MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering

Federico Ciccozzi*
*School of Innovation, Design and Engineering
Mälardalen University, Västerås (Sweden)
Email: federico.ciccozzi@mdh.se

Romina Spalazzese†‡
†Department of Computer Science
‡Internet of Things and People Research Center
Malmö University, Malmö (Sweden)
Email: romina.spalazzese@mah.se

Abstract—**The Internet of Things (IoT) unleashes great opportunities to improve our way of living and working through a seamless and highly dynamic cooperation among heterogeneous Things including both computer-based systems and physical objects. However, properly dealing with the design, development, deployment and runtime management of this breathing network of Things means to provide solutions for a multitude of challenges. Among them, we focus on: supporting complexity and heterogeneity management, supporting collaborative development, maximising reusability of design artefacts, and providing self-adaptation of IoT systems.**

In this paper we propose Model-Driven Engineering (MDE) as a key-enabler for solving these challenges and supporting the lifecycle of IoT systems, from their design to runtime management. More specifically, we: (i) introduce MDE4IoT, a Model-Driven Engineering Framework supporting the modelling of Things and self-adaptation of Emergent Configurations of connected systems in the IoT; and (ii) show how MDE in general, and MDE4IoT in particular, can help in tackling the above mentioned challenges by providing the Smart Street Lights case that we use throughout the paper as concrete case.

I. Introduction

Nowadays, connectivity and technology are becoming more and more ubiquitous and affordable, respectively. Consequently, a growing trend is to connect everything that can benefit from being connected, from both digital and physical world. It is estimated by Cisco and Ericsson that, by 2020, 50 billions devices will be connected to the Internet and this estimated number is assumed to grow to 500 billions by 2030 with 5G. "The Internet-of-Things has the potential to change the world, just as the Internet did. Maybe even more so." [3].

The IoT includes a huge set of *heterogeneous Things*: sensors, actuators, more or less complex devices and computers, as well as physical objects that might be equipped with some of the previous elements. Things can represent almost anything one can imagine, from the simplest RFID tag to modern self-driving cars fully equipped with sensors and computers. An interesting characteristic of IoT systems is that heterogeneity embraces both hardware and software. More specifically, Things might be totally different among themselves in terms of both hardware and software, or similar in principle but independently developed and with slight differences in hardware and/or software. However, the very same software functionalities are expected to be deployable on different devices having only a limited set of core common features [25]. Another interesting characteristic to observe within the IoT is that *Things can be lightweight*, i.e., with very limited resources and computation capabilities.

Within the IoT, we call *Emergent Configuration* (EC) of connected systems a set of Things/devices with their functionalities and services that connect and cooperate temporarily to achieve a goal. From a user's point of view, an EC acts as one coherent system. Due to the *continuously evolving* character of the IoT, ECs can change unpredictably. For instance, Things might become unavailable because of physical mobility, or due to insufficient battery level or hardware problems.

To be able to provide the user with a coherent IoT system over time, there is the need to manage *runtime* changes of ECs. For instance, *adaptation* mechanisms that re-allocate a set of software functionalities from a faulty device to another offering similar features might be needed as part of the execution of proper analysis and planning. In order to maximise *reusability* of software functionalities (e.g., software components) and minimise the need of functional modifications in response to ECs changes, it would be beneficial to be able to design software functionalities *abstracting* away platform-specific details, which are instead inferred at deployment time. Doing so, adaptation entailing re-allocation of a software functionality would not require modifications to the functionality itself. Finally, to effectively enable collaborative development of IoT systems, mechanisms for supporting *separation of concerns* are needed.

As previously mentioned, the availability of a huge number and variety of heterogeneous devices within the IoT represents a great opportunity for improving our way of living and working if one could allow their seamless and timely communication, data exchange, and collaboration. Application areas that would benefit from the IoT innovation include for instance: smart living, energy, transportation, city, health and industry. However, to benefit from the great advantages that the IoT will unleash, a whole set of challenges needs to be dealt with at all levels. Heterogeneity, adaptability, reusability, interoperability, data mining, security, abstraction, separation of concerns, automation, privacy, middleware and architectures are just some examples of the aspects that need to be taken into account both at design time and at runtime and for which new methodological solutions shall be envisioned [4], [19], [23], [24], [26], [35].

Among the many challenges, we focus on: providing support for (1) high-level abstraction to address heterogeneity and system complexity, (2) separation of concerns for collaborative development, (3) automation for enabling runtime self-adaptation, and (4) reusability.

In this paper we propose Model-Driven Engineering (MDE) as a key-enabler for solving the aforementioned challenges in the IoT. Moreover, we aim at combining MDE's strengths and the vision of ECs as self-adaptive systems [14] to support design, development, and runtime management of IoT systems. More specifically, we (i) introduce MDE4IoT, a Model-Driven Engineering Framework supporting the modelling and self-adaptation of Emergent Configurations of connected systems; and (ii) show how MDE in general, and our MDE4IoT in particular, help to tackle the above mentioned challenges and to boost self-adaptation within IoT by providing the Smart Street Lights case that we exploit throughout the paper.

The reminder of the paper is organised as follows. Section II introduces the Smart Street Lights case. In Section III we propose the Model-driven Engineering Framework for Internet of Things while in Section IV we describe its application to our case. We present related works in Section V and provide a discussion about our contributions, together with conclusions, in Section VI.

II. The Smart Street Lights Case

Smart cities are one of the application areas of the IoT where everything, including cars, bikes, emergency vehicles, infrastructures and people, will be connected. In this context, we describe the "Smart Street Lights" demonstrator [30] (basic system) and extend it (scenario). Both its hardware and software have been designed, developed, and assembled through a collaboration between Malmö University and Sigma Technology and is part of the ECOS project [15] within the Internet of Things and People (IoTaP) Research Center [20].

Basic system. The core idea of the Smart Street Lights system is that every car, bike and pedestrian has its own sphere of light provided by a set of smart street lights (or lampposts). The size of this sphere, i.e., the number of lampposts that increase the brightness of their LED lights, is based on the vehicle's or pedestrian's speed and adapts to it at runtime. Yellow lights are dimmed down when nobody is around thus saving energy and red lights are switched on when someone is driving over the speed limit, increasing traffic awareness and safety. The smart lampposts handle all the sensing and computation in a distributed fashion -each lamppost runs the same code. Each lamppost can: (1) detect the presence of an "object" (car, bike, or pedestrian); (2) compute an object's speed; (3) increase and decrease the brightness of its own lights, either yellow or red; (4) compute the number of lampposts that should increase the brightness of their yellow light or turn on their red lights; (5) send and receive messages to and from neighbour lampposts.

Figure 1 illustrates a concrete instance of the Smart Street Lights in use: car A is traveling at normal speed and car B is approaching at very high speed (over the speed limit) thus triggering the red lights to turn on. In the dashed ellipse we can see the set of elements composing a lamppost: a pair of sensors (s_1, s_2), a pair of actuators for yellow and red lights (a_1, a_2), and a computation unit (c_1). A set of lampposts that temporarily connect and cooperate to form the sphere of light accompanying a road user is an EC. An interesting *emergent property* of this EC (shown in Figure 1) is that, when a car traveling over the speed limit is approaching another car, bike or pedestrian from the back, the latter gets a heads-up through the red lights.

Scenario. Car A is approaching a street segment where, all of a sudden, the red lights of four subsequent lampposts break (at runtime), and car B is approaching the same street segment while traveling at a speed over the limit. Due to the lampposts malfunction, both car A and car B would not be warned by the system, thus decreasing awareness and safety. To avoid this, the system needs to self-adapt, i.e., to modify itself at runtime to keep its properties. The infrastructures, including the lampposts, are grouped into areas, each having an Area Reference Unit (ARU) providing storage capacity and powerful computation capabilities. The ARU, not shown in Figure 1, takes care of the aforementioned faulty situation by initiating (i) a repair procedure and (ii) a system adaptation at runtime. Initiating the repair procedure for fixing the lights consists of sending a proper message with all the needed information to the service, including for instance the kind of malfunctioning objects and their location. Moreover, (ii) the ARU continuously monitors ECs and detects (a) car A traveling towards the malfunctioning lampposts by exploiting information from the car's navigation system, (b) car B approaching, and, thanks to the lampposts sensors, (c) car B's too high speed. Since the cars' navigation system are available resources, the ARU (d) sends a warning message (e.g., graphical, textual and/or acoustic) to both cars to reproduce the warning issued by lampposts' red light in normal conditions.

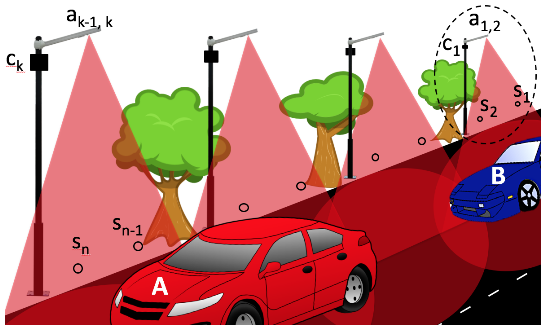

Fig. 1. An instance of the Smart Street Lights Case

III. MDE4IoT: A Model-Driven Engineering Framework for IoT

Figure 2 shows a high level model of our vision of IoT self-adaptive systems [14]. Starting from the bottom layer, we find heterogeneous Things, which are possibly self-adaptive. Each Thing is represented through both its software functionalities and its hardware platform. On the top layer there is a managing system implementing the MAPE-K loop [22]; a relevant aspect to mention is that such a system must have adequate (i) storage space and (ii) computation capabilities to support the management of the system and in particular of the lightweight Things. In the remainder of this section we describe the reasons behind the choice of MDE to tackle the many challenges related to design and runtime management of IoT self-adaptive systems, we introduce the Model-Driven Engineering Framework for IoT (MDE4IoT) and we show how it supports self-adaptation of ECs.

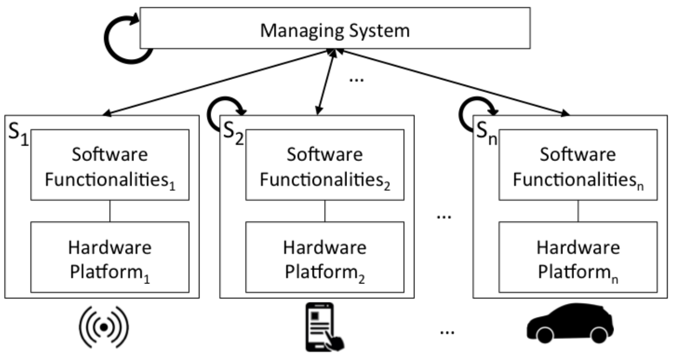

Fig. 2. IoT systems as Self-Adaptive Systems

Why MDE? In MDE, models represent the core concept and are considered an abstraction of the system under development. Rules and constraints for building models are described through a corresponding modelling language definition and, in this respect, a metamodel describes the set of available concepts and wellformedness rules a correct model must conform to [21]. Besides *abstraction*, a core pillar of MDE is the provision of *automation* in terms of model manipulation and refinement, which is performed through model transformations. A model transformation translates a source model to a target model while preserving their wellformedness [13].

Heterogeneity of software and hardware is at the same time a strength and a big challenge within the IoT [26]. Thanks to modelling languages, and more specifically domain-specific ones (DSMLs), MDE can provide unique means for the many aspects of *heterogeneous* systems to be represented all in one place. Models defined through these languages are meant to be much more human-oriented than common code artefacts, which are naturally machine-oriented. This means that, e.g., software can be defined with concepts that are not necessarily dependent on the underlying platform or technology. Doing so, the very same software functionality should be deployable on heterogeneous physical devices without modifications to enhance *reusability*. Platform-specificity would in fact be inferred by automated mechanisms (i.e., model transformations) in charge of executing models. The possible ways to generate execute models are usually three: (1) **interpretive**, (2) **translational**, and (3) **compilative** execution. Interpretive execution, where source models are executed through interpretation by a virtual machine or other kind of middleware; this is heavier than the other two both in computation and memory usage. Since many of the Things are embedded and have very limited resources, this solution is not always applicable. Translational execution (or code generation), where source models are translated into a third generation programming language (e.g., Java, C++) and then run on the target device after compilation or interpretation, is the one currently provided by MDE4IoT based on our validated code generator [10]. Compilative execution, where source models are directly compiled into a machine language, represents the most complex but at the same time most powerful option since it allows to exploit the many advantages and features of modern compilers without having to exploit intermediate semantically intricate translations from a high-level language (UML) to another (e.g., Java, C++). We are already working on possible solutions for providing this execution strategy in MDE4IoT. Even if more human-oriented than code, models can become complex and hard to grasp, even for experts. Especially when heterogeneity is constantly present, even within the same domain, mechanisms for properly rendering information in ways that are tailored to the specific developer are needed. MDE offers powerful instruments for combining multiple DSMLs to achieve the needed *separation of concerns*, and exploiting model transformations to support multi-view modelling, meant as the ability to define and render models from different design viewpoints. The IoT is a dynamic network of highly variable Things forming ECs. Besides the design and initial deployment of these complex systems, their *evolution at runtime* is a very challenging issue. This is particularly hard if operating on code-based artefacts. Imagine that a specific functionality is implemented for a specific physical device which, at a certain point, stops to be available. It would be hard to re-allocate the functionality to a different type of device without modifying the functionality itself; reusability of the functionality is hence undermined. In the following we describe how we address this.

MDE4IoT Framework. A graphical representation of MDE4IoT is shown in Figure 3. *Software functionalities* (i.e., deployable software components, SW_i) as well as *physical devices* or *platforms* (i.e., hardware components, HW_i) on which the software functionalities are meant to run, are modelled by means of a set of *modelling languages* (DSMLs). Deployment of software functionalities to physical devices in terms of *allocations* are modelled too. Note that physical devices can be represented at different granularity levels. For instance a physical device could be represented by a car navigation system, as a black-box with a specific set of available features and an operating system on which to run the allocated software functionalities. On the other hand, such a complex device could be divided into smaller pieces, such as sensors, actuators, and processing units. The granularity level to which the developer models physical devices depends on: (i) the purpose of the models and their intended use, and (ii) the capabilities of the involved model transformations that are in charge of deriving executable artefacts from them.

MDE4IoT is meant to exploit the combination of a set of DSMLs to achieve *separation of concerns*. In Figure 3 we can see what we call *horizontal* and *vertical viewpoints*. For instance, VP_1 represents a horizontal viewpoint related to a specific software application domain, while VP_2 represents a horizontal viewpoint for physical devices. VP_3 represents a vertical viewpoint since it embraces both software and hardware of a specific application domain. Different viewpoints are meant to be concurrently exploited by different domain experts and automated underlying mechanisms defined in terms of model transformations are meant to guarantee consistency among the different viewpoints [8]. Besides assistance at design time, MDE4IoT supports evolution scenarios within the IoT through self-adaptation mechanisms based on models and model transformations. More specifically, when an EC changes, and the managing system reasons about possible adaptations, two outcomes are possible: either (1) the affected executable artefacts can be directly re-deployed or (2) the system needs to first re-allocate the functionality at modelling level and then make executable artefacts run on alternative physical devices. The realisation of the managing system's reasoning is out of the scope of this paper where we rather focus on a framework that includes the features enabling self-adaptation.

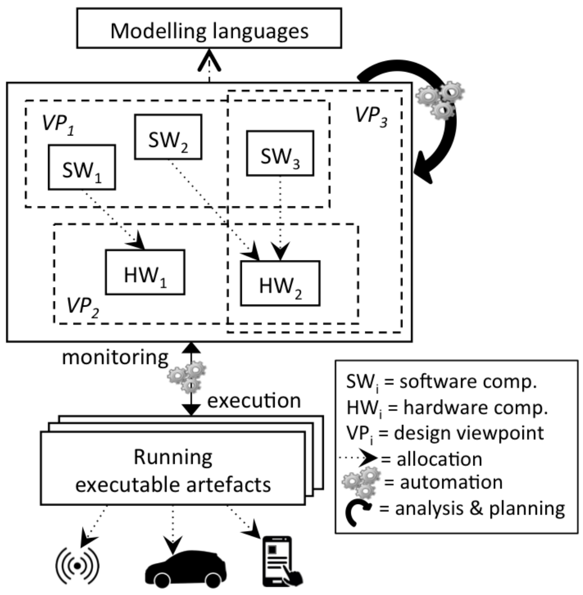

Fig. 3. MDE4IoT framework

IV. MDE4IoT applied to the Smart Street Lights

In this section, we instantiate MDE4IoT to support self-adaptation of ECs in the Smart Street Lights Case, both the basic system and the scenario. This will concretely highlight the usefulness and advantages of MDE4IoT. Among the many modelling languages that exist nowadays, UML is regarded as the de facto standard. Its wide adoption is partially motivated by its versatility, which enables (i) its usage as general-purpose language, and (ii) the possibility to customise it through the so-called profiling mechanisms [1] to give it domain specificity through domain-specific profiles. UML's latest incarnation (UML2), together with the standardisation of (i) the Semantics of a Foundational Subset for Executable UML Models (fUML), which gives a precise execution semantics to a subset of UML, and (ii) the Action Language for Foundational UML (ALF), to express complex execution behaviours in terms of fUML, has made UML a full-fledged implementation quality language [29]. Through (f)UML and ALF, the developer can fully describe the software functionalities of the system, while exploiting the UML profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [31] for modelling hardware components as well as allocations of software to hardware. Using UML to create domain-specific profiles based on a single metamodel helps MDE4IoT in providing and ensuring consistency among models in multiple viewpoints, in an hybrid multi-view fashion. If the various viewpoints leveraged different domain-specific languages, it would be much harder to ensure consistency since models would conform to different syntactical and semantic definitions. The exploitation of ALF as action language brings a set of advantages, such as easier model validation, analysis, and consistency checking. In fact, by expressing action code through third generation programming languages (e.g., Java, C/C++), the developer would infer assumptions on the target platform (e.g., memory management, parallelism, communication mechanism), which can hinder the generation of executable artefacts for different targets from the same input models, pivotal for MDE4IoT.

@Design time. In Figure 4 we can see a portion of the model representing the Smart Street Lights in a concrete graphical syntax. More specifically, the portion represents a single lamppost system in terms of software functionalities, physical devices and allocations. In terms of UML, `LampPost_Functional` represents the root software composite component, which contains 6 software components. Among them, `mY` of type `ManagerY` handles sensing and controls the lamppost's yellow light, and `mR` of type `ManagerR` handles sensing and controls the lamppost's red light. Both `mY` and `mR` are connected to `sf1, sf2` of type `SenseFunction` representing the sensing functionality of the two motion detectors. Moreover, `mY` is connected to `lf`, of type `LightFunction`, which represents a lightning functionality (i.e., on/off or dimmed yellow light), while `mR` is connected to `wf`, of type `WarnFunction`, which represents a warning functionality (i.e., on/off of the red light). Connections between software functionalities are achieved through connectors via ports. Behavioural descriptions of the software components are defined in terms of UML state-machines, for defining the overall behaviour by means of states and transitions, and ALF, for providing fine-grained actions. The state-machine diagram describing the behaviour of the type `ManagerR` is depicted in the upper right corner of Figure 4. A tiny example of ALF behaviour is provided in Listing 1. The ALF action code represents the state `Warn`'s *do activity*; the code is meant to make `wf` to display a warning.

```
{
   while(!this.toWarn)
     if(this.act1.sendWarning() == true)
       this.isWarned = true;
}
```

Listing 1. `Warn`'s *do activity* in ALF

The default physical devices of the Smart Street Lights case are modelled and grouped into the following two hardware components. `WarningSystemSensors` is composed of the two motion detection sensors `S1` and `S2` of type `MovementSensor` and stereotyped with MARTE's «HWSensor». `LampPost_Hardware` is composed of two actuators, `LPY` representing the lamppost's yellow light of type `LampPostYLight`, and `LPR` representing the lamppost's red light of type `LampPostRLight`, both stereotyped with MARTE's «HWActuator», and a computation unit `SC` of type `ComputationUnit_TypeA` and stereotyped with MARTE's «HWProcessor». Thanks to the attributes of the

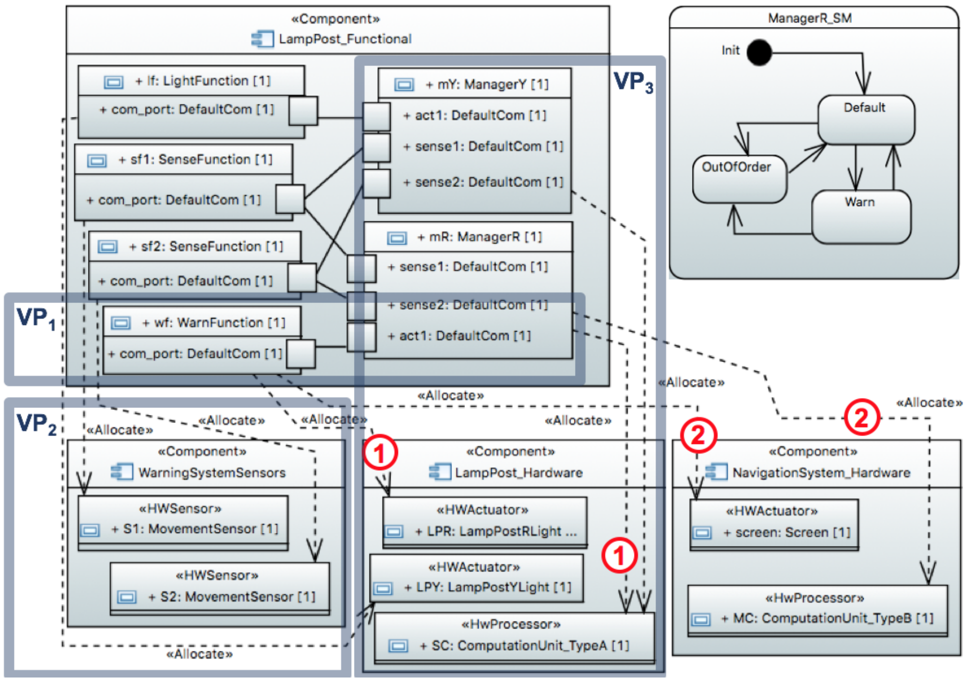

Fig. 4. Portion of the Smart Street Lights Model: allocation ① is for the basic system while ② the scenario

MARTE stereotypes we can specify needed information about the various devices. As an example for `SC` we specify its architecture as x86/x86-64 (using the attribute `architecture`) with only one core (using the attribute `nbCores`).
Software functionalities are allocated to hardware components through MARTE allocations, that is to say UML abstractions stereotyped as «Allocate». Software functionalities are transformed into executable artefacts by model transformation chains. The transformations, driven by the allocations defined in the model, are meant to infer platform-specific details to enable software functionalities to run on specific devices.
Moreover, in Figure 4 three possible different viewpoints are highlighted (due to the limited space we do not show them separately). More specifically, VP_1 represents a horizontal viewpoint for the development of the warning's logic, VP_2 for the physical motion sensors, and VP_3 represents a vertical viewpoint of actuators' logic and hardware. Different viewpoints are meant to be concurrently exploited by different domain experts and kept them synchronised by underlying mechanisms defined in terms of model transformations [8]
@Runtime. Car `A` is getting closer to a street segment where suddenly the red lights of four subsequent lampposts break, and car B is approaching the same segment while traveling over the speed limit (scenario). In our model this means that part of `LampPost_Hardware` suffers from a malfunction.
The ARU, our managing system (not shown in the model), initiates a repair procedure by sending a message to the maintenance service, including for instance the kind of malfunctioning objects and their location. Moreover, by continuously monitoring the IoT and ECs, the ARU detects car A and car B as well as their navigation systems as available resources and possible alternatives to replace the malfunctioning red lights.

In Figure 4, one of the two navigation systems is depicted. It is represented by `NavigationSystem_Hardware`, which is composed of an actuator `screen`, in terms of the navigator's screen, of type `Screen` and stereotyped with MARTE's «HWActuator», and a computation unit `MC` of type `ComputationUnit_TypeB`, stereotyped with MARTE's «HWProcessor»and set as an ARM architecture with two cores. MDE4IoT checks that `NavigationSystem_Hardware` provides the needed features for hosting the functionalities previously deployed on `LampPost_Hardware` that should be re-allocated: an actuator for showing warnings to replace the broken red light (`LPR`), and the routines to manage such an actuator to replace part of the intelligence provided by `mR`. Initially the lamppost's red light manager represented by `mR` and the warning functionality represented by `wf` are allocated to `LampPost_Hardware`'s actuator `LPR` and computation unit `SC` (① in Figure 4), respectively. MDE4IoT runs a feature-compatibility check[1] between `LPR` and `NavigationSystem_Hardware`'s actuator represented by `screen`, as well as between `SC` and `NavigationSystem_Hardware`'s computation unit represented by `MC`. If features are compatible, then MDE4IoT re-allocates `mR` from `SC` to `MC` and `wf` from `LPR` to `screen` (② in Figure 4). Re-allocations are meant to be performed automatically by specific in-place model transformations [13] that modify the source models. After the re-allocation has been successfully completed, MDE4IoT generates executable artefacts for the newly allocated devices. In case there is no compatibility of features MDE4IoT does not perform any re-

[1]Compatibility means that heterogeneous devices are considered interchangeable if they provide compatible features which can be modelled, e.g., by MARTE's constraints and/or stereotypes' properties.

allocation, notifies the Things about the incompatibility, and awaits for further notifications.

Clearly, ECs entail emergent and unknown devices that must be handled. The Things are in charge of providing the minimum information (amount of information can vary from case to case) about themselves so that they can be exploited by MDE4IoT and the provided model transformations exploited to re-allocate the affected software functionality and re-generate suitable executables that can be run on alternative devices. Re-allocation can be driven by an operator when human intelligence is required.

V. Related Work

The need of exploiting model-driven techniques to help the developer in designing applications for the IoT through separation of concerns and abstraction has been introduced by Patel et al. [26]. They provide automatic generation of an architecture framework and a vocabulary framework to help the developer to manually implement platform-specific portions. On the contrary we aim at supporting the developer by ensuring consistency among design viewpoints and fully generating and deploying executable artefacts, with the developer only focusing on the modelling activities. Several other works, such as [27], provide generation of skeleton code.

Conzon et al. [12] focus instead on a lower abstraction level, namely the platform and its software architecture. This approach targets a pretty specific type of IoT configurations, while we aim at providing a more generic solutions that can be instantiated in theoretically any type of configuration. Grace et al. [18] introduce the concept of lightweight interoperability models to monitor and check the execution of running software and quickly identify interoperability problems. While the approach is different in its purpose from ours, its concept of runtime interoperability models could be exploited by MDE4IoT for monitoring the IoT for emergent configurations.

In [32], the author proposes an open distributed architecture for the engineering of evolvable hybrid assembly systems. The architecture provides the basis for building frameworks to develop hybrid assemblies. Also in this case, our approach is meant to provide support for a wider set of activities rather than just assembling systems from hybrid components.

Chen et al. [7] present a runtime model-driven approach for IoT application development. While our approach focuses on detailed modelling of software functionalities and their allocation to physical devices, this approach employs models for describing sensor device models only.

Besides few approaches for IoT, the MDE community displays a notable amount of literature addressing adaptation, mostly based on models@runtime. Some approaches use models to specify self-adaptation in terms of mappings of assertions to adaptation actions [34] or to specify links between possible configurations [5]. Usually these models are leveraged to ease development by partially generating adaptation engines. Approaches that maintain runtime models for specifying adaptation and capturing feedback loop's knowledge exist too [2], [16], [17], [25], [28]. In [33], Vogel et al. provide an approach that enables the specification and execution of adaptation engines for self-adaptive software with multiple feedback loops.

The most novel characteristic of our approach, which cannot be found in any related work, is the ability to encompass several tasks of the lifecycle, from modelling, consistency assurance, and executables generation at design time, to self-adaptation due to evolution of ECs at runtime. Anyhow, we do not explicitly focus on the use of models for specifying self-adaptation but rather exploit models for the actual adaptation enactment. We plan to focus on how to exploit models for specifying the self-adaptation itself in the coming incarnations of MDE4IoT.

VI. Discussion and Conclusion

The Internet of Things (IoT) has a great potential for revolutionising our everyday life in all its aspects. Among other characteristics, the IoT is composed of an unprecedented combination of more or less complex and highly heterogenous constituents (i.e., Things) and it displays continuous evolution, thus leading to the need of methodological innovation.

In this paper we disclosed the opportunities provided by Model-Driven Engineering (MDE) to suffice this need. More specifically, we introduced the MDE4IoT framework to support modelling of Things and self-adaptation of Emergent Configurations in the IoT by exploiting: high-level abstraction and separation of concerns to manage heterogeneity and complexity of Things, to enable collaborative development, to enforce reusability of design artefacts, and automation, in terms of model manipulations, to enable runtime self-adaptation. The various parts of MDE4IoT have been validated individually: multi-view modelling for synchronised separation of concerns in [8], generation of executable artefacts for heterogeneous targets in [10], re-allocation and re-generation based on runtime feedback in [9], [11]. We are currently working on validating their synergy. Moreover, the Smart Street Lights case is a validated running prototype [30] of the ECOS project [15].

Self-adaptation of complex, heterogeneous, and variable systems-of-systems such the IoT, entails an endless set of complex issues to be tackled. The use of models and MDE can help out in this quest, but the way towards a full-fledged solution is far from unhindered. In this paper, we focused on the use of MDE for propagating adaptation from models to the running system. Clearly, other core features such as the use of models for adaptation, reasoning, and planning, as well as analysis of the possible impact of adaptation-triggered modifications on the unchanged parts of the system need to be investigated as well [6] as future works.

The incarnation of MDE4IoT presented in this paper is the outcome of our first engineering effort. We are already working on several enhancements with the goal of progressively and iteratively building an open framework for aiding design and self-adaptation of IoT systems.

Acknowledgments

This work is partially financed by the Knowledge Foundation (KKS) through the Internet of Things and People research profile (Malmö University, Sweden) and the SMARTCore project (20140051, Mälardalen University, ABB Corporate Research, Ericsson AB, Alten Sweden AB).

References

[1] A. Abouzahra, J. Bézivin, M. D. Del Fabro, and F. Jouault. A practical approach to bridging domain specific languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA*, volume 5. Citeseer, 2005.

[2] M. Amoui, M. Derakhshanmanesh, J. Ebert, and L. Tahvildari. Achieving dynamic adaptation via management and interpretation of runtime models. *Journal of Systems and Software*, 85(12):2720–2737, 2012.

[3] K. Ashton. That 'internet of things' thing. *RFiD Journal*, 22(7):97–114, 2009.

[4] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[5] N. Bencomo and G. Blair. Using architecture models to support the generation and operation of component-based adaptive systems. In *Software engineering for self-adaptive systems*, pages 183–200. Springer, 2009.

[6] A. Bennaceur, R. France, G. Tamburrelli, T. Vogel, P. J. Mosterman, W. Cazzola, F. M. Costa, A. Pierantonio, M. Tichy, M. Akşit, et al. Mechanisms for leveraging models at runtime in self-adaptive software. In *Models@ run. time*, pages 19–46. Springer, 2014.

[7] X. Chen, A. Li, X. Zeng, W. Guo, and G. Huang. Runtime model based approach to IoT application development. *Frontiers of Computer Science*, 9(4):540–553, 2015.

[8] A. Cicchetti, F. Ciccozzi, and T. Leveque. Supporting incremental synchronization in hybrid multi-view modelling. In *Models in Software Engineering*, pages 89–103. Springer, 2012.

[9] F. Ciccozzi, A. Cicchetti, and M. Sjödin. Round-Trip Support for Extra-functional Property Management in Model-Driven Engineering of Embedded Systems. *Information and Software Technology*, 2012.

[10] F. Ciccozzi, A. Cicchetti, and M. Sjödin. On the Generation of Full-fledged Code from UML Profiles and ALF for Complex Systems. In *Procs of ITNG*, February 2015.

[11] F. Ciccozzi, M. Saadatmand, A. Cicchetti, and M. Sjödin. An Automated Round-trip Support Towards Deployment Assessment in Component-based Embedded Systems. In *Procs of CBSE*. ACM, 2013.

[12] D. Conzon, P. Brizzi, P. Kasinathan, C. Pastrone, F. Pramudianto, and P. Cultrona. Industrial application development exploiting IoT vision and model driven programming. In *Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on*, pages 168–175. IEEE, 2015.

[13] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Procs of OOPSLA*, 2003.

[14] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke. Software engineering for self-adaptive systems: A second research roadmap. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475, pages 1–32. Springer-Verlag, 2013.

[15] Emergent Configurations of Connected Systems (ECOS). http://iotap.mah.se/ecos/, [Accessed: 2016-01-21].

[16] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software, IEEE*, 23(2):62–70, 2006.

[17] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[18] P. Grace, B. Pickering, and M. Surridge. Model-driven interoperability: engineering heterogeneous IoT systems. *annals of telecommunications-annales des télécommunications*, pages 1–10, 2015.

[19] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.*, 29(7):1645–1660, Sept. 2013.

[20] Internet of Things and People (IoTaP) Research Center. http://iotap.mah.se/, [Accessed: 2016-01-21].

[21] S. Kent. Model Driven Engineering. In *Third International Conference on Integrated Formal Methods (iFM)*, 2002.

[22] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[23] H.-D. Ma. Internet of things: Objectives and scientific challenges. *Journal of Computer science and Technology*, 26(6):919–924, 2011.

[24] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of things. *Ad Hoc Netw.*, 10(7):1497–1516, Sept. 2012.

[25] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, pages 122–132. IEEE Computer Society, 2009.

[26] P. Patel and D. Cassou. Enabling high-level application development for the Internet of Things. *Journal of Systems and Software*, 103:62–84, 2015.

[27] F. Pramudianto, I. R. Indra, and M. Jarke. Model Driven Development for Internet of Things Application Prototyping. *Book Model Driven Development for Internet of Things Application Prototyping (Knowledge Systems Institute Graduate School, 2013, edn.)*, 2013.

[28] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software engineering for self-adaptive systems*, pages 164–182. Springer, 2009.

[29] B. Selic. The Less Well Known UML. In *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2012.

[30] The Smart Street Lights Demonstrator. https://vimeo.com/137837738/, [Accessed: 2016-01-21].

[31] The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. http://www.omgmarte.org/, [Accessed: 2014-11-28].

[32] K. Thramboulidis. An Open Distributed Architecture for Flexible Hybrid Assembly Systems: A Model Driven Engineering Approach. *arXiv preprint arXiv:1411.1307*, 2014.

[33] T. Vogel and H. Giese. Model-driven engineering of self-adaptive software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):18, 2014.

[34] J. White, D. C. Schmidt, and A. Gokhale. Simplifying autonomic enterprise Java Bean applications via model-driven engineering and simulation. *Software & Systems Modeling*, 7(1):3–23, 2008.

[35] L. D. Xu, W. He, and S. Li. Internet of things in industries: A survey. *EEE Trans. Industrial Informatics*, 10(4):2233–2243, 2014.