Firmware Synthesis for Ultra-Thin IoT Devices Based on Model Integration

Arthur Kühlwein
FZI Research Center for Information Technology
Karlsruhe, Germany
kuehlwein@fzi.de

Anton Paule
FZI Research Center for Information Technology
Karlsruhe, Germany
paule@fzi.de

Leon Hielscher
FZI Research Center for Information Technology
Karlsruhe, Germany
hielscher@fzi.de

Wolfgang Rosenstiel
University of Tübingen
Tübingen, Germany
rosenstiel@informatik.uni-tuebingen.de

Oliver Bringmann
University of Tübingen
Tübingen, Germany
oliver.bringmann@uni-tuebingen.de

***Abstract*—Developing firmware for ultra-thin Internet of Things (IoT) devices is challenging due to exceedingly limited hardware resources, increasing functional requirements, and rigorous time-to-market constraints prevalent in industry. Model-driven approaches are often used to tackle these challenges. In the IoT and embedded systems domains, highly specialized metamodels are employed in systems engineering, including the development of firmware. However, these metamodels exist in isolation, limiting the capabilities of model-driven activities. In this paper, we show how firmware synthesis for ultra-thin IoT devices can be enhanced by model integration which is realized by a novel unifying modeling language that aims at integrating the large number of dedicated metamodels. We demonstrate our approach with an industrial use case where we synthesize parts of the firmware for one of the sensor peripherals of an IoT device along with contracts enabling static code verification.**

***Index Terms*—Internet of Things, Model-Driven Engineering, Code Generation, Software Verification**

I. Introduction

The Internet of Things (IoT) is on the rise and expected to grow dramatically over the next decade [1]. Ultra-thin IoT devices with sensors and actuators need to be smart and cheap, while consuming only minimal amounts of energy. Such resource-constrained IoT devices pose a serious challenge for the development of firmware, i.e. low-level software closely interacting with the hardware that is deployed on these types of devices. On the one hand, firmware for IoT devices must be ultra-thin with an extremely small memory footprint and ultra-low energy demands. On the other hand, IoT devices contain extensive software functionalities, such as real-time computing capabilities, connectivity, security, safety, and remote update mechanisms. A typical industrial environment puts additional pressure on firmware developers through rigorous time-to-market constraints. Model-driven approaches are a promising way of tackling the challenges at hand, as they have been used successfully in a number of different domains, such as telecommunications [2], health [3], and web engineering [4].

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) in the ITEA3 project COMPACT under grant 01—S17028C. The authors are responsible for the content of this publication.

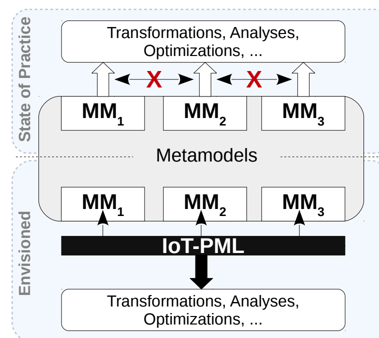

Fig. 1: The current state of practice and our vision with the IoT-PML.

In the domain of embedded systems and IoT, a number of dedicated, highly specialized metamodels are used for different purposes along the IoT value chain (see Section II-A).

In the current state of practice, the metamodels exist in isolation, as depicted in the top half of Fig. 1. Due to this separation, the metamodels cannot be easily integrated, leading to a variety of issues. It limits the capabilities of model-driven activities to the expressiveness of the metamodel they are working on. In particular, the automation of firmware development is difficult because no common interface exists between the different metamodels [5]. Furthermore, it hinders co-design and coordination [6], which is of particular relevancy since hardware/software co-design is a common practice in this domain. Design errors may be detected late in the development cycle or even worse, when the product is already in operation. Such errors can result in significant financial repercussions [7]. In the worst case, an error can potentially lead to catastrophic system failures, as IoT devices are typically embedded in a safety-critical context [8].

One way to tackle these issues is a modeling language that is able to tie together these multifaceted metamodels, as shown in the bottom half of Fig. 1. Such a language increases the power of model-based activities by exploiting data synergies, that is, complementary relationships between

the data provided by the metamodels. We envision model-based tools to use models of this language as their primary source of information, which enables a holistic approach to the automated synthesis of firmware for resource-constrained IoT devices by creating a common shared interface between the different metamodels. In this paper, we make two contributions in this direction. We first introduce a unifying, interoperable IoT Platform Modeling Language (IoT-PML), which captures the entire modeling worfklow at various levels of detail and links together the various metamodels. We then demonstrate how the IoT-PML can enhance synthesis of firmware annotated with contracts for static code verification for one of the sensor peripherals of an ultra-thin IoT device.

The remainder of this paper is structured as follows. Section II explains the technical background related to our contributions. Section III describes the IoT-PML. Section IV illustrates how the IoT-PML can enhance firmware synthesis in top-down use case. Section V describes related work. Section VI concludes this paper and provides an outlook on future work.

II. Background

A. Metamodels for IoT Firmware Development

Firmware running on IoT devices is embedded software. Thus, the collection of metamodels that are related to the IoT and embedded software domains is comprehensive, ranging from metamodels of general-purpose modeling languages to metamodels of highly specialized, dedicated domain-specific modeling languages (DSMLs). In addition, a large number of metamodels have been proposed by research, addressing issues such as IoT node connectivity and configuration [9], [10], security [11], service discovery [12], and runtime adaptability [13]. It is worth noting that these metamodels typically focus on the device network and have limited expressiveness in terms of modeling individual devices. As enumerating all relevant metamodels here would go beyond the scope of this paper, we will only present a limited selection of hardware- and software-centric metamodels most relevant for our work.

1) IP-XACT: IEEE 1685-2014 (IP-XACT) [14] is a standard for the detailed specification of intellectual property (IP). It defines an XML format for IP structure, configuration, and tool flow, enabling reuse and integration between different IP vendors. The IP-XACT metamodel is represented by an XML schema definition (XSD). An IP-XACT description of an IP is a vendor-independent electronic datasheet describing the IP interface, hardware registers, and related file sets required for IP integration. The standard provides only very limited support for the inclusion of software aspects.

2) UML and UML Profiles: The Object Management Group (OMG) has standardized a number of graphical modeling languages. The most prominent representative is arguably the Unified Modeling Language (UML) [15], which can be extended for domain-specific applications via its profile mechanism. In the embedded software domain, there exist profiles such as the OMG Systems Modeling Language (SysML) [16] or the OMG Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [17]. The metamodels of these languages all conform to the OMG Meta Object Facility (MOF) [18]. Constraints in MOF-conformant models can be expressed using the Object Constraint Language (OCL) [19].

B. Software Verification

IoT devices deployed in a safety-critical context have stringent requirements on quality and reliability with zero tolerance for failure, which entails tremendous design efforts that need to be spend on rigorous verification [20]. In particular, firmware can crash the entire system due to a faulty device configuration or critical runtime errors.

Static analyses can address the set of problems arising in software verification by proving the abscence of runtime errors such as out-of-bounds array accesses or integer overflows. Furthermore, they can show that critical program states which can result in a system malfunction or crash can never be reached. From the wide range of static analysis approaches [21] and associated tools that are available, we adopt the methodology of Verified Concurrent C (VCC) [22], which we utilize for our use case in Section IV. VCC is an industrial-strength verification environment for low-level C code that is geared towards modular and sound verification of functional properties using contracts and invariants on states.

Function contracts specify the properties of a function using three types of clauses. Pre-conditions, introduced by the **requires** clause, state the assumptions that must hold before calling the function. Post-conditions, introduced by the **ensures** clause, define the guarantees a function makes after it returns to the caller. In addition the **writes** clause specifies the way a function accesses areas in memory, i.e. it states explicit permissions to modify program state.

Object invariants can be considered as contracts on data. While function contracts specify state consistency on entry to or exit from the function, object invariants associate such consistency with the data itself. In VCC, object invariants are predicates expressed in first order logic on data and can be associated with compound C types (structs and unions).

Ghost code provides additional means of specification for functional properties in VCC. Ghost code is seen only by the static verifier, not the regular compiler and is mainly used to introduce abstractions and states one can reason about more efficently afterwards. Ghost code and operational code is strictly separated and thus any flow of data from ghost state to operational state of the software is forbidden.

III. Language Description

The IoT-PML captures essential concepts used within the embedded sytems domains, including functional requirements concerning the software and hardware platform, non-functional requirements such as power consumption, as well as device configurability and usage scenarios. The IoT-PML metamodel uses MOF as its meta-metamodel. Some of the key points behind this decision are described in Section III-D.

In order to integrate the various modeling languages into the IoT-PML, we have been carefully analyzing their metamodels (see Section II-A) and identifying common abstractions which

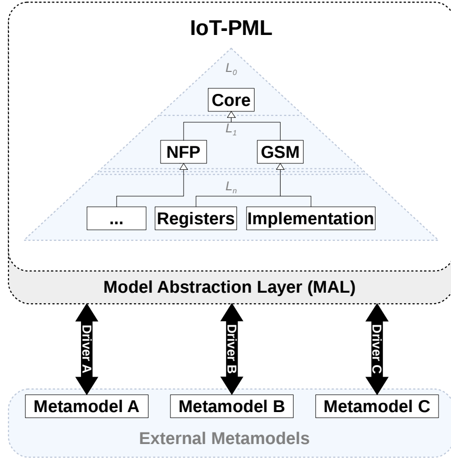

Fig. 2: Conceptual architecture of the IoT-PML.

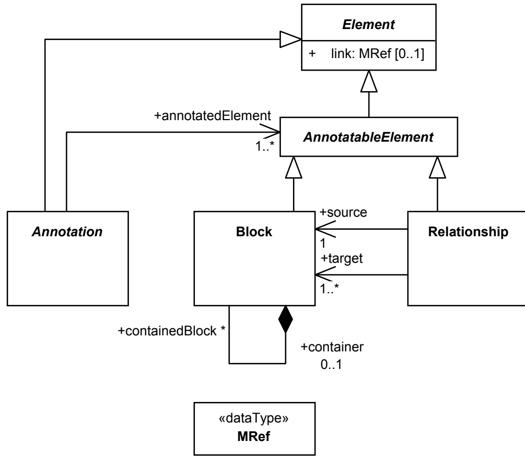

Fig. 3: The domain model of the IoT-PML Core module.

we then added to the IoT-PML metamodel. Our goal here is not to create a gargantuan metamodel that is able to reflect every detail of every IoT-related DSML. Instead, we aim to provide just the right level of abstraction to enable effective DSML integration and cooperation, enhancing the capabilities of model-based activities.

A. Architecture - Layers, Modules, and Concepts

The metamodel has a modular, layered architecture, which is illustrated in Fig. 2. Each layer L_i subsumes concepts at a particular level of genericity $i \in \mathbb{N} \cup \{0\}$, with $i = 0$ indicating the highest level of genericity. Related concepts are packaged into modules, which reside in exactly one layer. These concepts represent common abstractions of a number of heterogeneous metamodels. The lower the level of genericity, the smaller the number of metamodels to which these concepts apply. The modules within a given layer $L_i, i > 0$ may only depend on modules on the same or on lower layers. Each concept in a given module must inherit from at least one concept of the Core module, either directly or indirectly.

The Core module, which is located at layer L_0, is a lightweight kernel metamodel providing basic generic concepts used by the modules at all subsequent layers, such as the general system modeling (GSM) or non-functional property (NFP) modules. The NFP module provides basic concepts for modeling NFPs related to timing as well as memory and power consumption, enabling model-based analyses and optimizations. For instance, power saving modes of a given hardware component can be described using concepts from the NFP module. Fig. 3 shows the domain model of the Core module.

At the most abstract level, the Core module has the concept of an ***Element***[1], which is the basic type for each element in the IoT-PML. This concept is subdivided into the concepts ***AnnotatableElement*** and ***Annotation***. ***AnnotatableElement*** is further subdivided into the concepts ***Block***, which may contain other ***Blocks***, and ***Relationship***, which models directed relationships between ***Blocks***. Conceptually, the ***Block*** concept is aligned with the *Block* element introduced in SysML.

Layer L_n[2] is the outermost layer of the IoT-PML metamodel, containing modules at the highest level of detail which usually reflect concepts common to only a limited number of DSMLs. An example of such a module is the Registers module, which facilitates modeling of hardware registers. It is closely aligned with the IP-XACT standard, but extends it by introducing additional subtypes for registers and bit fields, such as command or configuration registers. Transformations using the IoT-PML typically work with modules located at this layer.

B. Facilitating Model Linkage

A layered architecture enables us to address semantic and structural heterogeneities of closely related metamodels by bridging related concepts via common abstractions. As can be seen in Fig. 3, the ***Element*** type is able to reference an external metamodel element via its ***link*** attribute. Consequently, every concept in the IoT-PML has this ability, which facilitates the linkage between data stored in heterogeneous metamodels. The reference to a given external model element e is realized as a model reference (***MRef***), which has the format

$$\mathrm{M}\,\mathbf{:}\,\mathrm{URI}_e,$$

where M is the globally unique identifier for the external metamodel and URI_e is the Uniform Resource Identifier (URI) of the referenced model element.

The actual linkage during runtime is performed by a model abstraction layer (MAL), which is illustrated in Fig. 2. The MAL provides a set of concept-specific APIs for reading the data contained in the referenced model, so they can be used for transformations or analyses. Metamodel-specific drivers implement the MAL APIs for each supported concept to facilitate data handling.

This approach is comparable to the model connectivity layer employed in the Epsilon language family [23] or the

[1]Throughout this paper, we adhere to the following stylistic conventions. Elements of the IoT-PML are ***boldfaced and italicized***, while elements of external metamodels are only *italicized*. We use upper-case letters for element types, while attributes of elements are written in lower-case letters.

[2]At the moment, the number of layers is not fixed as the IoT-PML is in its early stages of development.

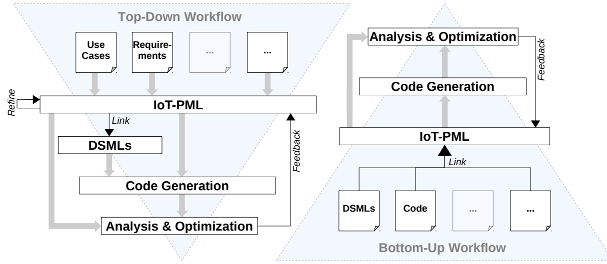

Fig. 4: Conceptual top-down and bottom-up workflows with the IoT-PML.

technological connectors of OpenFlexo [24]. No explicit semantics are defined for this link, which creates a loose coupling between the IoT-PML metamodel and the various external metamodels, giving users the ability to select and link to the metamodel best suited for the task at hand. For our approach to work, we have to assume that the ***MRef*** of the referenced model element does not change between subsequent serializations and deserializations of the external model.

C. Extensibility

Users can extend the IoT-PML by adding so-called user modules, which can be located at any layer in the IoT-PML, but may not introduce additional layers. Concepts in user modules have to conform to the same restrictions as the concepts contained in built-in modules, that is, the rules described in Section III-A apply. User modules extend the MAL by introducing additional interfaces related to the concepts defined in the user module. At model runtime, when an IoT-PML model is loaded, the IoT-PML metamodel and MAL are constructed dynamically by merging the data from the built-in IoT-PML modules and user modules. In this sense, user modules can be seen as plug-ins to the IoT-PML.

D. Implementation

Currently, the IoT-PML is a UML profile that uses a limited subset of the UML metamodel, comparable to SysML. We followed a best-practice approach[3] in the design on the profile. Each layer and module is represented by a package and sub-profile, respectively. The module sub-profiles contain stereotypes representing the module concepts. The stereotypes extend the UML metaclasses which represent the closest match in concept and semantics. In addition, attributes are transferred over to the extended metaclass where possible. On the one hand, this transfer reduces the size of the profile and avoids data duplication, but is also required for certain relationships between ***Blocks*** which cannot be modeled as inter-stereotype relationships, such as composition [25].

There are a number of alternatives for the implementation of the IoT-PML, but we deem a UML profile to be the best choice for a number of reasons. Being a software-centric modeling language, we can exploit the expressiveness of UML for software-related aspects. In addition, a profile allows us to leverage the large ecosystem that has evolved around UML and related MOF-based standards. This also includes model-related tooling, such as analysis and transformation frameworks. Furthermore, it simplifies the integration of other MOF-based metamodels and eases the facilitation of tool support, given the fact that industrial-grade UML-based modeling tools typically provide generic mechanisms to work with UML profiles.

Our implementation of the IoT-PML and the MAL is based on the Eclipse modeling ecosystem, in particular the Eclipse Modeling Framework[4] (EMF) and the Model Development Tools project[5], which includes the Papyrus modeling environment[6] and an EMF-based implementation of the UML metamodel. The MAL is a set of callback operations which can be used by model-to-model (M2M) and model-to-text (M2T) transformations. User modules and the corresponding MAL extensions are contributed to the IoT-PML via Eclipse extension points.

E. Workflow Integration

The IoT-PML is kept as generic as possible, making it methodology-independent and thus reusable in different contexts. It supports both top-down and bottom-up approaches.

1) Top-Down: The layered architecture of the IoT-PML lends itself to top-down workflows similar to the OMG notion of model-driven architecture. In addition, the language provides a number of architectural views which are motivated by the IEEE 1471-2000 [26] recommendations for the architectural description of software-intensive systems. Each of these views is designed to address the needs of different stakeholders during a specific phase of IoT software development and is encapsulated in individual modules. One example of such a view is the Subsystem view which describes the composition of and relation between different system components. The architectural views of the IoT-PML can be used to gradually refine the model in a classical top-down approach, starting from informal or formal use cases, requirements, and other specifications, as illustrated in the left-hand side of Fig. 4. Once the IoT-PML model is sufficiently refined, different aspects of the IoT device are modeled using dedicated DSMLs. These aspects are captured in the IoT-PML model using model links. Code is then generated primarily from the IoT-PML model, using model data from the referenced DSMLs. The workflow may also include models from DSMLs which are not linked to the IoT-PML model. These DMSLs may also contribute to the generation of code. A variety of analyses and optimizations can also be performed on the IoT-PML model and the generated code, and the results can be fed back to the IoT-PML model enabling an iterative top-down model-driven design approach.

2) Bottom-Up: In a bottom-up workflow, the IoT-PML can be used to integrate various existing artifacts, as depicted in the right-hand side of Fig. 4. These artifacts typically comprise models from other DSMLs, but may also include

[3]For instance, see https://www.itu.int/rec/T-REC-Z.119-200702-I/en

[4]https://www.eclipse.org/modeling/emf/

[5]https://www.eclipse.org/modeling/mdt/

[6]https://www.eclipse.org/papyrus/

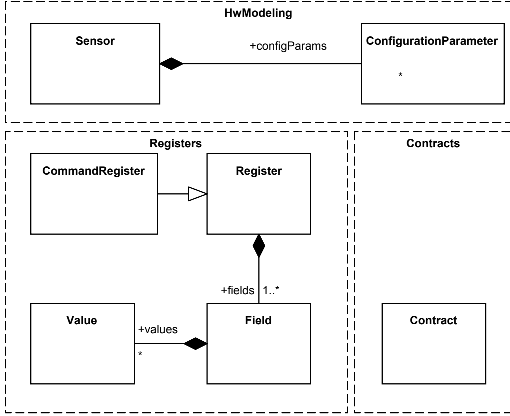

Fig. 5: Simplified domain models of the L_n IoT-PML modules containing some of the basic concepts referenced in the use cases.

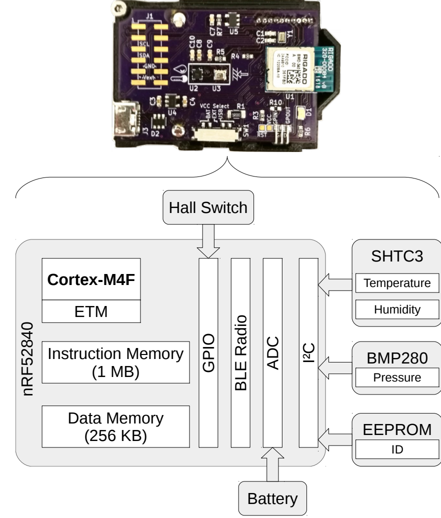

Fig. 6: Architecture of the IoT sensor device.

other artifacts, such as legacy code. Tools using the IoT-PML model, such as code generators, optimizers, and analyzers, can then exploit synergies between the data provided by the different models using the MAL. Like in the top-down workflow, results of these tools can be fed back to the IoT-PML model for an iterative design approach.

IV. Use Case

We present a use case illustrating a top-down workflow using the IoT-PML based on a typical scenario.

A. IoT-PML Modules

The use case utilizes a number of IoT-PML modules located at layer L_n, whose simplified domain models are shown in Fig. 5. Each of the concepts depicted here directly or indirectly specializes concepts of the Core module, but for the sake of clarity we have left out any cross-layer dependencies in the figure.

The Contracts user module comprises only one concept, the ***Contract***, which specializes the ***Annotation*** type in the Core module. The ***Contract*** concept models the function contracts and object invariants of VCC that are required for the formal verification of the device driver. In the use case, we show how modular verification can be achieved using contracts. We argue that contract-based design perfectly suits the top-down workflow using the IoT-PML by hiding irrelevant detail from the reasoning engine. The system designer can state the specification of a function that has not been written yet and provide implementation details during subsequent iteration steps. Because each function can be verified separately by only using the specification and not the implementation, changes to the function body can be continuously checked against the specification.

The Registers module contains concepts related to the modeling of hardware registers and is closely aligned with the IP-XACT standard. In addition, it comprises some semantic extensions to ***Registers***, such as ***CommandRegister***. Finally, the HwModeling module includes concepts related to the modeling of hardware, such as ***Sensors***, which can be configured at runtime with ***ConfigurationParameters***. The concepts in the HwModeling and Registers modules all specialize the GSM module, while the Contracts module directly specializes the Core module.

B. IoT Sensor Device

Fig. 6 depicts an ultra-thin IoT sensor device featuring commercial off-the-shelf (COTS) peripherals, which communicate with the main processor via the I^2C interface. One of these peripherals is the Sensirion SHTC3 temperature and humidity sensor, which uses a 16-bit command register.

In order to ease application software development and reduce development time, the firmware for the sensor device is running on RIOT OS[7], which provides the basic infrastructure for essential tasks such as thread scheduling and hardware interfacing. Because the sensor device is running in a safety-critical environment, the part of the firmware responsible for handling the SHTC3 peripheral, i.e. the driver, needs to be verified. We assume that the peripheral vendor provisioned a datasheet and supplementary IP-XACT description. The RIOT OS driver for the COTS sensor needs to be developed first, since no such driver is available for the OS. To speed up development time, we want to generate a skeleton of the driver code from an IoT-PML model, including contracts for verification. An overview of the use case illustrating the main steps and artifacts is given in Fig. 7.

In order to facilitate the generation of the driver skeleton, a driver-centric IoT-PML model (1) of the COTS sensor is created using its datasheet and IP-XACT description. Here, the COTS sensor is modeled as a *Component* with a ***Sensor*** stereotype. The command register of the COTS sensor is modeled as a ***CommandRegister*** which links to its IP-XACT description using an ***MRef*** as described in Section III-B. In this case, the IP-XACT model and the register within the model are referenced using a relative path pointing to the XML file and the XML identifier of the register, respectively. Device

[7]https://riot-os.org/

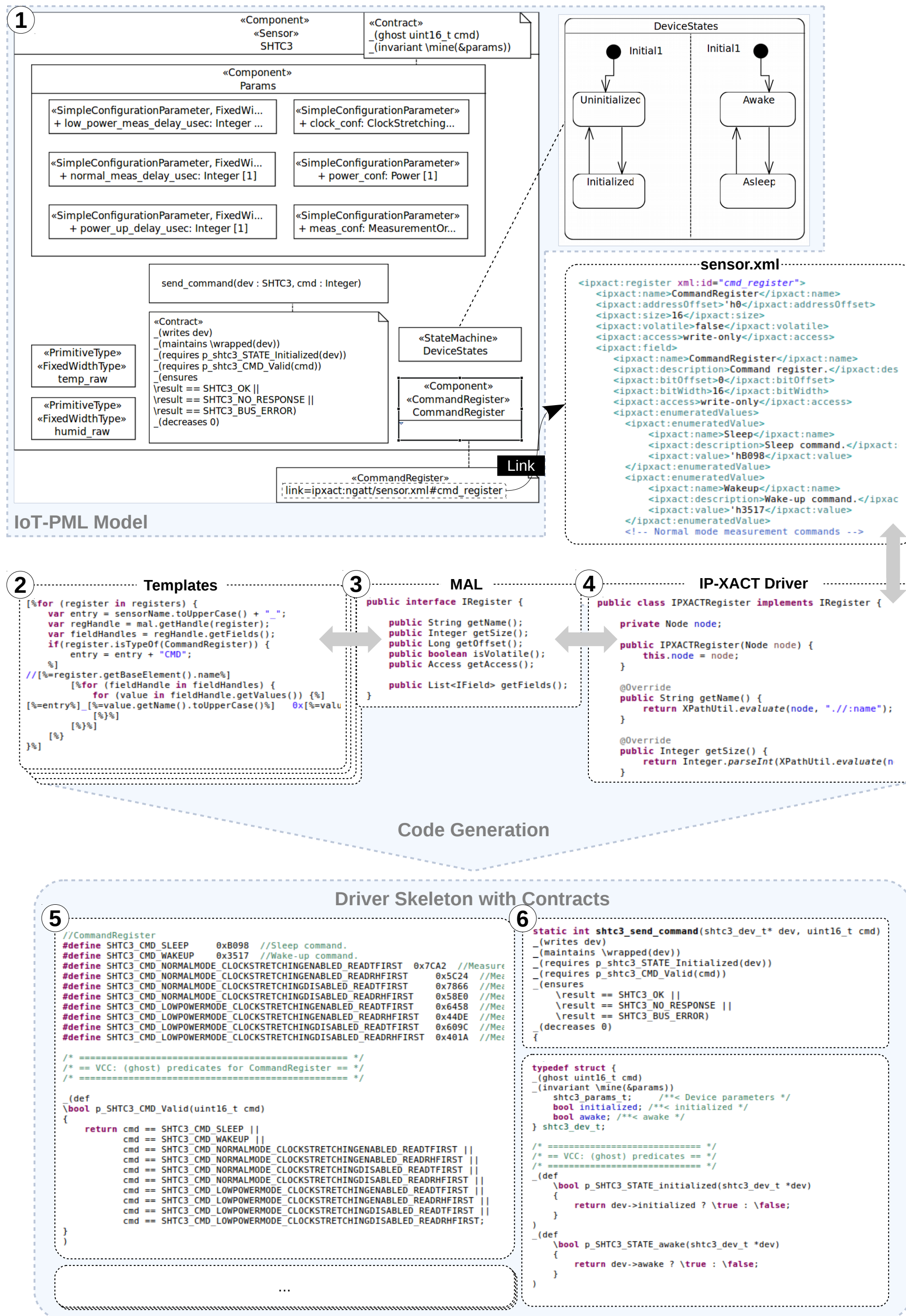

Fig. 7: Overview of the top-down use case.

states and device-related driver configuration parameters are modeled as *States* and ***ConfigurationParameters***, which are applied to *Properties*. User-provided functions are modeled as *Operations*. The properties and functions are annotated with ***Contracts***.

From this IoT-PML model, the basic skeleton of the COTS sensor driver for the RIOT OS is generated by a template-based approach utilizing the Epsilon Generation Language[8] (EGL). The skeleton comprises the entire driver infrastructure, including C headers and source code files. One header contains the device commands (5), which is generated by the EGL template depicted in (2). In addition, ghost predicates for the device are generated, which are ghost functions. In our example we use the ghost predicate `p_SHTC3_CMD_Valid` to check if a command, represented as a 16-bit unsigned integer, is one of the valid commands that can be written to the command register of the CTOS sensor. The template requests a handle on the ***CommandRegister*** from the MAL via the `getHandle()` method. The returned handle is an instance

[8]https://www.eclipse.org/epsilon/doc/egl/

of the Java interface `IRegister` (3), which is part of our implementation of the MAL for ***Registers***. The interface specifies a number of methods for data access, including methods for retrieving the ***Fields*** and ***Values*** of a given ***Register***. In order to create the handle, the MAL first resolves the ***MRef*** specified in the ***CommandRegister*** and dispatches the instantiation of an `IRegister` to the driver corresponding to the metamodel in the ***MRef***. In our case, the handle returned by the MAL is an instance of `IPXACTRegister` (4), which is the driver for ***Registers*** that enables access to data of register model elements stored in IP-XACT files. The implementation utilizes the Java API for XML processing (JAXP) and retrieves the data from registers using the XML Path Language[9] (XPath).

The generated C source code files contain function stubs (6) annotated with the contracts specified in the IoT-PML model. Furthermore, object invariants are included in the corresponding sections of the generated code. The generated driver skeleton contains the specifications for functions and objects. In our example the device state is represented by the `struct shtc3_dev_t` with an additional invariant that specifies that the device parameters represented by the `struct shtc3_params_t` are owned by the device itself. In addition, the ghost field `_(ghost uint16_t cmd)` is added to allow for reasoning about the last command that was sent to the device. To reason about the device state in function contracts, we introduce the ghost predicates `p_SHTC3_STATE_awake` and `P_SHTC3_STATE_initialized` that are automatically generated from the IoT-PML model.

The contract for the function `shtc3_send_command` states that the function will write to the argument `dev` using the VCC **writes** clause. The part `_(maintains \wrapped(dev))` of the function contract defines that the function guarantees that all invariants of the object `dev` are preserved. Additionally, the contract specifies that the device must be in the initialized state and that the argument `uint16_t cmd` must be a valid command. We realize this requirement by utilizing the aforementioned ghost predicates in the **requires** clause. The **ensures** clause of the contract guarantees that the function will always return one of the three possible return values: `SHTC3_OK, SHTC3_NO_RESPONSE, SHTC3_BUS_ERROR`.

With contracts in place, developers can then implement the device driver by filling out the function stubs. The implementation process can now be accompanied by an iterative static verification process using the VCC tool. Because the verification process is modular, not all function implementations must be available as long as the function stubs and function declarations are annotated with contracts. Due to the loose coupling provided by the model reference mechanism, the IoT-PML model is not affected by any modifications of the IP-XACT model, as long as the referenced model element is not deleted from the latter. This enhances co-design, as the IP-XACT and IoT-PML models can be worked on in parallel. If the IP-XACT model is modified, parts of the code need to be regenerated to accomodate the changes.

[9]https://www.w3.org/TR/xpath/all/

V. Related Work

There exists a large body of work that addresses the problem of heterogeneous metamodel integration. As DSMLs can be seen as individual viewpoints on the system under study [27], the topic also extends to model viewpoint integration. Some of the more recent approaches include facet-oriented modeling [28], model federation [24], and template-based metamodeling [29], which deal with metamodel heterogeneity on the syntactical, structural, and semantic level. Bruneliere et al. give a comprehensive survey of related approaches in [30]. Common to these works is that they are not focused on a particular domain, but instead provide generic mechanisms which we use as inspiration and technical foundation for the IoT-PML, as described in Section III-B. In addition, they typically lack MOF-compliancy, limiting access to existing model-based frameworks and tools.

For particular domains, there have been a number of efforts to address metamodel heterogeneity by creating unified modeling languages. For instance, the Unified Enterprise Modeling Language [31] unified different metamodels for the enterprise modeling domain. UML is also the result of such an effort in the software domain, as it unified the large number of different software modeling approaches and related methodologies that were in use in the early 1990s. Another example is WebDSL [32] which is an integrated modular language for web application development. To the best of our knowledge, there exists no such modeling language in the domain of ultra-thin software for resource-constrained IoT devices. General-purpose languages, such as UML have only very limited support for hardware-related aspects, even when used in conjunction with extensions such as MARTE. On the other hand, DSMLs, such as IP-XACT are focused on their domain and thus typically ignore aspects not directly required for the modeling task at hand. The unified metamodel for resource-constrained embedded systems proposed by Ziani et al. in [33] is a combination of a limited selection of related modeling languages, such as MARTE and SysML. However, their metamodel does not allow hardware modeling at the same level of detail as the IoT-PML. For instance, it is not possible to model hardware registers. Moreover, a model linkage mechanism is not provided.

VI. Conclusions and Future Work

We have presented the IoT-PML, a novel unifying modeling language for the development of firmware running on resource-constrained IoT devices. The language integrates the various DSMLs related to the IoT and embedded software domains by introducing concepts common to these DSMLs on multiple layers of abstraction and providing a reference mechanism that allows IoT-PML model elements to link to elements contained in DSML models. The potential benefits of this integration are considerable. Primarily, it allows model-based activities based on the IoT-PML to leverage data synergies between the different DSMLs, increasing the power of

code generators, optimizations, and analyses. Furthermore, the language simplifies co-design and coordination between users of the different DSMLs.

The language is in its early stages of development, which means the collection of topics for future work is extensive. Currently, the IoT-PML is implemented as a UML profile, allowing us to leverage the extensive software modeling capabilities of UML and the tooling ecosystem that has evolved around the language over the last decade. This implementation may be subject to change, as both the development of the language and the analysis of metamodels used in the IoT and embedded software domains is still ongoing. Should we deem the current implementation too restrictive in terms of expressiveness, we plan to implement the IoT-PML as a standalone MOF-compliant modeling language.

While the model reference mechanism and model abstraction layer can be used by tools running on IoT-PML models, there is currently very limited in-editor support for these features. At the moment, the model reference has to be manually put in and no indication inside the editor is given that the reference is valid. Furthermore, the data contained in the referenced model element are not shown in the editor. Similarly, feedback from analysis and optimization tools, such as the XML report generated by VCC, cannot be fed back to the IoT-PML model. How this data can be visualized and possibly edited from inside the editor is one of the primary subjects of future work.

Another related major topic is the consistency between the data in the referenced models and its representation in the IoT-PML. Although the coupling is relatively loose, consistency issues may still arise if the referenced model element is deleted or an attribute is modified that is reflected in the IoT-PML model. With respect to the particular use case we presented in this paper, there are extensive possibilities for future work in terms of general automation of the workflow processes. For instance, contracts are manually annotated in the model and require some a priori knowledge about the structure of the generated code when used in a top-down workflow. Future work may investigate the generation of contracts based on constraints expressed in the model using OCL.

References

[1] A. Thierer and A. Castillo, “Projecting the growth and economic impact of the Internet of Things,” https://www.mercatus.org/publication/projecting-growth-and-economic-impact-internet-things, 2015.

[2] P. Baker, S. Loh, and F. Well, “Model-driven engineering in a large industrial context - Motorola case study,” in MODELS 2005, pp. 476–491, 2005.

[3] J. Davies, J. Gibbons, S. Harris, and C. Crichton, “The CancerGrid experience: metadata-based model-driven engineering for clinical trials,” in in Sci. Comput. Program, vol. 89, pp. 126–143, 2014.

[4] M. Brambilla and P. Fraternali, “Large-scale model-driven engineering in a large industrial context - the WebML and WebRatio experience,” in Sci. Comput. Program, vol. 89, pp. 71–87, 2014.

[5] M. Wimmer and P. Langer, “A benchmark for model matching systems: the heterogeneous metamodel case,” in Softwaretechnik-Trends, vol. 33, 2013.

[6] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J. M. Jézéquel, and J. Gray, “Globalizing modeling languages,” in COMPUTER, vol. 47, pp. 68–71, 2014.

[7] J. Stecklein, “Error cost escalation through the project life cycle,” https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20100036670.pdf, NASA, Tech. Rep., 2004.

[8] X. Bellekens, A. Seeam, K. Nieradzinska, C. Tachtatzis, A. Clearyy, R. Atkinson, and I. Andonovic, “Cyber-physical-security model for safety-critical IoT infrastructures,” in WWRF35, 2015.

[9] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, “ThingML: a language and code generation framework for heterogeneous targets,” in MODELS 2016, pp. 125–135, 2016.

[10] B. Negash, T. Westerlund, A. M. Rahmani, P. Liljeberg, and H. Tenhunen, “DoS-IL: a domain-specific Internet of Things language for resource constrained devices,” in ANT 2017, pp. 416–423, 2017.

[11] D. Beaulaton, N. B. Said, I. Cristecu, R. Fleurquin, A. Legay, J. Quilbeauf, and S. Sadou, “A language for analyzing security of IoT systems,” in SoSE 2018, pp. 37–44, 2018.

[12] M. Adda and R. Saad, “A data sharing strategy and DSL for service discovery, selection and consumption for the IoT,” in EUSPN-2014, pp. 92–100, 2014.

[13] M. Hussein, S. Li, and A. Radermacher “Model-driven development of adaptive IoT systems,” in MODELS 2017, 2017.

[14] IEEE Standards Association, “IEEE 1685-2014 - IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows,” 2010.

[15] Object Management Group, “OMG Unified Modeling Language™ (OMG UML),” 2015.

[16] Object Management Group, “OMG Systems Modeling Language (OMG SysML™),” 2015.

[17] Object Management Group, “UML profile for MARTE: modeling and analysis of real-time embedded systems,” 2011.

[18] Object Management Group, “OMG Meta Object Facility (MOF) core specification,” 2016.

[19] Object Management Group, “Object Constraint Language,” 2014.

[20] T. Jürgen, “Hardware/software codesign: the past, the present, and predicting the future,” in Proc. IEEE, vol. 100, pp. 1411–1430, 2012.

[21] V. D’Silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” in IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 27, pp. 1165–1178, 2008.

[22] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, “VCC: a practical system for verifying concurrent C,” in TPHOLs 2009, pp. 23–42, 2009.

[23] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, “The Epsilon Object Language (EOL),” in ECMDA-FA 2006, pp. 128–142, 2006.

[24] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. Guychard, “Addressing modularity for heterogeneous multi-model systems using model federation,” in MODULARITY Companion 2016, pp. 206–211, 2016.

[25] Object Management Group, “UML24 - composite tags,” https://issues.omg.org/issues/UML24-50 (Last accessed: 2019/02/15).

[26] IEEE Standards Association, “IEEE 1471-2000 - IEEE recommended practice for architectural description for software-intensive systems”, 2000.

[27] A. Vallecillo, “On the combination of domain specific modeling languages,” in ECMFA 2010, pp. 305–320, 2010.

[28] J. de Lara, E. Guerra, J. Kienzle, and Y. Hattab, “Facet-oriented modelling: open objects for Model-Driven Engineering,” in SLE 2018, pp. 147–159, 2018.

[29] J. de Lara, E. Guerra, “Generic meta-modelling with concepts, templates and mixin layers,” in MODELS 2010, pp. 16–30, 2010.

[30] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer, “A feature-based survey of model view approaches,” in SoSyM 2017, pp. 1–22, 2017.

[31] F. Vernadat, “UEML: towards a unified enterprise modelling language,” in International Journal of Production Research, vol. 40, pp. 4309–4321, 2002.

[32] E. Visser, “WebDSL: a case study in domain-specific language engineering,” in GTTSE 2007, 2007.

[33] A. Ziani, B. Hamid, and S. Trujillo, “Towards a unified meta-model for resources-constrained embedded systems,” in SEAA 2011, pp. 485–492, 2011.