Runtime modeling and analysis of IoT systems

Alfred Åkesson
Görel Hedin
Niklas Fors
alfred.akesson@cs.lth.se
gorel.hedin@cs.lth.se
niklas.fors@cs.lth.se
Lund University
Sweden

Rene Schöne
Johannes Mey
rene.schoene@tu-dresden.de
johannes.mey@tu-dresden.de
Technische Universität Dresden
Germany

ABSTRACT

Internet-of-things systems are difficult to understand and debug due to their distributed nature and weak connectivity. We address this problem by using relational reference attribute grammars to model and analyze IoT systems with unreachable parts. A transitive device-dependency analysis is given as an example.

CCS CONCEPTS

• **Software and its engineering** → **Software as a service orchestration system**; **Integrated and visual development environments**.

KEYWORDS

internet of things, model-driven development, reference attribute grammars, program analysis, dependency analysis

ACM Reference Format:
Alfred Åkesson, Görel Hedin, Niklas Fors, Rene Schöne, and Johannes Mey. 2020. Runtime modeling and analysis of IoT systems. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3417990.3421397

1 INTRODUCTION

Internet-of-Things (IoT) systems are heterogeneous distributed systems that include embedded devices with sensors and actuators, as well as edge computers and/or servers. IoT systems can be even more difficult to understand and debug than ordinary distributed systems, since they may include mobile devices with weak connectivity. This results in a dynamic topology where devices are not always available [13].

One way of supporting better understandability and debugging is to compute a runtime model of the IoT system, and to support analysis of such a model [3]. For example, in a smart home, this could allow the network of connected devices to be visualized. An example of an analysis could be to compute which devices need to work in order for the lights to turn on when the door is opened.

Analyses like these can be useful both for debugging and for finding problems when testing a system, before release.

For this approach to work well, the architecture has to be *explicit* in the sense that communicating components and connectors can be extracted from the system. In particular, it can be beneficial if the connectors have first class status [10], and are not implicit in the code of the components. In particular, if a domain-specific language is used for specifying connectors between components, the architecture is easy to extract. Furthermore, it is desirable to specify analyses of the architecture in a high-level way, preferably using a declarative approach.

It is also important to note that an IoT system might not be defined by a single description in one location, but can emerge from many partial descriptions on different devices. The system may be constantly changing due to new components and connectors being added or removed. Furthermore, because of weak connectivity, the view of the system from any single device can be incomplete.

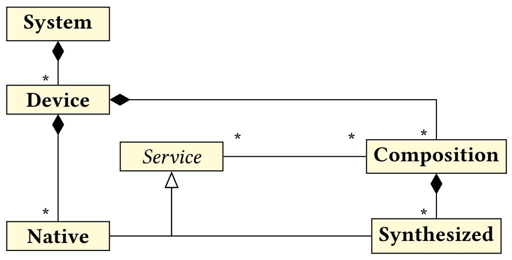
While most language and modeling approaches rely on conceptual modeling frameworks (e.g., EMF [11]) or runtime modeling frameworks (e.g., KMF [5]), these only explicitly support specific kinds of analysis, such as type analysis or constraint checking, and fall back on general-purpose languages for other kinds of analysis.

Thus, in this paper we propose using *Relational Reference Attribute Grammars* [8] for modeling and analyzing IoT systems with an explicit architecture. Reference Attribute Grammars (RAGs) support declarative analysis over abstract syntax trees and are used for building compilers and other language-based tools. Relational RAGs extend the abstract syntax of RAGs with relations, so that the structure to analyze is a conceptual model rather than an abstract syntax tree.

We evaluate the approach by developing a runtime model for PalCom [12], an IoT middleware toolkit that uses an explicit architecture. PalCom components are called *services*, and connectors are called *compositions*. The compositions are described by DSL scripts that define what services to connect to, and how messages are mediated. The toolkit provides a discovery manager that keeps track of all connected devices, and what services and compositions that run on them.

As an example analysis, we have formulated and implemented a simple *device dependency analysis* (DDA). The DDA computes what devices need to be available and connected in order for a specific event to happen, such as turning on a light when a door opens.

Our contributions are the following:



```
System ::= Device*;
Device ::= Native* Composition*;
abstract Service;
Native:Service;
Synthesized:Service;
Composition ::= Synthesized*;
rel Composition.connectedTo* <-> Service.connectedFrom*;
```

Figure 1: Basic model for running PalCom system. *Upper:* conceptual model diagram. *Lower:* Corresponding abstract grammar.

- We present a basic runtime model for the PalCom IoT architecture, formalized using Relational RAGs (Section 2).
- We present a home automation scenario as a motivating example (Section 3).
- We present an extended runtime model that can handle incomplete systems (where devices can be unavailable) and that includes composition scripts that enable more fine-grained analyses than the basic model (Section 4).
- We introduce and formalize the Device Dependency Analysis (DDA), and show how it can be specified using Relational RAGs on top of our extended runtime model (Section 5)

We end with related work and conclusions (Sections 6 and 7).

2 BASIC RUNTIME MODEL

A running PalCom IoT system consists of a set of devices running the PalCom middleware, together with a number of service and composition instances, each hosted on a particular device. Each service provides an API of asynchronous messages that it can send and receive. However, a service does not set up any connections on its own. Instead, *compositions* act as connectors, each connecting to a number of services.

For this paper, all compositions are written in a domain-specific language called ComPOS [1]. A composition mediates between services by receiving and sending messages from/to them. A composition can also provide *synthesized* services that other compositions can connect to. Ordinary services (that are not synthesized) are called *native*, and are typically written in a general-purpose language like Java.

Because all connections are available in the composition scripts, which are easy to analyze, the complete component-connector architecture is available without having to analyze the general-purpose code of the native services.

Figure 1 shows the basic conceptual model of a running PalCom IoT system, and the corresponding abstract grammar of the relational RAG. The grammar specifies containment relations using grammar productions, and non-containment relations using the `rel` construct.

Listing 1: Attributes defining derived properties

```
↑ Device.allServices : 𝒫(Service)
↓ Service.host : Device
↓ Composition.host : Device
eq Device.allServices = Native ∪ (⋃_{c∈Composition} c.Synthesized)
eq Device.**.host = this
```

A relational RAG can be extended with attributes and equations in order to declaratively specify derived properties. Each attribute of a node is defined by an equation, either in the node itself (labelled by ↑), or in an ancestor in the containment hierarchy (labelled by ↓). (In attribute grammar terminology, ↑/↓ attributes are called synthesized/inherited respectively.)

Listing 1 shows an attribution example (syntax slightly simplified for presentation purposes). Here, the attribute ↑`allServices` of a device is defined as the union of all its native services and all the synthesized services of its compositions. The compositions and services have attributes ↓`host` that refer to the hosting device. This attribute is defined by an equation on the `Device` node, which is an ancestor of compositions and services.

3 RUNNING EXAMPLE

As a running example of IoT runtime models and their analyses, we tell a story about the tech-savvy Mark and how he deploys an IoT system to communicate measurements to his physician[1].

3.1 A simple IoT system

Mark has a chronic illness and needs to regularly inform his physician about his body temperature. The hospital provides a database service to which measurements can be sent from smart devices like thermometers, blood pressure monitors, etc. All the hospital staff have access to the database, but Mark would like only his physician to access his data, and would therefore like to encrypt his measurements with the public key of his physician.

Mark has a smart thermometer with a temperature service that sends a message whenever a measurement is taken. To build the system, Mark deploys an encryption service on the smart thermometer, and then creates a composition in ComPOS to connect the services. An overview of the system is shown in Figure 2.

The composition script is shown in Listing 2. It specifies what services to use and on what devices the services run (lines 2-4). Line 5 waits for the thermometer to send a measurement, storing it in the variable `t`. Lines 6-7 sends the temperature to the encryption service and receives the encrypted temperature. Line 8 sends the encrypted temperature to the measurement database at the hospital.

3.2 Updating the system

The system works fine, but Mark's illness makes it difficult for him to move around in the apartment. He therefore gets one more thermometer, so he can have one in his bedroom and one in the living room. He first updates the composition to work on any thermometer by changing line 2 to `service tmp = Temperature on this` (`this` binds to the device the composition is running on). He then copies the composition to the new thermometer, and it seems to work.

To get more confidence in that the system works as it should, he runs a *device dependency analysis* (DDA). With the DDA, he

[1]We use the example as a pedagogical tool, and thus it is not entirely realistic.

checks which devices are needed for his new thermometer to send a `store` message to the database. The DDA is run on the deployed system in order to resolve `this` expressions and to compute what concrete devices the system depends on. To Mark's surprise, the analysis reports that not only the new thermometer and the hospital database are needed, but the old thermometer is needed as well!

Mark looks at the composition again, and realizes that the old thermometer is used for the encryption. While this works, it means that the new thermometer will stop working if the old one runs out of battery. He fixes the problem by updating the compositions on line 3 to bind to the `Encrypter` on the same device (using `this` for the device).

4 THE FULL SYSTEM MODEL

The full system model includes support for incomplete systems (due to unavailable devices) and composition scripts for allowing fine-grained analyses. Both the conceptual model and its corresponding specification using Relational RAGs is shown in Figure 3.

A `System` contains both a dynamic part for what is currently known about devices, service instances, and composition instances on the network, and a static part that contains the composition scripts. Each composition instance is related to its corresponding script (line 13 in the grammar). There is an abstract grammar for the composition scripts as well, that we omit here for brevity, and that is reused from the implementation of ComPOS [1].

The dynamic part is similar to the basic runtime model in Figure 1, but introduces handles `DeviceHandle` and `ServiceHandle`. These are used for representing devices and service instances that a composition (tries to) connect to, regardless of if they are available on the network or not. The handles have optional relations to the corresponding true `Device` and `Service` entities, that are present in the model if they are available on the network.

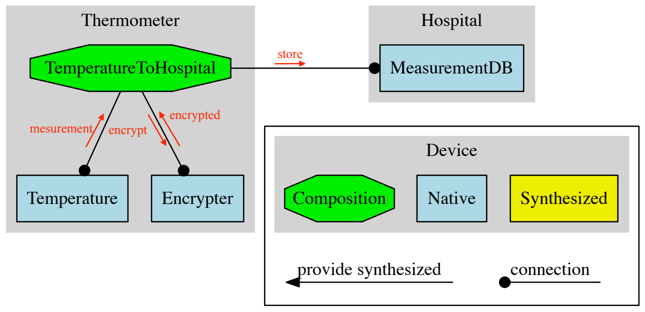
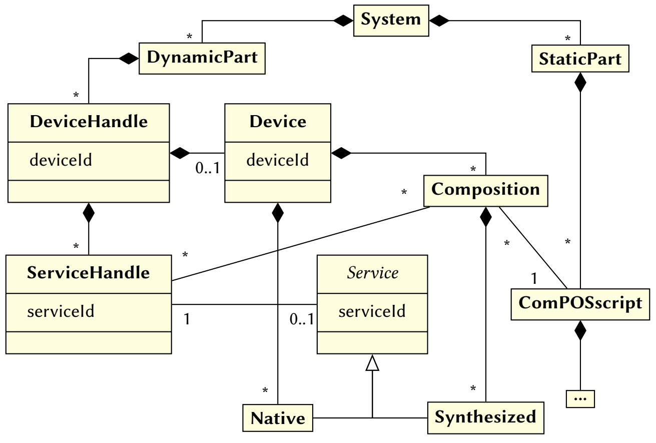


Figure 2: Overview of Mark's initial system. (See Figure 5 for example with synthesized service.)

Listing 2: Composition connecting services

```
1  composition: TemperatureToHospital
2   service tmp = Temperature on Thermometer
3   service enc = Encrypter on Thermometer
4   service mDB = MeasurementDB on Hospital
5   when receive measurement(var t) from tmp do
6    send encrypt(t) to enc
7    receive encrypted(var et) from enc
8    send store(et) to mDB
```



```
1   System ::= DynamicPart StaticPart;
2   DynamicPart ::= DeviceHandle*;
3   DeviceHandle ::= <deviceId> ServiceHandle* [Device];
4   Device ::= <deviceId> Native* Composition*;
5   ServiceHandle ::= <serviceId>;
6   abstract Service ::= <serviceId>;
7   Native:Service;
8   Synthesized:Service;
9   Composition ::= Synthesized*;
10  rel Composition.connectedTo* <-> ServiceHandle.connectedFrom*;
11  rel ServiceHandle.service? <-> Service.serviceHandle;
12  StaticPart ::= ComPOSscript*;
13  rel Composition.implementation <-> ComPOSscript.instances*;
14
15  ComPOSscript ::= ...
```

Figure 3: Full system model with diagram and grammar (ComPOS details omitted)

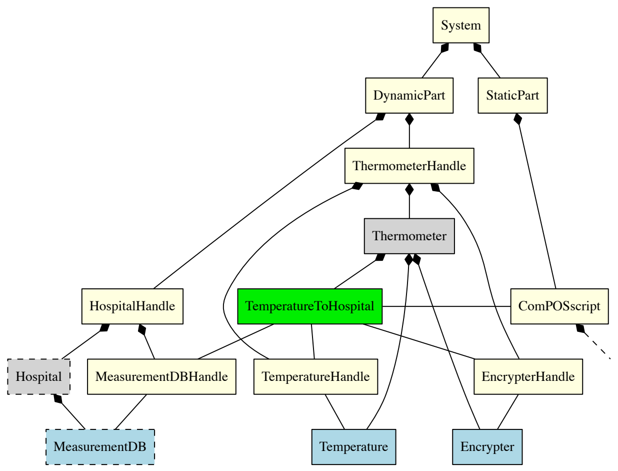


Figure 4: Object diagram over Mark's system

4.1 Handling unavailable devices

Each device is identified by a globally unique id, *deviceId*, and each running service instance has a device-locally unique id, *serviceId*. A running service is globally identified with a tuple *(deviceId, serviceId)*.

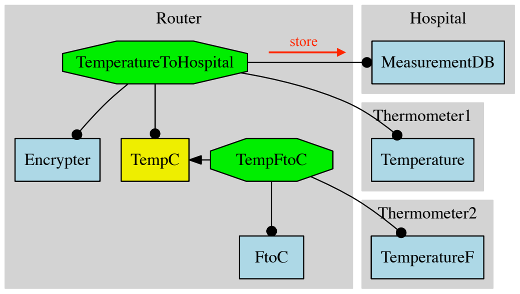


Figure 5: System with two thermometers and a router.

The PalCom middleware includes a discovery manager that can be run on any device. It keeps track of transitively available devices, as well as services and compositions that run on them.

To populate the model, the discovery manager is queried for its information, adding device handles, devices, service handles, services, and compositions. When a composition is added, it has information about each service instance it (tries to) connect to, i.e., the tuple *(deviceId, serviceId)*. If handles for the corresponding device and/or service are not already present in the model, they are added. At a later point in time, when new devices and service/composition instances are discovered, the model is automatically updated.

Figure 4 shows the object diagram from the perspective of Mark's home. If the hospital device is unreachable, the model will not contain the dashed elements. Devices, compositions, and services are shown in the same colors as in the overview figure.

5 DEVICE DEPENDENCY ANALYSIS

The Device Dependency Analysis (DDA) computes what sets of devices need to be available for a given message to be sent from a given composition and received by a given service.

5.1 Example

Consider the IoT system in Figure 5, where Mark has added two thermometers and moved the encrypter service to a router device. Because Thermometer2 uses Fahrenheit, Mark has created TempC, a synthesized service provided by TempFtoC, to convert the temperature to Celsius using a native service FtoC. The DDA could, for example, compute which devices are needed for the `store` message to be sent. To compute this, the analysis takes into account the control flow for both *TemperatureToHospital* and *TempFtoC*.

The analysis takes three arguments: c, m, and s, where c is the sending composition, m is the name of the message, and s is the receiving service. The output of the analysis is a set of device sets $\{D_1, D_2, \ldots\}$, where each D_k is the set of devices needed in one control-flow that leads to c sending message m to s.

For the system in Figure 5, the DDA for the `store` message, i.e., `dda(temperatureToHospital, "store", MeasurementDB)`, would give the result $\{ \{t_1, r, h\}, \{t_2, r, h\} \}$, where r is the router, t_1 and t_2 are the thermometers, and h is the hospital server. The result can be interpreted as a logical formula on disjunctive normal form, in this case $(t_1 \wedge r \wedge h) \vee (t2 \wedge r \wedge h)$, where a device is true if it is available on the network. The `store` message can be sent when this formula is true, i.e., when the router and hospital servers are available, and at least one of the thermometers.

Listing 3: Control-flow of composition language

```
↑ ComPOSscript.ddaC(c:Composition, m:String, s:Service) : 𝒫(𝒫(Dep))
↑ ComPOSscript.ddaS(s:SynthesizedService, m:String) : 𝒫(𝒫(Dep))
eq ...
```

Listing 4: DDA analysis

```
1  ↑ System.dda(c:Composition, m:String, s:Service) : 𝒫(𝒫(D))
2  eq System.dda(c, m, s) = let 𝔼 = c.implementation.ddaC(c, m, s)
3    in let res = ⋃_{E∈𝔼} ⨂_{e∈E} e.serviceHandle.expand(e.message)
4      in toDevices(res) ⊗ { {c.host, s.host} }
5
6  ↑  ServiceHandle.expand(m): 𝒫(𝒫(Dep))
7  eq ServiceHandle.expand(m) =
8   this.hasService ? this.service.expand(m) : { {(this, m)} }
9
10 ↑  Service.expand(m): 𝒫(𝒫(Dep))
11 eq Native.expand(m) = { {(this.serviceHandle, m)} }
12 eq Synthesized.expand(m) =
13   let 𝔼 = this.composition.implementation.ddaS(this.serviceHandle, m)
14   in let res = ⋃_{E∈𝔼} ⨂_{e∈E} e.serviceHandle.expand(e.message)
15     in res ⊗ { {(this.serviceHandle, m)} }
```

5.2 Control-flow analysis of compositions

The control-flow analysis of the composition scripts is abstracted into two attributes ↑`ddaC` and ↑`ddaS`, that represent two kinds of control-flow. ↑`ddaC` computes dependencies needed for a composition to send a given message to a given service. ↑`ddaS` computes the dependencies needed for a synthesized service to send (or reply) a given message to some composition that connects to it. By introducing these attributes, the rest of the DDA can treat the actual composition language as a black box, see Listing 3. The equations for these attributes are omitted for brevity.

A dependency $dep \in Dep$ is relative to a given composition, and is modeled as a tuple `(serviceHandle, message)`, representing that a given service sends a given message to the composition. Both ↑`ddaC` and ↑`ddaS` return a set of sets of dependencies, i.e., a value of type $\mathcal{P}(\mathcal{P}(Dep))$, that we call a *dependency expression*, $\mathbb{E}$. It can be thought of as a logical expression in disjunctive normal form, similarly as for devices discussed earlier, but where a variable is a dependency.

5.3 Implementation of the DDA

We can now describe how the DDA is implemented for the conceptual model, using Relational RAGs, as shown in Listing 4.

To correctly analyze TemperatureToHospital and TempFtoC in Figure 5, the analysis needs to be transitive and follow dependencies between different compositions that are connected via synthesized services. In our analysis, we consider transitive dependencies only in the forward direction, when a composition sends a message through a synthesized service. To also consider messages received through a synthesized service, the analysis would need to be extended.

The analysis makes use of the attributes ↑`ddaC` and ↑`ddaS` to compute control flow in composition scripts, and the attribute ↑`expand` to compute a transitive closure of all dependency expressions. It then projects the expanded dependency expression down to a set of sets of devices.

The calculation of the transitive closure is done by first calculating the `ddaC` (line 2) and then expanding each of the dependency sets in the resulting dependency expression. The expansion on line

3 uses the operator $\otimes : (\mathcal{P}(\mathcal{P}(Dep)), \mathcal{P}(\mathcal{P}(Dep))) \mapsto \mathcal{P}(\mathcal{P}(Dep))$, where $\mathbb{E}_l \otimes \mathbb{E}_r = \bigcup_{E_l \in \mathbb{E}_l, E_r \in \mathbb{E}_r} (\{E_l \cup E_r\})$.

For synthesized services, the expansion leads to calls to $\uparrow$ddaS for the composition containing the synthesized service (line 13). These dependency expressions are then expanded recursively (line 14), to compute the transitive closure.

For native services, the expansion cannot go further, and just returns the same expression that was expanded (line 11).

For an incomplete ServiceHandle (i.e., when there is no corresponding Service), the expansion also stops, and the same dependency expression is returned (line 8). This lack of information is also computed, so the user can see where the analysis is not complete, but the specification of this is left out for brevity.

The result of the DDA computation is shown on line 4: The fully expanded dependency expression is projected to the corresponding set of sets of devices, adding the devices of the original sending composition and receiving service to each device set.

5.4 Discussion

The DDA analysis is not trivial, and while it would have been possible to write it using an ordinary general-purpose language like Java, writing it using Relational RAGs gives a concise executable high-level specification. The performance of RAGs is on par with ordinary general-purpose languages, and a reason for this is that RAG evaluation is optimal in that each attribute value is computed at most once. This is achieved by automatic memoization of attributes during evaluation [7].

6 RELATED WORK

A recent survey [2] of works in the area of models@run.time revealed two research challenges tackled by our approach. First, only few works directly address uncertainty, which in our work was taken into account using handles for both services and devices. Secondly, the need for distributed runtime models was identified, which we tackle by incorporating the PalCom middleware.

A similar approach in the domain of smart homes was presented in [9]. There, the problem on how to connect multiple smart home middleware systems with different machine learning components was discussed and a runtime model to include all necessary information was presented as solution. Similar to our approach, RAGs were used to describe the model. However, the distributed nature of application in the domain of Internet of Things was not considered and left to the middleware systems.

Hartmann presented an approach in [6] for modeling large cyber-physical systems. Similar to our approach, the changing nature of runtime systems was in the focus, but they use streams to split up the complete model into atomic information and use analysis in both time and multiple worlds to support what-if-analysis. As an implementation basis, KMF [5] was chosen and extended to support a fixed set of derived properties, similar to attributes of RAGs.

Dai et al. use logic programming to implement dependency analysis for workflows in [4]. Instead of having a model, they have facts, and instead of attributes, they have rules. The use of logical programming allows them to express different kinds of queries, which our current approach does not support. Compared to our work, Dai et al. do not consider distributed systems and thus can do their analysis statically. They also explore data dependencies, which we might want to support in the future.

7 CONCLUSION

We have presented how Relational RAGs can be used for modeling and analyzing IoT systems with weak connectivity. The approach was evaluated by developing a runtime model for the PalCom IoT middleware, and implementing a forward transitive device dependency analysis as an example. The analysis can be implemented in a high-level compact way using the Relational RAGs.

In the future, we plan to use this work as a basis for exploring different kinds of analysis in IoT systems, such as information flow and placement of compositions, and thus helping developers find problems before making their systems available. It would also be interesting to implement the DDA in a conventional model transformation tool, for comparison.

ACKNOWLEDGMENTS

This work was in part supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation and in part by the Swedish Foundation for Strategic Research, grant RIT17- 0035. It is also partly supported by the German Research Foundation (DFG) as part of Germany's Excellence Strategy – EXC 2050/1 "CeTI" and the project "HybridPPS" (project number 418727532), and using tax money based on the budget approved by the Saxon state parliament for the project "PROSPER".

REFERENCES

[1] A Åkessson, G Hedin, B Magnusson, and M Nordahl. 2019. ComPOS: Composing Oblivious Services. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 132–138.
[2] N Bencomo, S Götz, and H Song. 2019. Models@run.time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling* 18, 5 (Jan. 2019), 1619–1374.
[3] G Blair, N Bencomo, and R B. France. 2009. Models@run.time. *Computer* 42, 10 (Oct. 2009), 22–27.
[4] W. Dai, D. Covvey, P. Alencar, and D. Cowan. 2009. Lightweight query-based analysis of workflow process dependencies. *Journal of Systems and Software* 82, 6 (2009), 915 – 931.
[5] F Francois, G Nain, B Morin, E Daubert, O Barais, N Plouzeau, and J-M Jézéquel. 2014. Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use. http://arxiv.org/abs/1405.6817
[6] T Hartmann. 2016. *Enabling Model-Driven Live Analytics For Cyber-Physical Systems: The Case of Smart Grids.* PhD Thesis. University of Luxembourg.
[7] M Jourdan. 1984. An Optimal-time Recursive Evaluator for Attribute Grammars. In *International Symposium on Programming, 6th Colloquium (LNCS)*, Vol. 167. Springer, 167–178.
[8] J Mey, R Schöne, G Hedin, E Söderberg, T Kühn, N Fors, J Öqvist, and U Aßmann. 2020. Relational reference attribute grammars: Improving continuous model validation. *Journal of Computer Languages* 57 (2020), 100940.
[9] R Schöne, J Mey, B Ren, and U Aßmann. 2019. Bridging the Gap between Smart Home Platforms and Machine Learning using Relational Reference Attribute Grammars. In *Proceedings of the 14th International Workshop on Models@run.time.* Munich, 533–542.
[10] M Shaw. 1993. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In *WSSD 1993, Studies of Software Design (LNCS)*, Vol. 1078. Springer, 17–32.
[11] D Steinberg, F Budinsky, E Merks, and M Paternostro. 2008. *EMF: Eclipse Modeling Framework* (2 ed.). Addison-Wesley Professional.
[12] D Svensson Fors, B Magnusson, S Gestegård Robertz, G Hedin, and E Nilsson-Nyman. 2009. Ad-hoc composition of pervasive services in the PalCom architecture. In *Proceedings of the 2009 international conference on Pervasive services.* ACM, 83–92.
[13] A Taivalsaari and T Mikkonen. 2017. A roadmap to the programmable world: software challenges in the IoT era. *IEEE Software* 34, 1 (2017), 72–80.