

Weaving Rules into Models@run.time for Embedded Smart Systems

Ludovic Mouline
Thomas Hartmann
Francois Fouquet
Yves Le Traon

University of Luxembourg
firstName.lastName@uni.lu

Johann Bourcier
Olivier Barais
Universite de Rennes 1
firstName.lastName@irisa.fr

ABSTRACT

Smart systems are characterised by their ability to analyse measured data in live and to react to changes according to expert rules. Therefore, such systems exploit appropriate data models together with actions, triggered by domain-related conditions. The challenge at hand is that smart systems usually need to process thousands of updates to detect which rules need to be triggered, often even on restricted hardware like a Raspberry Pi. Despite various approaches have been investigated to efficiently check conditions on data models, they either assume to fit into main memory or rely on high latency persistence storage systems that severely damage the reactivity of smart systems. To tackle this challenge, we propose a novel composition process, which weaves executable rules into a data model with lazy loading abilities. We quantitatively show, on a smart building case study, that our approach can handle, at low latency, big sets of rules on top of large-scale data models on restricted hardware.

KEYWORDS

Models@run.time, Reactive systems, Rule engines, Lazy loading, Smart systems, Embedded Systems

ACM Reference format:

Ludovic Mouline, Thomas Hartmann, Francois Fouquet, Yves Le Traon, Johann Bourcier, and Olivier Barais. 2017. Weaving Rules into Models@run.time for Embedded Smart Systems. In *Proceedings of Modularity in Modelling Workshop, Brussels, Belgium, April 2017 (MOMO'17)*, 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND MOTIVATION

To make sustainable decisions and to take appropriate actions, smart systems need to continuously analyse their context, *i.e.*, their environment and internal state [20, 21]. For example, a smart building can contain hundreds of devices that continuously generate data which contribute to an understanding of the context. The interest of end users with such systems is, for example, to know the current state of the building, *e.g.*, the current temperature, or to remotely switch on/off a heating system. To do so, such systems often need to process thousands of data updates—*e.g.*, sensor

values—per second in near-time and on hardware with limited computational capabilities, like a Raspberry Pi [32]. Processing these updates mostly consists in a verification against a set of domain-defined conditions that can trigger specific actions [34]. Which actions need to be triggered for which updates, *i.e.*, for which *patterns*, can be defined in so-called *rules*. Performance of processing these rules is key critical in such domains, since it directly defines the reactivity level of a smart system. This applies even more when considering security rules that need to aggregate various data to fire the relevant counter-actions. For instance, in smart building, various temperature sensors (*e.g.*, indoor and outdoor) need to be correlated to detect that a door is open and to ultimately fire an alarm. Thus, independent time series are not suitable to correlate data from different data sources. Instead, efficiently correlating data relies on navigable data structures [21]. The models@run.time paradigm [12, 27] has proven its suitability to represent the context of such systems and to provide a navigable structure for reasoning engines. To accurately reflect a current system context, models@run.time are regularly updated, *e.g.*, with sensor measurements. Smart systems need mechanisms to process these updates and trigger actions based on pattern detection—for many domains, in near real-time.

Several approaches try to address the challenge of live processing updates to detect which actions need to be triggered. *Complex Event Processing* (CEP) [26] investigates how to detect predefined *events*, *i.e.*, particular patterns, like sequences of specific values. Others suggest to use rule engines, like Drools [13]. In the home automation domain, rule engines like IFTT [3], openHab [4] and Pimatic [5] are used to automate actions. For patterns on complex structures, OCL-like queries have been defined on top of MOF-based models [8]. All these approaches, model-based or not, rely on that rules and data models fit completely into main memory or on high latency persistence storages, which severely damage the reactivity of smart systems. This leads to limitations for systems, which need to process large amounts of rules on limited hardware. To address these limitations, we propose to combine a set of *if <pattern(context)>then <actions>* within models@run.time structure with lazy loading abilities. By combining lazy loading mechanisms with a low-latency persistent storages, we do not assume that rules or models must fit completely into main memory. More specifically we investigate the following research questions:

Listing 1: ANTLR grammar of the rules language

```

metamodel: ruleDef*;
ruleDef: 'rule' STRING condition action 'end';
condition: 'when' ('not')? (term op term);
term: (type '.' attribute) | NUMBER | STRING;
op: ( '=' | '>' | '>=' | '<' | '<=' | '!=' );
type: IDENT ('.' IDENT)*;
attribute: IDENT;
action: 'then' task;
task: operation ('.' operation)*;
operation: IDENT ('(' (value(',' value)*)? ')');
value: STRING | '{' task '}' ;

```

- **RQ1:** How can we process rules, on limited hardware, with nearly constant memory, regardless of the model size?
- **RQ2:** Despite the lazy loading mechanism, can we obtain sufficient latency to enable near real-time process?

The remainder of this paper is as follows. Section 2 describes our contribution, Section 3 its evaluation, Section 4 discusses related work and Section 5 concludes the paper. Background is explained in the sections when needed.

2 WEAVING RULES INTO MODELS@RUN.TIME

In this section we detail our rule language and weaving process to combine rules and models@run.time.

2.1 Language Definition

To weave rules into models@run.time, we leverage two kinds of input: one for the definition of the data structure and another one for defining rules. For the data structure, we reuse common meta-model formalisms, such as defined by MOF and implemented by EMF/Ecore [14]. A data model is a set of classes, which contain a set of attributes and references to other classes. To meet models@run.time requirements, we use the Kevoree Modeling Framework (KMF) [18, 19], which has been specifically designed for this purpose. Using a textual syntax, KMF allows to define data structures with built-in lazy loading mechanisms, used for this approach in the processing engine (*cf.* Section 2.4). For the second kind of input, rule definitions, we reuse state-of-art rules modelling concepts, where each rule is composed of: a *name*, a *condition*, and an *action*. The grammar of our rule modelling language is depicted in Listing 1. This language is inspired by the *when <condition> then <actions>* pattern.

Conditions allow to specify two things: *i*) to which class a rule is attached, we will refer to this as the context of the rule, and *ii*) the condition of the context to trigger the execution of an action, *i.e.*, a condition on values. The current version of our rule language can only define rules based on single attributes. We intent to extend this in future work, however this is out of scope of this paper, which focus on the performance and lazy-loading impact of rule engines. For rule action definition, we reuse the formalism proposed by Gremlin [2], which has proven its expressiveness to define a

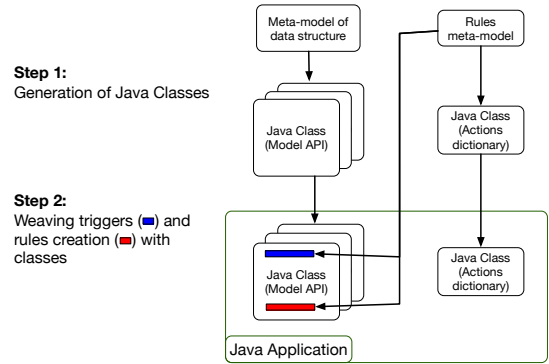


Figure 1: Compilation process

flow of processing actions on graph structures. Our rule action can be regarded as a pipeline, where each action is chained and propagates results to the next one. We provide actions for four kinds of operations: *i*) to navigate in the model, *ii*) to manipulate the control flow, *e.g.*, an *if* statement, *iii*) to manipulate the result, *e.g.*, a filter, *iv*) to manipulate variables, like saving results in a variable. Currently, all rules have its own graph condition. In future work we plan to use approaches like Rete [17], which suggest to share condition trees.

An example rule is shown in Listing 2, relying on a simple meta-model where a class *Room* has a relationship to a class *HeatingSystem*, containing an attribute *status*. The then block considers the current room as the starting point to chain actions, such as *traverse*. As effect, this simple rule activates the heating system when the temperature is below 18 degree. To trigger actions, such rules just modify the current model itself by expecting a synchronisation by a models@run.time engine. In addition we support arbitrary action code through injection of lambda functions within the chain of actions.

Listing 2: Rule example

```

rule "SwitchOnHeatingSystem"
when
    building.Room.temperature < 18
then
    relation('heatingSystem')
    .setAttribute("status", "on")
end

```

2.2 Rule Action Compilation Process

In this section, we describe the weaving process to inject executable rules during the generation process of KMF. The result of this weaving process is a standalone artefact, which contains Java classes ready to be used as a models@run.time backbone. An overview of this is shown in Figure 1.

KMF generates, based on a meta-model, a set of classes, which are referred to as *model API*. This API allows to manipulate data compliant to a meta-model during execution: creating new model elements, modifying existing ones, or navigating inside the model. Every KMF model element is lazily

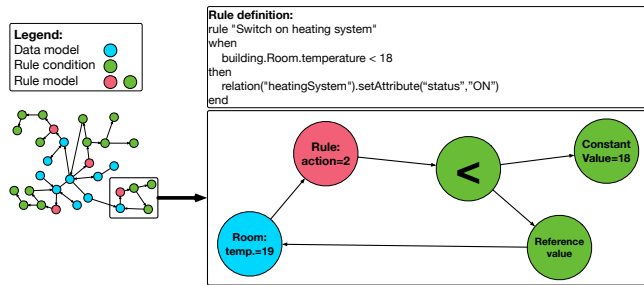


Figure 2: Model of a rule condition

loaded during the execution when relationships are traversed. Once elements are modified in-memory, KMF persists them to disk using Key-Value storages, such as RocksDB. To automatically trigger the verification of rules, setter methods of the KMF API are overridden for classes concerned by any rule. This generation process can be seen on the upper-left part of Figure 1 and is divided in two steps. The first one, depicted on the upper-right part of Figure 1, consists in the creation of one Java class containing all actions, called *Actions dictionary*. Actions are compiled into static Java code, identified within the dictionary with an integer value. This way, when a rule condition triggers an action, the system can simply hit the dictionary with a previously stored integer reference to execute the corresponding action code. The second step aims at generating the trigger code to verify automatically rules. Because conditions are always based on model element value updates, we override the model API setter to trigger all rule conditions related to this particular class and attribute. This is depicted as blue rectangle in Figure 1. As a result, we obtain a standalone Java artefact, embedded all actions as static methods automatically called when KMF elements are modified through an extended setter. This, together with the KMF lazy loading mechanism, allows to workaround the need to keep every model element in memory to listen for updates.

2.3 Weaving Condition Trees and Models

In this Section, we describe how rule conditions are woven into the model. Like most object-oriented modelling frameworks, KMF uses a graph-based approach to model a system, *i.e.*, the model can be seen as a graph of interacting objects, where the graph structure conforms to the meta-model. Each node in the graph conforms to a meta-class and is editable and accessible using the model API, generated as explained in Section 2.2. The node related to the KMF classes are referred to as data node. These nodes are depicted in blue in Figure 2. For rules, we use two parts: a *rule node* and a *condition graph*. A rule node represents a rule, stores actions, and has a relationship to the graph condition. As mentioned in the previous section, actions are compiled into static fields, using integers as identifiers. We store this identifier as an attribute in a rule node. Rule nodes are depicted in red in Figure 2.

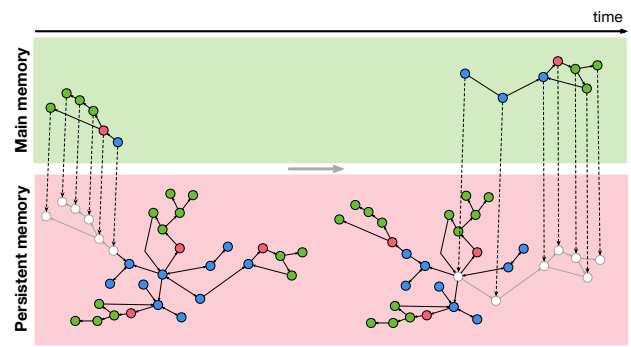


Figure 3: Lazy loading during graph navigation

Moreover, in Section 2.1, we explained that a rule first defines a context, *i.e.*, the class for which the rule should be applied. A rule node is created for each data node, implied in a rule as context and that conforms to a KMF class. These two nodes are linked, to first enable an efficient rule verification and secondly to set the input of the first operation with the data node. Additionally, rule nodes contain an integer reference to an executable action present in the dictionary.

Rule conditions are modelled using a graph, which represents its *Abstract Syntax Tree (AST)*. For each different term and operator, there is a specific type of nodes. Currently, we defined twelve different nodes for this AST—not all of them are currently used in the rule language, *i.e.*, statements for the rule condition: boolean operator (and, or, not), arithmetic operator ($=$, \neq , $>$, $>=$, $<$, $<=$), constant value and reference value, that refers to the value of another node. The AST can be modelled using these nodes. An example is depicted in green in Figure 2. On the upper-right part of the figure, in the rectangle, the result of the compilation of the rule is shown in Listing 2. On this graph, we can see the second composition of rule nodes and data model nodes: the condition node *Reference value* has a relationship with the node *Room*, which is part of the data model.

2.4 Rule Processing

In this section, we describe how we traverse the model, woven from the data structure definition and rule definition, using lazy loading techniques. When an attribute, which is part of a rule definition, is modified, we navigate through the relations of a rule, process the condition graph and, if the condition is validated, get the action using its identifier and executed it. This process uses lazy loading techniques to dynamically load the necessary node on demand into main memory [22]. When a data model node is accessed, the system first looks into the main memory if the node is present. If not, it will load it from a persistent storage, and vice versa, stores unused nodes if needed in order to free memory. Figure 3 shows the processing of two rules, with a memory size that can contain six elements. The upper zone describes model elements that are loaded in main memory, while the lower zone shows a view on a composed graph.

To process the condition graph AST, we leverage a classical binary tree interpretation. Each node involved in the condition AST has at most two children. If a node has at least one children, it means that it is an operator. There is only one exception: the *node value* that returns the value of an attribute of a data node. A node value is defined by two things: a relation to the target node and the name of the attribute stored as an attribute. For instance, in the example depicted in Figure 2, the reference node has a relationship to a room node and store the attribute name 'temperature' as its own attribute. Processing such nodes mean: resolve the node of the relationship and return the value stored in the attribute, in our example the temperature attribute. To compute the value of an operator node, the system needs to get the values of its children and apply the semantic of the operator. For nodes without children, the node returns the stored value or applies a process to compute or get one.

3 EVALUATION

To answer the two research questions formulated in Section 1, we conducted in this section an experimental evaluation. Source code is publicly available on GitHub¹. For all experiments we rely on a smart building dataset with simulated sensor values.

RQ1: processing rules with constant memory. In our approach, we defined a combination of model and condition AST with lazy loading abilities. As a result, we should be able to run with an arbitrary size of memory, regardless of the actual model size. If rules are sequentially processed, the memory limitation is given by the number and size of nodes implied in one rule (condition AST, rule node, and related data node). In case rules are processed in parallel, the memory requirement is to fit at least the rule with the largest number of nodes implied for all threads. During this experiment, we fix the cache size of KMF to 10,000 elements to force the lazy loading mechanism to mainly work from disk. Moreover, the model size increases from 100,000 elements to 5,000,000 elements. During each iteration, we sequentially check all rule conditions and trigger those that are evaluated to true. Figure 4 shows that memory consumption stays constant and below 50MB all along the process. From these results we can conclude that our approach allows to process a massive model even with limited memory.

RQ2: latency of rule checking using lazy loading. Because our approach relies on a lazy loading mechanism with a persistence storage, we drastically reduce the memory usage, as shown in RQ1. However, these benefits come at the price of potentially decreasing the latency of the rule engine. We conducted a second experiment to quantify the latency of our approach against various rule sizes and numbers. We setup a model of 1 million elements (simulating sensor values) and a RocksDB storage. Then, we measured three batches of verification with rules, containing small to large conditions to check. For every batch we evaluated 100,000 rules and

Table 1: Processing throughput (rules/seconds)

Condition rule size	Throughput
3	70,028
31	58,788
255	41,152

measured the total execution time: time to load nodes, process the AST condition and execute the action. As there is a direct link between the execution time of an action and the process time of a rule, we do not modify this parameter in our study. Results are presented in Table 1 in rules per second processed. From these results we can conclude that despite the lazy loading mechanism and less than 200MB memory allocated for the processing JVM the throughput is still above 50,000 rules per seconds in average. Such an engine can be embedded in devices like a Raspberry Pi, with less than 500MB memory.

4 RELATED WORK

The execution of rules on top of models has been previously discussed by the modelling community. For example, to handle model to model or model to text transformations. In [30], Varró *et al.*, define rules as a set of three elements: a graph pattern to look for, a set of application conditions, and a graph result. Bergmann *et al.*, [10] proposed EMF-IncQuery, a model transformation framework, based on graph pattern matching, for big models. Other solutions which have been suggested in the context of model transformations, are the ATLAS Transformation Language (ATL) [23], Henshin [11], and Jouault *et al.*, [24]. To enable live processing of rules, [16] analyses model modifications using a CEP engine. These approaches require that all data and rules are fully in-memory, whereas our solution loads only the currently processed elements on demand into main memory. Furthermore, for these approaches, all rules are stored aside of the model, whereas our solution suggests to combine model and rules. Textual OCL [33] is also related to our approach. It allows to define model constraints and derived attributes. Approaches [9] have been investigated to check model constraints during the execution of a system, *e.g.*, in [7], the authors propose a solution to generate Java Modelling Language (JML) [25] assertions, a language to specify pre and post conditions on top of Java methods, from OCL. These approaches are made for model checking, whereas we propose a solution for rules, *i.e.*, they do not support to execute actions. Another approaches consist in providing OCL interpreter [28]. This approach executes the constraints on model snapshots, which are regularly taken and cannot process the events in a short amount of time. Another application for rules is goal modelling, where goal models represent goals and scenarios of a system with languages like URN [6]. [29] and [31] define an approach to combine rule engines with goal modelling techniques. These approaches rely on an external rule engine. To efficiently query large models, several approaches have been

¹<https://github.com/lmouline/momo17-bench>

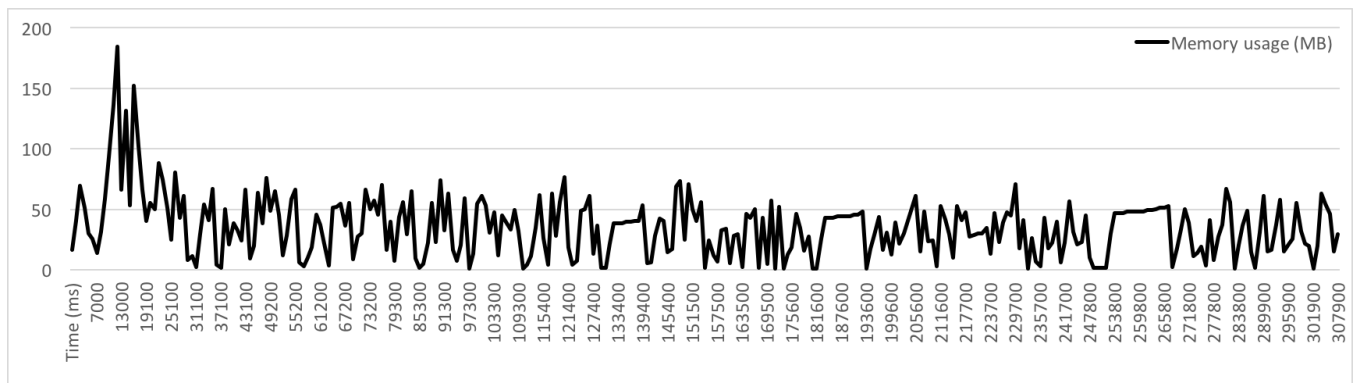


Figure 4: Results for RQ1

investigated. EMF-Query [1] defines an API to access model elements. To address its difficulty to deal with large models, they propose a new version, EMF-Query 2, that can lazy load the model element from persistent memory. Recently, a new approach has been proposed to query large models efficiently: the Mogwai framework [15]. OCL constraints are compiled to Gremlin [2] and then executed at the database level. This approach allow to generate a Gremlin request from OCL constraints. Contrary to our approach where our rules are directly executed at the main memory level, this approach implies that the request are executed at the database level. Furthermore, the approach defends in [2] and the one explains in [1] have only been designed to query large models, thus cannot execute actions.

5 CONCLUSION AND PERSPECTIVE

Executing rules or queries on large data models is still an open challenge in the modelling community, especially when executed on limited hardware. Solutions proposed so far, mostly rely on loading the full model and all rules in memory or push the problem to the database layer, resulting in high latency. In this paper we presented a novel approach to weave rules into models@run.time using a lazy loading mechanism, in order to deal with the execution of rules on large-scale models. We claim that most rules do not require the full data model but only relatively small parts. Therefore, we first proposed to weave rule models into data models. Secondly, we used a lazy loading mechanism to load/unload required data on demand. We integrated this approach into the Kevoree Modeling Framework and showed that it can handle thousands of rules, combined in a large model, with a small and constant memory consumption.

We plan to extend this approach in future work on several aspects. Firstly, we want to provide a more expressive language to define rules and manage conditions that are based on more than one attribute. Our approach is restricted to one attribute. To extend it to several attributes, we intend to explore a publish-subscribe-based system combined with a buffer-based system. The Pub-Sub system should be able to notify a rule node that a specified attribute has changed.

The buffer-based system should allow to synchronise different updates coming from several attributes, belonging to one or different nodes. Secondly, we aim at improving the memory required to store the model by using a Rete like approach to represent conditions. And finally, we want to introduce temporal aspects in rules.

ACKNOWLEDGMENTS

This work was supported by POST Luxembourg.

REFERENCES

- [1] 2017. EMF Query. (6 February 2017). <https://projects.eclipse.org/projects/modeling.emf.query>
- [2] 2017. Gremlin. (9 February 2017). <http://gremlin.tinkerpop.com>
- [3] 2017. IFTT. (5 February 2017). <https://ifttt.com/>
- [4] 2017. openHAB. (5 February 2017). <http://docs.openhab.org/tutorials/beginner/rules.html>
- [5] 2017. pimatic. (5 February 2017). <https://pimatic.org/>
- [6] 2017. URN specification. (6 February 2017). <http://www.itu.int/rec/T-REC-Z.151-201210-1/en>
- [7] Carmen Avila, Guillermo Flores, and Yoonsik Cheon. 2008. A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking. In *Proceedings of the 2008 International Conference on Software Engineering Research & Practice, SERP 2008, July 14-17, 2008, Las Vegas Nevada, USA, 2 Volumes*. 403–408.
- [8] Carmen Avila, Amritam Sarcar, Yoonsik Cheon, and Cesar Yeep. 2010. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010), Redwood City, San Francisco Bay, CA, USA, July 1 - July 3, 2010*. 393–398.
- [9] Carmen Avila, Amritam Sarcar, Yoonsik Cheon, and Cesar Yeep. 2010. Runtime constraint checking approaches for OCL, a critical comparison. (2010).
- [10] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh, and András Ökrös. 2010. Incremental evaluation of model queries over EMF models. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 76–90.
- [11] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. 2008. Precise semantics of EMF model transformations by graph transformation. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 53–67.
- [12] Gordon S. Blair, Nelly Bencomo, and Robert B. France. 2009. Models@ run.time. *IEEE Computer* 42, 10 (2009), 22–27.
- [13] Paul Browne. 2009. *JBoss Drools business rules*. Packt Publishing Ltd.
- [14] Frank Budinsky. 2004. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional.

- [15] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. 2016. Mogwai: A framework to handle complex queries on large models. In *Tenth IEEE International Conference on Research Challenges in Information Science, RCIS 2016, Grenoble, France, June 1-3, 2016*. 1–12. DOI:<http://dx.doi.org/10.1109/RCIS.2016.7549343>
- [16] István Dávid, István Ráth, and Dániel Varró. 2014. Streaming model transformations by complex event processing. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 68–83.
- [17] Charles L Forgy. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence* 19, 1 (1982), 17–37.
- [18] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 2012. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*. 87–101.
- [19] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 2014. Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use. *CoRR* abs/1405.6817 (2014). <http://arxiv.org/abs/1405.6817>
- [20] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. 2014. A Native Versioning Concept to Support Historized Models at Runtime. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*. 252–268.
- [21] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. 2014. Reasoning at Runtime using time-distorted Contexts: A Models@run.time based Approach. In *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013*. 586–591.
- [22] Thomas Hartmann, Assaad Moawad, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. 2015. Stream my models: Reactive peer-to-peer distributed models@run.time. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*. 80–89.
- [23] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Science of computer programming* 72, 1 (2008), 31–39.
- [24] Frédéric Jouault and Massimo Tisi. 2010. Towards Incremental Execution of ATL Transformations. In *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*. 123–137.
- [25] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 1–38.
- [26] David Luckham. 2008. The power of events: An introduction to complex event processing in distributed enterprise systems. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*. Springer, 3–3.
- [27] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. 2009. Models@ Run.time to Support Dynamic Adaptation. *IEEE Computer* 42, 10 (2009), 44–51.
- [28] Mark Richters and Martin Gogolla. 2000. Validating UML Models and OCL Constraints. In *"UML" 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*. 265–277.
- [29] William N. Robinson. 2008. Extended OCL for Goal Monitoring. *ECEASST* 9 (2008). <http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/105>
- [30] Dániel Varró, Gergely Varró, and András Pataricza. 2002. Designing the automatic transformation of visual languages. *Science of Computer Programming* 44, 2 (2002), 205–227.
- [31] Mira Vrbaski, Gunter Mussbacher, Dorina Petriu, and Daniel Amyot. 2012. Goal models as run-time entities in context-aware systems. In *Proceedings of the 7th Workshop on Models@ run.time*. ACM, 3–8.
- [32] Vladimir Vujović and Mirjana Maksimović. 2015. Raspberry Pi as a Sensor Web node for home automation. *Computers & Electrical Engineering* 44 (2015), 153–171.
- [33] Jos B Warmer and Anneke G Kleppe. 2003. *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional.
- [34] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance Complex Event Processing over Streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*.