Experiences with an Internal DSL in the IoT Domain

Matthias Tichy[1], Jakob Pietron[1], David Mdinger[2], Katharina Juhnke[1], and Franz J. Hauck[2]

[1] Institute of Software Engineering and Programming Languages,
[2] Institute of Distributed Systems,
Ulm University, Germany
firstname.lastname@uni-ulm.de

Abstract. Modeling the architecture and behavior of embedded systems has long been a success story in the engineering of embedded systems due to the positive effects on quality and productivity, e.g., by declarative specifications, by enabling formal analyses, and by the generation of optimized code. These benefits, however, can only be reaped with extensive investments in specialized languages and tools which typically come with a closed and highly restrictive ecosystem. In this paper, we report our experiences while building an internal domain-specific language for IoT systems. We present our modeling language realized in TypeScript and integrated into the TypeScript/JavaScript ecosystem. The modeling language supports the declarative specification and execution of components, connectors, and state machines. We also provide a simple state space exploration to enable quality assurance techniques like test case generation and model checking. The language is illustrated by a running example with IoT devices. We believe that our solution lies at a sweet spot of providing a declarative modeling experience while reaping benefits from modern programming languages and their ecosystem to boost productivity.

Keywords: IoT · Domain Specific Languages · Modeling · Experience

1 Introduction

Empirical studies report that model-driven engineering is successfully employed in the engineering of embedded systems (e.g., [2,1,17]). Reported benefits are improved productivity and quality. However, many of those same empirical studies also report that there are drawbacks like bad usability of tools, difficult tool integration, difficult integration into standard development tools, e.g., versioning systems and build systems. Furthermore, our own experience is that the ecosystem of those tools is small (also reported in [15]) which leads to problems such as high

effort for developing systems connected to other "outside" systems and scarcity of information in developer networks like StackOverflow.

The area of Internet of Things (IoT) is an interesting domain in that regard, since on the one hand much of the functionality is provided by embedded systems, but on the other hand an extensive programming language ecosystem exists with numerous libraries. In SORRIR, an ongoing research project on resilient IoT systems, we investigated how to use model-driven engineering while exploiting the benefits from using TypeScript/JavaScript and its ecosystem. Our project's focus of realizing resilient IoT systems [8] requires the ability to be highly flexible and efficient w.r.t. where each component of the system is executed in order to mask and/or recover from failures with minimal downtime. Hence, we decided to use an interpreted language that runs on a wide variety of different hardware – TypeScript/JavaScript. However, that means that the behavior of very restricted devices, e.g., Arduinos and ESP8266 ones, is configured using existing technology and are integrated via MQTT[3] into our framework.

The contribution of this paper is threefold: (1) an internal domain-specific modeling language built using TypeScript which provides modeling for architectural configurations [18] as well as state machines [5] for behavioral definition, which cleverly integrates declarative aspects with code from the TypeScript/ JavaScript ecosystem, (2) a state-space exploration algorithm on these models, which enables further quality assurance techniques like test case generation [26] and model checking [7], and (3) a report on our experiences while developing the DSL and corresponding libraries. The internal DSL and the supporting framework has been published as NPM package[4] and a web frontend running an example [5].

We introduce the running example in the following section. Section 3 contains a description of the modeling language and the realized tool support. This is followed by a discussion of our experiences in Section 4. After a comparison to related work in Section 5, we conclude with an outlook on future work.

2 Running Example

In the course of this paper, we will illustrate our concepts by a simplified parking-garage hardware prototype, see Figure 1. Future parking garages will be enabled to offer smart services to their customers and operators as well such as on-the-fly parking-space reservations or individual guiding.

These intelligent features require different sensor nodes and actors which communicate with a local application server (edge). To manage more than one parking garage and to control the parking traffic of a whole city or to bill customers, a cloud connection is required, too. Furthermore, a parking garage must be operational in case of faulty sensors, servers, or network connection.

The application logic is realized inside the SORRIR framework and running on local server. It consists of four different components. Components barrier and

[3] `https://mqtt.org/`
[4] `https://www.npmjs.com/package/sorrir-framework`
[5] `https://sorrir.github.io/web-demo`

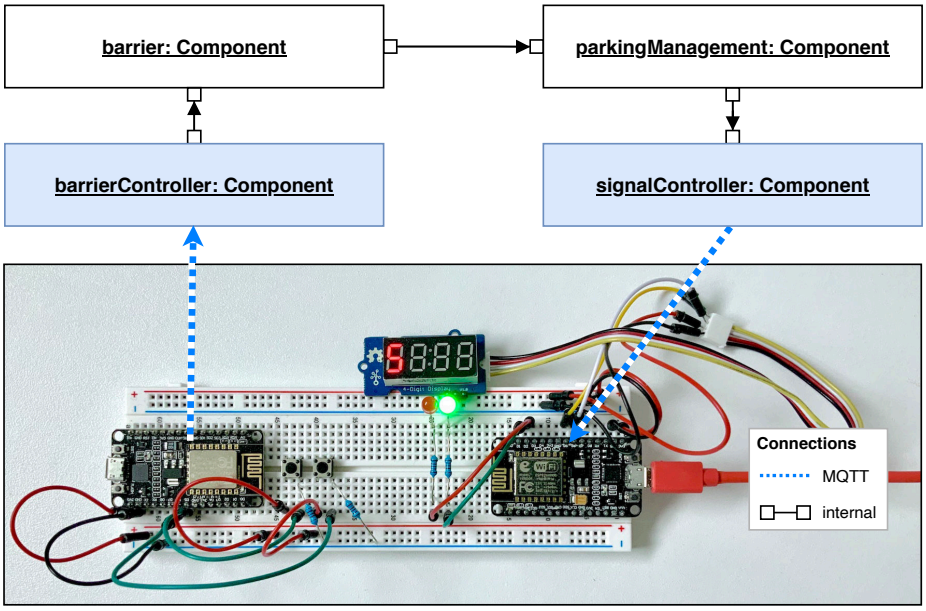

Fig. 1. Prototype overview

parkingManagement – highlighted white – are regular and posses user-defined behavior. Component barrier performs the semantic lifting from button events to car in/out events and ensures that only one car per second can pass the "barrier". Component parkingManagement, see Figure 2, manages free parking lots. It controls the red and green LED as well as the display regarding the available parking lots in the garage. The MQTT components – highlighted blue – are special kind of I/O components provided by the SORRIR framework and implement the communication via MQTT.

On the hardware side there are two ESP8266 microcontrollers (C). The first C (left) is used to simulate the barrier of a parking garage. Two buttons are connected to it to simulate a car entering or leaving the garage. The other C (right) is used to visualize the status of the parking garage. A small display shows the remaining free parking spaces. Additionally, a green LED lights up when there are free parking lots available, otherwise a red LED lights up.

To connect the C with the application we use MQTT and – in our case – the Eclipse Mosquitto MQTT broker[6]. The firmware running on the C is quite simple: it is responsible to send a corresponding MQTT message when a button is pressed or switch the LEDs as well as the display when a message is received. Apart from this, it do not have any further logic. In the course of our testbed, both Cs run a firmware generated by ESPHome[7] through a simple YAML config file. The source code of the example is online[8].

[6] https://mosquitto.org/
[7] https://esphome.io/
[8] https://github.com/sorrir/hardware-testbed

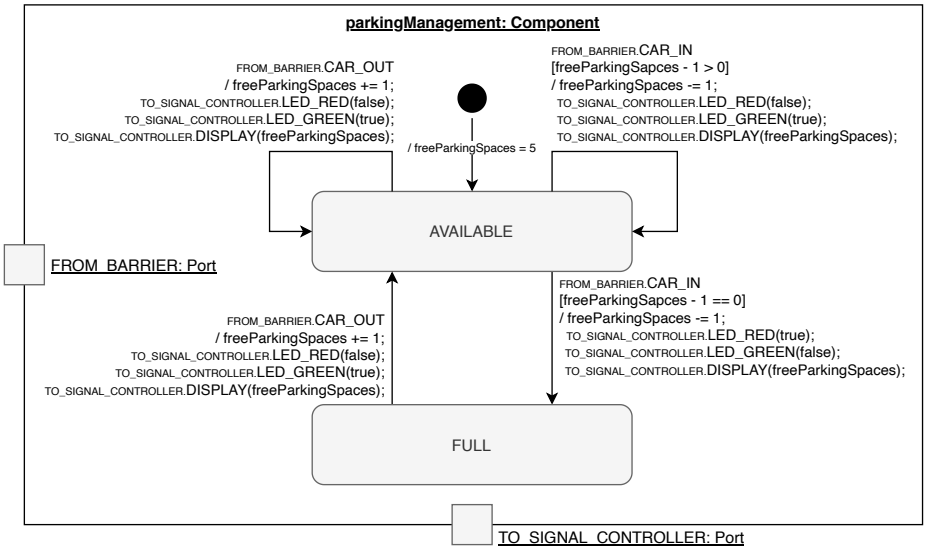

Fig. 2. Behavior of parkingManagement (in simplified pseudo-code).

3 Modeling Language

In the following, we present the modeling language developed in the SORRIR project. The modeling language is inspired by component-based domain-specific languages like Matlab/Simulink/Stateflow, SCADE, and ASCET. Particularly, we draw on our previous work on the MECHATRONIC UML [3], which is a modeling language for the software of embedded systems. Unlike the aforementioned languages, MECHATRONIC UML follows an asynchronous modeling formalism, which better matches distributed IoT systems.

3.1 Architecture

Our architectural description language covers the most relevant structural elements of an architectural description language based on [18], particularly, we provide components, ports and connections. The description is specified using TypeScript types [4,19]. TypeScript types define the structure of an object with respect to both attributes as well as functions. The listing shown in this section are simplified to show only the key aspects.

Listing 1.1 shows the types for the definition of components. Type AbstractState defines the overall state of a component. This type has two attributes covering the component's state and a queue of events. The interesting part here is the extensive usage of TypeScript generics. These generics allow us to have generic code for executing components (and later architectural configurations and state machines) while still enjoying type safety guaranteed by the compiler. The generic parameter E covers the type of events which can be stored in the event queue and processed by the component, the generic parameter P covers the types of

ports of a component. Finally, the generic parameter S enables the typing of the internal state of the component. To pass data as payload across components it is possible to extend the type Event and, for example, add a payload attribute.

```
type AbstractState<S, E, P> = {
  readonly state: S,
  readonly events: Event<E, P>[ ]
}
type Component<E, P> = {
  readonly ports: Port<E, P>[ ],
  step: (current: AbstractState<any, E, P>) => AbstractState<any, E, P>,
  allNextStates: (current: AbstractState<any, E, P>) => AbstractState<any, E, P>[ ]
}
```

Listing 1.1. Component and AbstractState definitions

The type Component defines the structure of a component which is basically the list of component ports and two functions step and allNextStates. These two functions contain the behavior of a component. They get as input an AbstractState and return the new resulting state. It is important that these functions are side-effect free, i.e., they do not modify some internal state, to enable state space exploration and model checking (cf. Section 3.4). Furthermore, this improves testability of the component behavior. The allNextStates function will return all potential resulting states if the component has non-deterministic behavior, e.g., when two different new states are possible depending on which event in the event queue is processed. Our framework includes an interpreter for the state machines providing those two functions. However, these functions can also be implemented manually, which we used for the integration IoT devices reachable via MQTT.

Listing 1.2 shows the simplified type definitions for the definition of an architectural configuration. A Connection connects (multiple) ports of (multiple) components unidirectionally for asynchronous event exchange.

```
type Connection<E> = {
  source: {
    readonly sourceComponent: Component<any, any>,
    readonly sourcePort: Port<E, any>,
  }[ ],
  target: {
    readonly targetComponent: Component<any, any>,
    readonly targetPort: Port<E, any>,
  }[ ]
}
type Configuration = {
  readonly components: Component<any, any>[ ],
  readonly connections: Connection<any>[ ],
}
```

Listing 1.2. Connection and Configuration definitions

The use of the E generic in the definition of the Connection type ensures through type checking by the TypeScript compiler that we only connect ports which share the same E event type. Hence, we do not support a connector where the source emits a subset of events of the target which in principle would be fine.

A Configuration is then just a set of components and a set of connections between them. In this type, we use any as argument for the type parameters to define that for the full configuration any event and port types can be used for

all components and connections, while the aforementioned type definition for connections ensures the event-type correctness for each connection individually.

Our framework executes all components of the configuration by continuously executing the step function as well as exchanging all events as defined by the ports and connections.

3.2 Behavior

The above definitions of components can be used to develop any kind of component by creating an object in TypeScript which conforms to the aforementioned component type and particularly implementing a function which satisfies the type definition of the step and allNextStates function.

However, those fully manually implemented components do not reap the benefits from modeling the behavior like a higher-level of behavioral specification and enabling quality assurance techniques like test case generation and model checking. Our aim is to find a sweet spot for behavioral modeling which combines declarative parts and parts implemented in TypeScript while still supporting the aforementioned techniques. Based on previous experience in MECHATRONIC UML, we decided to realize a state machines formalism notwithstanding that other formalisms could be also employed.

Listing 1.3 shows the type definitions for state machines. The type StateMachineState is an alias for the AbstractState type which refines the above introduced generic parameter S. S is defined as {fsm:F, my:M}, i.e., we distinguish two parts of the state of a state machine component. The first part (fsm:F) covers discrete states as used in the typical state machine formalisms—the type of the discrete states is defined by the type parameter F. The second part (my:M) covers an abstract state as known from abstract state machines [5]. This part is used for arbitrary (TypeScript) data structures in addition to the discrete state.

```
type StateMachineState<F, M, E, P> = AbstractState<{ fsm: F, my: M }, E, P>;
type Transition<F, M, E, P> = {
  readonly sourceState: F,
  readonly event?: [E,P?],
  readonly condition?: (myState: M, event?:Event<E, P>) => Boolean,
  readonly action?: (myState: M, raiseEvent:RaiseEventCallBack<E, P>, event?: Event<E, P>) =>
      M,
  readonly targetState: F,
}
type RaiseEventCallBack<E, P> = (newEvent:Event<E, P>) => void;
type StateMachine<F, M, E, P> = {
  readonly transitions: Transition<F, M, E, P>[ ],
}
```

Listing 1.3. Types for State Machines

The type Transition, typed with the same generics, allows the definition of a transition following the usual event/condition/action-paradigm. Each transition explicitly declares a source and target state as well as whether an event of a certain type E needs to arrive at a port P. The two functions condition and action deal with the abstract state part (my:M) of the state-machine state. They enable the developer to write arbitrary code which checks whether the transition is enabled (function condition evaluates to true), and if yes, creates a new abstract

state based on the current state via the action function. The action function has in addition to the current state a parameter for a callback function which enables the implementation to send an event during the action possible with parameters based on the abstract state.

We believe this mix of declarative parts and manually written parts could be a sweet spot. The declarative parts enable execution by a generic state-machine engine whereas the manually written parts enable to reap the benefits of the TypeScript/JavaScript ecosystem. By developing our modeling language as internal DSL in TypeScript, we can simply use TypeScript as action language for both of these functions instead of manually developing an action language like ALF [24] for executable UML and proper tooling. Particularly, we cannot compete with the efforts spent by all stakeholders in the TypeScript ecosystems and, consequently, would not be able to provide good developer experience if we develop our own action language. We strongly believe that providing good developer experience is of paramount importance for an acceptance by developers. The use of generics ensures that the compiler will directly check that source and target states of transitions are indeed discrete states of that component. Similarly, the type checking of the compiler ensures that only action and condition functions are possible with the correct types for the abstract state as well as the events. Also these checks would have been to be implemented by ourselves if we would have invented our own action language. Furthermore, the type system of TypeScript is much more elaborate than the type system in the usual action languages of MDE approaches.

You can see an implemented transition of ParkingManagement in Listing 1.4. For event CAR_IN on port FROM_BARRIER (line 4) it will switch from AVAILBLE to FULL (lines 2ff.) if the number of free parking lots – stored within the internal state myState – is equal to 0 (line 5). Two events (lines 9ff.) are raised to switch off the green and switch on the red LED. Additionally, an event is sent to update the number of free parking lots at the display. Finally, the updated internal state is returned (line 13).

```
1  {
2    sourceState: ParkingManagementStates.AVAILABLE,
3    targetState: ParkingManagementStates.FULL,
4    event: [EventTypes.CAR_IN, ParkingManagementPorts.FROM_BARRIER],
5    condition: myState => myState.freeParkingSpaces - 1 == 0,
6    action: (myState, raiseEvent) => {
7      const newFreeParkingSpaces = myState.freeParkingSpaces-1;
8
9      raiseEvent({type: EventTypes.LED_RED, port: ParkingManagementPorts.TO_SIGNAL_CONTROLLER,
         payload: {status: true}});
10     raiseEvent({type: EventTypes.LED_GREEN, port: ParkingManagementPorts.TO_SIGNAL_CONTROLLER
         , payload: {status: false}});
11     raiseEvent({type: EventTypes.DISPLAY, port: ParkingManagementPorts.TO_SIGNAL_CONTROLLER,
         payload: {freeSpaces: newFreeParkingSpaces}});
12
13     return {
14       freeParkingSpaces: newFreeParkingSpaces,
15       totalParkingSpaces: myState.totalParkingSpaces
16     };
17   }
18 }
```

Listing 1.4. Transition from AVAILABLE to FULL of ParkingManagement

3.3 IoT Ecosystem

To interact with hardware nodes equipped with sensors and actors we need a possibility to interact with the world outside of our application. Therefore, we add *I/O components* to our approach. These are special kinds of of I/O-enabled components with a predefined internal state and behavior offered by the underlying SORRIR framework.

Initially, we decided to implement MQTT I/O components within the SORRIR framework, because MQTT is a wide-spread, lightweight, IP-based machine-to-machine protocol for the IoT. Clients connected to a central broker can publish messages to certain topics and receive message on topics they are subscribed to.

To communicate with both microcontrollers, we added two I/O components to our running example: barrierController and signalController, see Figure 1. These MQTT components encapsulate all network and MQTT-specific tasks, such as connecting to the broker or subscribing as well as publishing to MQTT topics.

Generally, events passed from component to component or even within a component require at least a Type and, optionally, a Port and further user-defined attributes. MQTT, instead, requires for each message a *topic* and a *payload*. This rises the need for a function that decodes MQTT messages to Events and an encode function vice versa. In our approach, the user has to specify and pass such functions to the corresponding I/O component.

For our running example we choose MQTT to demonstrate the applicability of I/O components. Of course, it is possible to implement further protocols or techniques, such as communicating with a ZigBee gateway.

3.4 Supporting Quality Assurance

Many quality-assurance approaches used in Model-Driven Engineering use the state space of the modeled system as a basis for quality analysis. For example, model checking algorithms generate the state space of the system to check whether a specific property, e.g., a CTL formula, is satisfied by the system model. Similarly, model-based testing approaches use the state space to generate test cases. Finally, the state space or at last an excerpt is an important tool in manual fault-finding activities as it enables developers to check how a system reacts to certain event sequences starting in a certain state.

Our approach supports such an exploration of the state space due to side-effect free functions and explicit state objects independent of the components and state machines. Furthermore, the components' allNextStates function (cf. Listing 1.1), either manually implemented or our implemented by our interpreter for state machines, calculates all next states of a component in case of non-deterministic behavior. We additionally provide a function, which computes all next states of an architectural configuration by computing the cross product of all next states of all components. Based on these functions, our framework allows the computation of the explicit state space of the system as well as the next N states in the state space as a bounded state space exploration, with the caveat that physical parts of our IoT system needs to be replaced by components simulating the behavior.

4 Experiences

Generally, we had positive experiences in developing the framework as well as examples including the illustrative one of Section 2.

Internal DSL embedded in TypeScript. Generally, our experiences developing both the interfaces, the execution framework and the web frontend as well as the illustrative examples are very positive. We do not have quantitative data to compare our spent effort with the effort required for developing a similar system in traditional meta-modeling environments like EMF. However, our experience in developing Mechatronic UML and Henshin [25], an EMF-based model-transformation system, indicates that a textual DSL for the declarative parts would require similar effort using state-of-the-art tools like Xtext whereas a graphical DSL would require much more effort. However, developing an action language similar to the expressiveness of TypeScript would be infeasible. Further, providing the same quality of tooling as provided by Visual Studio Code would be similarly infeasible.

Type Checking. TypeScript provides a mature type system. We extensively use generics for typing of events, ports and component states. Hereby, we offload type checking to the TypeScript compiler instead of manually writing a type checker (and a corresponding advanced type system) for an action language. The type checking ensures that connections only connect ports with the same set of events, i.e., the same types. The only downside is that our solution does not support connections where one port supports just a subset of events of the corresponding port on the other side of the connection.[9] Our experience is furthermore that the type inference provided by the TypeScript compiler is generally helpful, with type hints only required very seldomly for type generics when instantiating components.

Pure functions. We designed the TypeScript interfaces as side-effect-free pure functions. While this required sometimes a bit more effort to clone and modify the data instead of modifying the data directly, we feel that this effort is well spent as it greatly simplifies the execution framework, enables the state-space exploration and makes it easier to test the components and the execution framework itself. Unfortunately, while interfaces in TypeScript support readonly annotations for attributes, the attributes themselves could be complex object structures with non-readonly attributes. Hence, the language cannot enforce pure functions and we are dependent on the developer to only write pure functions.

The TypeScript/JavaScript ecosystem. Particularly, the TypeScript/JavaScript ecosystem has been a huge benefit to us. For example, our web fronted was developed in a few hours using React. Furthermore, this also allows in principle each component to provide its own UI additions to the generic one provided by us. Similarly, the integration of the IoT microcontrollers in our illustrative example proved to be very easy. We used an existing firmware which already supports communication via MQTT. It was very easy to develop specific bridge components that integrate any other service available via MQTT by just reusing

[9] This would be possible if TypeScript allowed a `super` definition in generics as in Java.

existing MQTT libraries and writing a bit of glue code. Furthermore, every user of our framework can write similar components for interaction with outside services. Hence, our framework can be easily extended as needed. The only downside is that this integration of external services does not easily match our design requirement of pure functions. Thus, for activities like state-space exploration, users need to replace those components by simulated proxies.

5 Related Work

In a recent study on approaches to deployment and orchestration of IoT applications, Nguyen et al. [21] emphasize that language support plays an important role. In this respect, the use of Domain-Specific Languages (DSLs) is now more popular than General-Purpose Languages (GPLs). This applies to DSLs in both textual and graphical form. The underlying programming models are mostly component-based, service-based or flow-based models. Furthermore, the results of their systematic literature research [21] show that only very few primary studies deal with resilience.

When it comes to modeling languages for IoT systems, however, there are various approaches. One of the best known is Node-RED [22], which enables browser-based flow editing of nodes within an IoT system using a graphical DSL. There are several other approaches based on Node-RED, such as Distributed Node-RED (D-NR) [10] or glue.things [16]. Node-RED is similar to DSL-4-IoT [23], which provides an integrated development environment and a graphical programming approach that aims to simplify the programming complexity of IoT and overcome heterogeneity. However, DSL-4-IoT is limited in terms of scalability and expandability. Another graphical DSL based on the UML is the Visual Domain-Specific Modeling Language (VDSML) [9], which aims to enhance its usability for people without engineering background. Similar goals are also targeted by the graphical DSLs Midgar [11] and "If this than that" (IFTTT) [14]. As Negash et al. [20] also mention, most of these DSLs are proposed to simplify the process of creating the workflow. However, to the best of our knowledge there are no DSLs that are explicitly designed for experienced developers. Furthermore, compared to our DSL, the modeling languages mentioned are no textual DSLs.

Negash et al. [20] address the limitations regarding scalability and extensibility, as they exist for DSL-4-IoT, with their textual DSL called Domain-Specific IoT Language (DoS-IL). Based on a client-server-client architecture, the system behavior can be adapted using DoS-IL scripts. Furthermore, Cherrier et al. [6] provide another textual DSL called SALT (Simple Application Logic Description using Transducers) for the configuration of nodes in an IoT application and thus the description of the application logic. Moreover, Hussein et al. [13] describe a model-driven approach to modeling adaptive IoT systems using SysML4IoT, which is an adaptation of SysML for modeling system functionality with the Papyrus UML Modeler. Adaptive behavior can be modeled with state machines. Another textual, external DSL is presented by Harrand et al. [12] with the ThingML approach. The approach consists of a DSL and complementing tooling,

particularly, code generation is provided. While the modeling language also has basic elements like components, connectors and state machines and theoretically allows arbitrary integrations with the target code, it provides an own restricted action language. Consequently, the expressiveness is limited to the DSL provided by the ThingML approach. In comparison, our approach provides an internal DSL which allows us to fully use TypeScript/JavaScript as action language and easily integrate with the complete JavaScript ecosystem boosting productivity. However, this means that, in constrast to ThingML, we are also bound to platforms that can execute JavaScript and our capabilities for behavioral analysis are restricted.

6 Conclusion and Future Work

In this paper we described a modeling language for the Internet of Things. The modeling language provides state-machine–based declarative components, as well as connections to form complex systems. The language is integrated in the TypeScript and JavaScript ecosystem and can be accessed via a published npm package. Further, we provide tooling for state-space exploration and graphical components for state introspection and monitoring. With this support, our system lays the foundation for test-case generation, model checking and other quality-assurance techniques.

We showed the applicability of our language, technique and tooling through a real world example. The experiences collected and described further validate our belief of approaching a sweet spot between declarative modeling as well as programming features and available ecosystem.

Our major area of future work is on extending the internal DSL with additional resilience features. First and foremost, the declarative modeling should allow for graceful degradation. We plan to enable this on the state-machine level through failure modes. These trigger annotated state transitions, but are restricted to quiescent states to preventing undesired side effects.

The model is designed to support further resilience mechanisms. Configuration-transformation functions, taking a configuration and producing a new one, supporting state and component replication, need to be added next. Lastly, as the Internet of Things is inherently distributed, our implementation should naturally support distributed components. This challenge can mostly be delegated to the execution environment, as it is responsible to deliver events through channels, even when they interconnect physical machines.

Once these additions are included in the language, execution environment and tooling, our system will provide a solid base for developers to create truly resilient IoT applications.

Acknowledgments. The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany (grant nr. 01IS18068, SORRIR).

References

1. Agner, L.T.W., Soares, I.W., Stadzisz, P.C., Simão, J.M.: A brazilian survey on UML and model-driven practices for embedded software development. J. Syst. Softw. **86**(4), 997–1005 (2013). https://doi.org/10.1016/j.jss.2012.11.023
2. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context - motorola case study. In: Briand, L.C., Williams, C. (eds.) 8th Int. Conf. on Model-Driven Eng. Lang. & Sys.—MoDELS. LNCS, vol. 3713, pp. 476–491. Springer (2005). https://doi.org/10.1007/11557432_36
3. Becker, S., Dziwok, S., Gerking, C., Schäfer, W., Heinzemann, C., Thiele, S., Meyer, M., Priesterjahn, C., Pohlmann, U., Tichy, M.: The MechatronicUML design method—process and language for platform-independent modeling. Tech. Rep. tr-ri-14-337, Heinz Nixdorf Inst., Univ. of Paderborn, Germany (Mar 2014)
4. Bierman, G.M., Abadi, M., Torgersen, M.: Understanding TypeScript. In: Jones, R.E. (ed.) 28th Eur. Conf. on Obj.-Orient. Prog.—ECOOP. LNCS, vol. 8586, pp. 257–281. Springer (2014). https://doi.org/10.1007/978-3-662-44202-9_11
5. Börger, E., Raschke, A.: Modeling Companion for Software Practitioners. Springer (2018). https://doi.org/10.1007/978-3-662-56641-1
6. Cherrier, S., Ghamri-Doudane, Y.M., Lohier, S., Roussel, G.: D-LITe: Distributed Logic for Internet of Things Services. In: Proc. 4th Int. Conf. Cyber Phys. & Soc. Comp.—iThings/CPSCom. pp. 16–24. IEEE (2011). https://doi.org/10.1109/iThings/CPSCom.2011.33
7. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
8. Domaschka, J., Berger, C., Reiser, H.P., Eichhammer, P., Griesinger, F., Pietron, J., Tichy, M., Hauck, F.J., Habiger, G.: SORRIR: A resilient self-organizing middleware for iot applications [position paper]. In: Proceedings of the 6th International Workshop on Middleware and Applications for the Internet of Things, M4IoT@Middleware 2019, Davis, CA, USA, December 09-13, 2019. pp. 13–16. ACM (2019). https://doi.org/10.1145/3366610.3368098
9. Eterovic, T., Kaljic, E., Donko, D., Salihbegovic, A., Ribic, S.: An Internet of Things visual domain specific modeling language based on UML. In: 25th Int. Conf. Inf., Comm. & Autom. Techn.—ICAT. pp. 1–5. IEEE (2015). https://doi.org/10.1109/ICAT.2015.7340537
10. Giang, N.K., Blackstock, M., Lea, R., Leung, V.C.: Developing IoT Applications in the Fog: A Distributed Dataflow Approach. In: 5th Int. Conf. Internet of Things—IOT. pp. 155–162. IEEE (2015). https://doi.org/10.1109/IOT.2015.7356560
11. González García, C., Pelayo G-Bustelo, B.C., Pascual Espada, J., Cueva-Fernandez, G.: Midgar: Generation of heterogeneous objects interconnecting applications. A Domain Specific Language proposal for Internet of Things scenarios. Computer Networks **64**, 143–158 (2014). https://doi.org/10.1016/j.comnet.2014.02.010
12. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: Thingml: A language and code generation framework for heterogeneous targets. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS'16). p. 125135. Association for Computing Machinery (2016). https://doi.org/10.1145/2976767.2976812, `https://doi.org/10.1145/2976767.2976812`
13. Hussein, M., Li, S., Radermacher, A.: Model-driven Development of Adaptive IoT Systems. In: 4th Int. Worksh. Interplay of Model-Driv. & Comp.-Based Softw. Eng.—ModComp (2017)

14. IFTTT Inc.: Ifttt: If this then that (2020), https://ifttt.com
15. Karg, S., Raschke, A., Tichy, M., Liebel, G.: Model-driven software engineering in the openetcs project: project experiences and lessons learned. In: Baudry, B., Combemale, B. (eds.) 19th Int. Conf. on Model-Driven Eng. Lang. & Sys.—MoDELS. pp. 238–248. ACM (2016)
16. Kleinfeld, R., Steglich, S., Radziwonowicz, L., Doukas, C.: glue.things - a Mashup Platform for wiring the Internet of Things with the Internet of Services. In: 5th Int. Worksh. on Web of Things—WoT. pp. 16–21. ACM (2014). https://doi.org/10.1145/2684432.2684436
17. Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J.: Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. Softw. & Sys. Model. **17**(1), 91–113 (2018). https://doi.org/10.1007/s10270-016-0523-3
18. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000). https://doi.org/10.1109/32.825767
19. Microsoft Corp.: TypeScript Language Specification v1.8 (2016), http://typescriptlang.org
20. Negash, B., Westerlund, T., Rahmani, A.M., Liljeberg, P., Tenhunen, H.: DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices. Procedia Comp. Sci. **109**, 416–423 (2017). https://doi.org/10.1016/j.procs.2017.05.411
21. Nguyen, P., Ferry, N., Erdogan, G., Song, H., Lavirotte, S., Tigli, J.Y., Solberg, A.: Advances in Deployment and Orchestration Approaches for IoT - A Systematic Review. In: IEEE Int. Congr. Internet of Things—ICIOT. pp. 53–60. IEEE (2019). https://doi.org/10.1109/ICIOT.2019.00021
22. OpenJS Foundation: Node-red – lox-code programming event-driven applications, http://nodered.org
23. Salihbegovic, A., Eterovic, T., Kaljic, E., Ribic, S.: Design of a domain specific language and IDE for Internet of things applications. In: 38th Int. Conv. Inf. & Comm. Techn., Electr. & Microelectr.—MIPRO. pp. 996–1001. IEEE (2015). https://doi.org/10.1109/MIPRO.2015.7160420
24. Seidewitz, E.: UML with meaning: executable modeling in foundational UML and the alf action language. In: Feldman, M., Taft, S.T. (eds.) ACM SIGAda Ann. Conf. High Integr. Lang. Techn.—HILT. pp. 61–68. ACM (2014). https://doi.org/10.1145/2663171.2663187
25. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for EMF model transformation development. In: de Lara, J., Plump, D. (eds.) 10th Int. Conf. Graph Transf.—ICGT. LNCS, vol. 10373, pp. 196–208. Springer (2017). https://doi.org/10.1007/978-3-319-61470-0_12
26. Tretmans, J.: A formal approach to conformance testing. In: Rafiq, O. (ed.) Protocol Test Systems, VI, IFIP TC6/WG6.1 6th Int. Worksh. Protocol Test Sys. IFIP Trans., vol. C-19, pp. 257–276. North-Holland (1993)