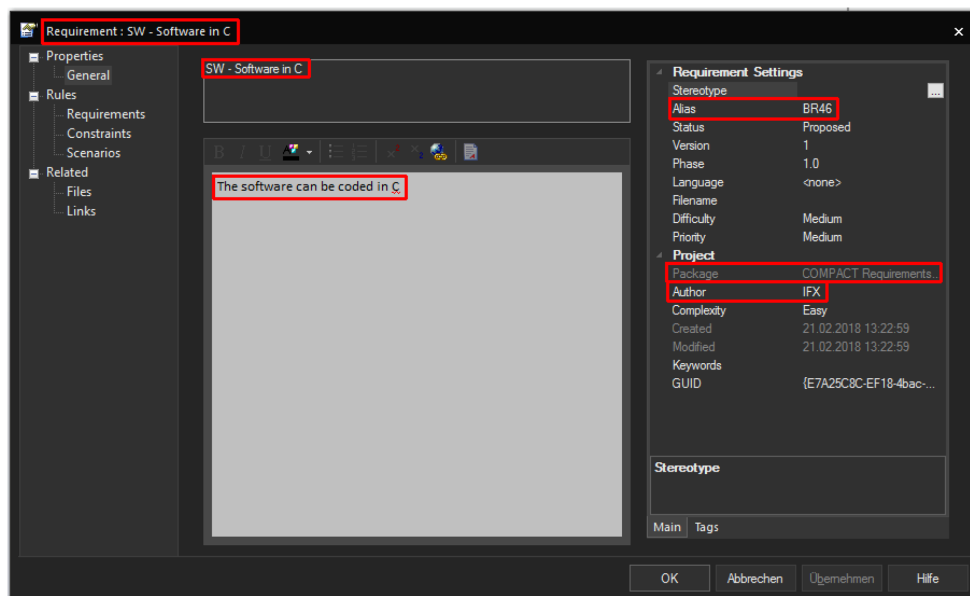

Figure 7: The Properties dialogue box of the Requirement element “SW - Software in C”

COMPACT Evaluation Criteria [UPB]

Overview

This section will introduce metrics and measurements, which are applied for the evaluations conducted during the project. Measurements are taken during the evaluations on use cases and demonstrator platforms to generate quantifiable results. The use cases and demonstrators provide means to compare manual IoT software developments with tool generated IoT software.

Metrics and Measurements

For the evaluation of project results, we can identify the following metrics and measurements divided into two main categories: I. Methodologies & Tools, II. IoT Software. The first one evaluates the effectiveness of the methodology and tools; the second one evaluates properties of the generated software with respect to a specific IoT hardware platform. In each category, the different measurements are listed with a short description, the baseline for comparison, and the metrics with physical units for measurements.

I. Methodology & Tools

a. System Development Time

Description: The project expects reduced IoT software development time from the model-based design flow. The development time includes construction of the model definitions and code generation and compares it to the classical design flow with manual designs.

Baseline: manual development time vs. time with code generation

Metrics:

- time [hours]: initial development time for specification, implementation/generation, and test.

b. System Migration Time

Description: The project expects reduced IoT software change time in the model-based design flow for variant generation, adaption to new platforms, and systems updates. The manual change time of the IoT SW is compared to the change time of the IoT models and the code generation time.

Baseline: manual change time of IoT SW vs. change time of IoT SW models with code generation

Metrics:

- time [hours]: change time of implementations, generation, and test.

c. Proportion of Generated Code

Description: The project targets at a model-based methodology and tools for IoT software generation. An increased proportion of tool generated code will be the main driver for an increased IoT software Quality, Maintainability, and Time-To-Market.

Baseline: manual implemented code vs. tool generated code

Metrics:

- code size [LoC]: size of IoT software source code in Lines-of-Code

II. IoT Software

a. ROM memory footprint

Description: The project expects that the tool generated software code will not require a higher ROM memory footprint compared to manually coded software. The ROM software will include application software and the runtime environment.

Baseline: manual coded software vs. tool generated software

Metrics:

- Memory size [KBytes]: size of software binary after compilation

b. RAM memory footprint

Description: The project expects that the tool generated software code will not require a higher RAM memory footprint compared to manually coded software. The footprint will consider instructions (code segment) and data (data segment).

Baseline: manual coded software vs. tool generated software

Metrics:

- Memory size [KBytes]: code size of data and code segment

c. Heap memory footprint

Description: The project expects that the tool generated software code will not require a higher heap memory footprint during the runtime of the application compared to manually coded software. Evaluations will be taken with varying testbenches.

Baseline: manual coded software vs. tool generated software

Metrics:

- Memory size [KBytes]: maximum heap size

d. Stack memory footprint

Description: The project expects that the tool generated software code will not require a higher memory footprint of the call stack during the runtime of the application compared to manually coded software. Evaluations will be taken with varying testbenches.

Baseline: manual coded software vs. tool generated software

Metrics:

- Memory size [KBytes]: maximum stack size

e. I/O communication

Description: The project expects that the number of I/O access will remain in the same order of magnitude. This will also give a first rough indication for increased latencies in communication without taking network measurements.

Baseline: manual coded software vs. tool generated software

Metrics:

- I/O instructions [#]: static number of I/O access instructions

f. Performance

Description: The project expects insignificant performance losses of the tool generated code compared to manually coded software. Based on a set of different samples the evaluations will measure the minimum, maximum, and average dynamic execution times. Evaluations will be taken on a specific hardware platform with varying testbenches.

Baseline: manual coded software vs. tool generated software

Metrics:

- Execution time [milliseconds]: minimum, maximum, average software execution time on a specific hardware platform

g. Power consumption

Description: The project expects insignificant power consumption losses of the tool generated code compared to manually coded software. Based on a set of different samples the evaluations will measure the peak and average power consumption. Evaluations will be taken for a specific hardware platform with varying testbenches.

Baseline: manual coded software vs. tool generated software

Metrics:

- Power consumption [milliWatt]: peak and average power consumption of the software for a specific hardware platform

h. Code locality

Description: For high security IoT software code, the project expects that the tool generated code provides a guarantee that the cryptographic code is not distributed over different files, nor shall security functions be distributed over different libraries. This evaluation is conducted by a manual inspection of the source code. It identifies and counts the lines of code with security functions.

Baseline: tool generated software

Metrics:

- code size [LoC]: size of security function code in Lines-of-Code

i. Timing side channel protection

Description: The project expects that the cryptographic part of the code of tool generated IoT software is protected against timing side channel attacks. The evaluation will verify that the cryptographic part of the software always runs in constant execution time on a specific hardware platform when applying different input samples.

Baseline: tool generated software

Metrics:

- execution time [milliseconds]: execution time on a specific hardware platform

References

[1] O. Alt, Practical Model-based Systems Engineering with SysML, Vienna: LieberLieber Press, 2016.

[2] SOPHIST GmbH, Schablonen für alle Fälle, 1. Auflage Hrsg., Die SOPHISTen, 2013.

[3] Holder, Zech, Ramsaier, Stetter, Niedermeier und Rudo, Model-Based Requirements Management in Gear Systems Design Based On Graph-Based Design Languages, Applied Sciences by MDPI, October 27, 2017.

[4] M. Junker, „Requirements Traceability With Microsoft Word," 6 10 2016. [Online]. Available: https://www.qualicen.de/blog/?p=272. [Zugriff am 20 3 2018].

[5] B. Stockdale, „5 Joys of using Excel as a Requirements Management Tool," [Online]. Available: http://www.seilevel.com/requirements/5-joys-of-using-excel-as-a-requirements-management-tool. [Zugriff am 23 3 2018].

[6] Atlassian, [Online]. Available: https://marketplace.atlassian.com/plugins/de.griffel.confluence.plugins.plant-uml/server/overview.

[7] [Online]. Available: http://www.metabusinessanalyst.com/managing-business-requirements-on-sharepoint/.

[8] H. Huttunen und F. Shokrollahi Yancheshmeh, „Car Type Recognition with Deep Neural Networks," in *IEEE Intelligent Vehicles Symposium*, June 2016.

[9] K. Devarajegowda, J. Schreiner, R. Findenig und W. Ecker, „Python based framework for HDSLs with an underlying formal semantics: (Invited paper)," in *ICCAD*, 2017.

[10] [Online]. Available: https://www.xilinx.com/products/boards-and-kits/arty.html.

[11] M. Christel und K. C. Kang, „Issues in Requirements Elicitation," September 1992.

[12] O. Gotel und A. Finkelstein, „An Analysis of the Requirements Traceability Problem," in *Proc. of First International Conference on Requirements Engineering*, 1994.

[13] ISO/IEC/IEEE, „Systems and software engineering – Vocabulary".