Morpheus: A Degradation Framework for Resilient IoT Systems

Alexander Heß[1], Franz J. Hauck[1], David Mödinger[1], Jakob Pietron[2], Matthias Tichy[2] and Jörg Domaschka[3]

[1]Institute of Distributed Systems, Ulm University, Germany
[2]Institute of Software Engineering and Programming Languages, Ulm University, Germany
[3]Institute of Information Resource Management, Ulm University, Germany

Abstract

Graceful degradation is an established concept to improve the resilience of systems, especially when other resilience mechanisms have failed. Its implementation is often heavily tied to the application code and, thus, cumbersome and error prone. As IoT systems get not only ubiquitous but also critical, reliable graceful degradation would be ideal. In this paper, we present the Morpheus framework that provides a TypeScript-internal DSL to enable a systematic development of degradable IoT systems. The design of the framework is based on the concept of separation of concerns by providing distinct yet linked languages to specify hierarchical components and their connections; the components' operating modes and transfer functions between them; as well as state machines for the specification of the components' behaviour in each operating mode. The operating modes for each component serve as degradation levels. Automatic degradation of a component is triggered in case of failures of connected components. With recovery from underlying failures, the component is automatically upgraded back to a higher level. We illustrate our framework using a simplified prototype of an entrance barrier of a parking garage.

Keywords

Graceful Degradation, Graceful Upgrade, Resilience, Internal DSL, Framework, Internet of Things

1. Introduction

The Internet of Things (IoT) has made huge progress from a generic vision to concrete realisations. IoT systems are becoming more and more ubiquitous and pervasive. Simultaneously, our daily lives depend even more on them, e.g., smart homes, smart cities as well as smart transportation, delivery, and manufacturing processes. By their nature, IoT systems can directly influence the real world and as a consequence their failures can cause harm and even fatalities. Thus, it is important that they are highly resilient and particularly, never enter undefined states.

While IoT systems are conceptually close to distributed applications that have been planned, implemented, and operated for decades, significant differences exist: (i) IoT systems span

MeSS'21: International workshop on MDE for Smart IoT Systems, June 21–25, 2021, Bergen, Norway
alexander.hess@uni-ulm.de (A. Heß); franz.hauck@uni-ulm.de (F. J. Hauck); david.moedinger@uni-ulm.de (D. Mödinger); jakob.pietron@uni-ulm.de (J. Pietron); matthias.tichy@uni-ulm.de (M. Tichy); joerg.domaschka@uni-ulm.de (J. Domaschka)
0000-0001-6837-2861 (A. Heß); 0000-0002-7480-9617 (F. J. Hauck); 0000-0002-5917-2419 (D. Mödinger); 0000-0001-8308-6636 (J. Pietron); 0000-0002-9067-3748 (M. Tichy); 0000-0002-5451-3480 (J. Domaschka)

large geographical regions; (ii) they consist of a large number of heterogeneous, even battery-powered, devices with different capabilities, network bandwidths, and failure characteristics; (iii) depending on the region, replacing failed devices and patching broken software may be a matter of days and weeks instead of minutes and hours as with Web-like distributed systems.

Graceful degradation is a known approach to reduce a system's capabilities in a controlled manner [1]. Degradation thus can step in when classical approaches to fault tolerance are unavailable or can no longer mask failures. While classical approaches to resilience, e.g., replication or checkpointing, can be provided by libraries in an application-agnostic way, degradation has to be interlaced with application code. This makes the design and implementation of a degradable IoT system cumbersome, error prone, time intensive, and hard to test and maintain.

To overcome these limitations, we propose Morpheus, a degradation framework for IoT systems as extension of our previous work [2] within the SORRIR project—an internal DSL for IoT systems built in TypeScript, which allows the definition of components, their ports and connections, as well as the behaviour of components using state machines. The contribution of this paper is an extension that supports developers of IoT systems with the integration of graceful degradation into their systems. Morpheus ensures automatic degradation and upgrade of a component, as soon as developer-defined constraints are met. By design, our DSL ensures the principle of separation of concerns, by splitting the definition of a component's behaviour into different operation modes that can serve as degradation levels.

Section 2 discusses related work, while Section 3 introduces previous work and a running example used throughout the text. Section 4 introduces Morpheus' concepts and illustrates their use via the running example. Section 5 concludes and presents future work.

2. Related Work

With graceful degradation, an IoT system is able to sustain some functionality when faults can no longer be tolerated by other resilience mechanisms [3]: the application copes with partial failures and adapts. Examples include to retain reduced functionality at the edge in case of loss of connectivity between cloud and edge. The degraded service could even abstain from executing certain tasks. Thus, the degradation can affect *function*, *quality* and *performance*.

Graceful degradation approaches are well known for design and operation of distributed embedded systems [4, 5, 1], but have recently also been introduced for IoT scenarios, e.g., for a video surveillance application [6], a smart-office case study [7], and a drone application [8]. Degradation often depends on defining a set of different *service levels* that determine the graduations of quality-of-service (QoS) a system can operate at. This results in an application specific mechanism, as the levels and their interpretation are specific to their application domain.

A degraded system may automatically switch back to the originally desired service level, or at least to a better level, if the cause of the degradation has disappeared or changed. This *graceful upgrade* is addressed in only few related works [9, 4].

Degradation is supported by various modelling approaches such as the Architecture Analysis & Design Language (AADL) and its error annex [10, 11]. Bozzano et al. extend AADL to enable the specification of multiple modes for components to define normal and degraded behaviour using event-data automata [12]. The approach supports formal analysis of such models by

model checking as well as safety analyses like generation of fault trees. A similar approach has been presented in the area of self-adaptive systems where Borda and Koutavas [13] present a formal approach which supports the specification of adaptation transitions between automata. The approach supports the formal verification of safety requirements by a translation to CSP.

3. Background

3.1. Modelling System

Morpheus builds upon our previous work—an internal Domain Specific Language [2] developed in TypeScript [14]—that covers the two relevant aspects: structure and behaviour. The structure of the IoT system is defined as an architecture covering the most relevant structural elements of an architectural description language based on [15]. Particularly, it provides atomic and hierarchical components, ports and connections. Components interact with one another by exchanging events over connections which connect two ports.

Each atomic component contains internal state consisting of a discrete finite state as in automata and local data structures (similar to the state in abstract state machines [16]) as well as a user-developed TypeScript-function which takes this internal state, the contents of the event queue, and updates state and queue. This basic definition of behaviour is complemented by a framework, which allows the specification of state machines using TypeScript data structures and TypeScript as action language and includes an interpreter for these state machines. Hierarchical components do not have their own behaviour, but aggregate the behaviour of their subcomponents to simplify the design of systems, particularly, the definition of degradation.

Our experience as presented in our previous paper [2] has been very positive. The usage of TypeScript as "meta-modelling" and action language streamline the development. Our result fits very nicely into the overall JavaScript/TypeScript ecosystem. For example, we can easily reuse libraries for communication via MQTT and HTTP as well as logging. Another advantage is our ability to use IDEs and all of their features, including debugging facilities.

3.2. Running Example

In order to demonstrate the capabilities of Morpheus, we implemented a physical prototype[1] of a *Smart Entrance Barrier (SEB)* that could be deployed in a parking garage. Customers of the garage reserve a parking spot through an online reservation system using credit card details and license plate number. The *SEB* automatically recognises customers on entrance and exit. Our *SEB* prototype is composed of multiple dependent sub-components: a barrier unit `BU`, a barrier sensor `BS`, a car sensor `CS`, a card reader `CR`, and a camera `CAM`. Further, it uses two external software components: a plate recognition service `PRS` and a parking management system `PMS`.

When a car approaches the `SEB`, the car sensor detects its presence and the camera scans the license plate. The image is then sent to the `PRS`, which extracts the license plate number in text format. This information is used to query the `PMS`, and if a valid reservation is found, the `SEB` will open. In this seamless authentication process with multiple interacting components, a faulty component shall not prevent a car from passing the barrier. The prototype is equipped with

[1]The prototype uses an *ESP8266-based* microcontroller and a *Raspberry Pi Zero-based* camera *OctoCam*.

degradation functionality: Morpheus adjusts the SEB's functionality based on the availability of internal and external components as detailed in Section 4.

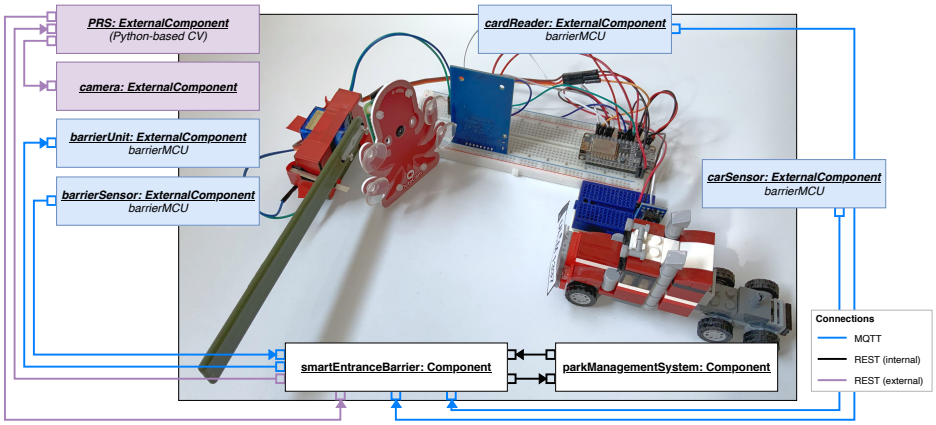

Figure 1: Physical prototype of our running example *Smart Entrance Barrier*. Components and their connections are also shown.

4. Morpheus Degradation

4.1. Operating Mode

In Morpheus, an *operating mode* defines a component's functionality. By default, each component implements the operating mode `Off`, which specifies that the component is currently unable to provide any reasonable functionality. This operating mode is entered if essential sub-components are unreachable or have failed, or the component itself suffers from internal faults. It is required that each component is equipped with at least one additional non-trivial operating mode, which implements the component's intended feature set. Developers can now define further operating modes that provide a graduated subset of the component's capabilities. In order to ensure that the specified functionality can actually be provided, it has to be defined which dependant sub-components have to be available.

Our example uses the following operating modes for the SEB:

- `Off`: The SEB does not provide any reasonable functionality.
- `Blocked`: The SEB is locked, no car can enter or leave, e.g., to lock the garage.
- `Open`: The SEB is open and allows entrance and exit of cars, e.g., for maintenance.
- `Manual`: The SEB requires manual authentication with a credit card for every car.
- `Automatic`: The SEB automatically recognises cars by their licence plates, authenticates the user, and authorises entrance.

Table 4.1 presents the pre-conditions for each non-trivial operating mode implemented by the SEB. We assume that all sub-components and external components only implement the two

Table 1
Pre-conditions for the *SEB*'s implemented operating modes.

Level	CAM	CS	BU	BS	CR	PRS	PMS
L4 - Automatic	On	On	On	On	DC	On	On
L3 - Manual	Off	DC	On	On	On	DC	On
	DC	Off	On	On	On	DC	On
	DC	DC	On	On	On	Off	On
L2 - Open	DC	DC	On	On	DC	DC	DC
L1 - Blocked	Off	DC	On	On	DC	DC	DC
	DC	Off	On	On	DC	DC	DC
	DC	DC	On	On	Off	DC	DC
	DC	DC	On	On	DC	Off	DC
	DC	DC	On	On	DC	DC	Off
L0 - Off	DC	DC	DC	DC	DC	DC	DC

operation modes `On` and `Off`, which means that they are either fully functional or not available at all. Further, `DC` (don't care) entries indicate that the operating mode of the given component is irrelevant. `Automatic` requires all other components to be `On` except the card reader, as it is not used. In case the camera `CAM`, the car sensor `CS` or the plate recognition service `PRS` are faulty, the automatic authentication procedure is unavailable and therefore the SEB should be operated in `Manual`. If further the card reader `CS` is also unavailable, the `SEB` can not provide reasonable service and should be operated in `Blocked`, which effectively closes the gate to the garage. If the parking garage utilises multiple entrance barriers, the garage could proceed to operate with degraded performance. The operating mode `Open` is intended for maintenance or free parking campaigns, because the barrier will remain open until it is manually reconfigured. In case either the barrier unit *BU* or the barrier sensor *BS* suffer from a mechanical failure, none of the pre-conditions can be fulfilled which leads to the operating mode `Off`. In this case only the physical state of the barrier is unknown and has to be inspected by a technician, but the application logic is still in a well-defined state.

Implementation – Framework: A component can be switched into one of the available operating modes either by human operators, software logic, or Morpheus. To realise these features, we first extend the basic type `AbstractState` [2] with an `operatingMode` value, as shown in Listing 1. Additionally, the `DegradableState` is equipped with a `degradationHistory` field that can be used to track the component's degradation behaviour.

We further introduced a new type called `DepdencyFunction` that is used to define the pre-conditions for each operating mode. Essentially, this is a boolean function that takes the components current state and the sub-component's operating modes into account. These are called *Shadow Modes* since it can not be reliably determined wether an external component is unavailable due to a hardware failure or because of an interrupted network connection. We choose to realize this as a generic function, which evaluates declarative conditions like Table 4.1, while still providing great freedom, e.g., to perform arbitrary calculations.

```
interface AbstractState<S, E, P> = {
    readonly state: S,
    readonly events: Event<E, P>[]
}
interface DegradableState<S, E, P, D> extends AbstractState<S, E, P> {
    readonly operatingMode: D;
    readonly degradationHistory: [D, S][];
}
```

Listing 1: The datatypes that are used to store a component's state.

```
type DependencyFunction<S, E, P, D> = (
    state: DegradableState<S, E, P, D>,
    shadowMap: Map<string, string>
) => boolean;
```

Listing 2: The `DependencyFunction` is used to define pre-conditions for an operating mode.

Implementation – Running Example: An example usage of the `DependencyFunction` is shown in Listing 3, which provides a snippet of the dependency map implementation for the SEB. In essence, this map stores the pre-conditions for each operating mode. The snippet shows an example implementation for the operating mode `Automatic`.

```
enum Modes {L0="OFF", L1="BLOCKED", L2="OPEN", L3="MANUAL", L4="AUTOMATIC"};
enum ShadowModes {OFF="OFF", ON="ON"};
enum SubComp {CAM="CAM", CS="CS", BU="BU", BS="BS", CR="CR", PRS="PRS", PMS="PMS"};
const dependencyMap = new Map<Modes, DependencyFunction<any, Events, Ports, Modes»([
  [Modes.L4, (state, shadowMap) => {
    if (shadowMap.get(SubComp.CAM) === ShadowModes.ON &&
        shadowMap.get(SubComp.CS) === ShadowModes.ON &&
        shadowMap.get(SubComp.BU) === ShadowModes.ON &&
        shadowMap.get(SubComp.CR) === ShadowModes.ON &&
        shadowMap.get(SubComp.PRS) === ShadowModes.ON &&
        shadowMap.get(SubComp.PMS) === ShadowModes.ON) {
          return true;
    }
    return false;
    }], // further DependencyFunctions follow here
]);
```

Listing 3: Implementation of a `DependencyFunction` for the operating mode `Automatic`.

4.2. Degradation DAG

At any given time, each component C has a *target operation mode*. Under normal circumstances this is the operating mode that provides the full intended feature set (`Automatic` in our example). Yet, due to external dependencies to other components, C may not be able to provide this ideal mode. In this case, Morpheus picks another operation mode for C for which all dependencies are met. We refer to this operation mode as C's degradation level. For enabling Morpheus to select an appropriate degradation level Morpheus requires a developer to specify a directed acyclic graph (DAG) for each degradable component. The graph connects the component's implemented operating modes, such that each of them is directly or indirectly degradable to operating mode `Off`. Morpheus selects the next degradation level along the graph that fulfils its preconditions, and executes adjacent transitions. Operating modes that are connected through a path that leads to the operating mode `Off` should provide a graduated subset of the component's functionality. Whenever the operating mode of a subcomponent changes, Morpheus first checks if the current operating mode/degradation level can be kept and if not selects a new operation mode to degrade/upgrade to. Further, based on the DAG, it determines the path to reach that state and iteratively degrades/updates along the path from one operation mode to the next until the selected mode has been reached.

Implementation – Framework: For each operation mode associated with its own state machine, a mechanism is needed to not only switch between modes, but also to transfer the state of one state-machine to the state of another. This functionality can be implemented using the `TransferFunction` whose signature is provided in Listing 4. While the main task of a transfer functions is to take one operating mode as an input and output another one, it can be used to perform additional tasks. First, it is possible to save a component's internal state on a stack, such that it can be restored if the operating mode is reached later again. Second, the component's degradation path can be recorded and analysed in order to make fine-grained adjustments to the component state.

```
1 type TransferFunction<S, E, P, D> =
2   (current: DegradableState<S, E, P, D>) => DegradableState<S, E, P, D>;
```

Listing 4: The `TransferFunction` is used for the state transfer between two operating modes.

Morpheus now provides the `updateOperatingMode` function, which automatically selects the optimal operating mode for the component based on the defined pre-conditions, the degradation DAG and the availability of the dependent subcomponents. The implementation of this function is provided in Listing 5. Internally, it evaluates the implemented `DependencyFunctions` in descending order. If a function returns true, it is checked whether the degradation level of the assigned operating mode is higher or lower than the degradation level of the current operating mode. If the degradation level is lower, the internal `degrade` function is executed, which performs a depth-first search on the degradation DAG, that produces an array of `TransferFunctions`. This array of `TransferFunctions` is then iteratively processed, and the function finally returns the degraded state. In case no path could be found the state is returned unaltered. The implemen-

tation of the upgrade function is quite similar. This function is executed if the degradation level of the new operating mode is higher than the degradation level of the current operating mode. However, it additionally guarantees that an upgrade path is only executed if it leads to the component's target operating mode. This property is especially important if only a partial upgrade can be performed, i.e. if the component can be upgraded, but the target operating mode can not be reached yet.

```
function updateOperatingMode<S, E, P, D> (component: Component<E, P, D>,
 currentState: DegradableState<S, E, P, D>, shadowModes: ShadowMap)
 : DegradableState<S, E, P, D> {
    const degradationLevel = [...(component.dependencyMap.keys()].sort().reverse();
    for (const level of degradationLevel) {
        const dependency = component.dependencyMap.get(level);
        if (dependency(currentState, shadowModes)) {
            if (level > currentState.operatingMode) {
                const upgradedState = upgrade(component, currentState, level);
                if (upgradedState != currentState)
                    return upgradedState;
            } else if (level < currentState.operatingMode) {
                const degradedState = degrade(component, currentState, level);
                if (degradedState != currentState)
                    return degradedState;
            } else {
                return currentState;
            }
        }
    }
    return currentState;
}
```

Listing 5: The implementation of the exported updateOperatingMode function

Implementation – Running Example: Figure 2 shows an example DAG for the SEB, based on the conditions from Table 4.1. If the SEB's current operating mode is Automatic and the conditions are no longer true, e.g., the PRS switched to mode Off, then the next possible mode is considered in the DAG. Concretely, operating mode Manual is the first degradation level for target operating mode Automatic. Further, the operating mode Open can only be reached through manual reconfiguration, if it is assumed that Automatic is the standard target operating mode. Listing 6 provides a snippet that shows how the TransferFunction can be used to implement the transition from Automatic to Manual. Here, we utilise the degradationHistory to record the component's current operating mode and internal state.

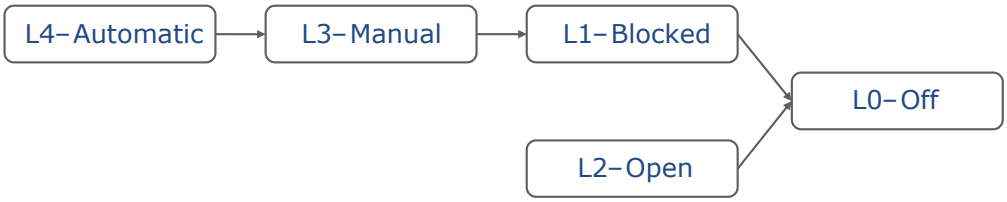

Figure 2: Degradation DAG of the Smart Entrance Barrier.

```
const DegradationDAG: [[OperatingModes, OperatingModes],
 TransferFunction<any, Events, Ports, OperatingModes>][] = [
    [[OperatingModes.L4, OperatingModes.L3], (current) => {
        let newState = { ... current };
        newState.degradationHistory.push([current.operatingMode, current.state]);
        newState.operatingMode = OperatingModes.L3;
        if(current.state.fsm === States.OPEN)
            newState.state.fsm = States.OPEN;
        else
            newState.state.fsm = States.CLOSED;
        return newState;
    }], // Further Transferfunctions follow here
]
```

Listing 6: Snippet of the SEB's degradation DAG implementation.

5. Conclusion

In this paper we presented Morpheus, a framework that aims to assist developers in realising graceful degradation and automatic recovery. Morpheus provides lightweight integration into our previously presented internal DSL based on operating modes of interconnected components. In our previous internal DSL [2] we provided integrated state-space exploration. Currently, this exploration is not realised for Morpheus, as the automatic recovery algorithm and transfer functions complicate the state-transition exploration. Otherwise, Morpheus provides a functional extension of our previous results. Morpheus allows developers to separate their concerns on the levels of functionality and interdependence of components, while providing clear guidance for realising resilience through graceful degradation and upgrade.

Acknowledgments

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany (BMBF grant nr. 01IS18068, SORRIR). Further, this work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 435878599 ; 453895475.

References

[1] M. Glass, M. Lukasiewycz, C. Haubelt, J. Teich, Incorporating graceful degradation into embedded system design, in: Des. Autom. Test in Eur. Conf. & Exh. (DATE), 2009, pp. 320–323. doi:10.1109/DATE.2009.5090681, ISSN: 1558-1101.

[2] M. Tichy, J. Pietron, D. Mödinger, K. Juhnke, F. J. Hauck, Experiences with an internal DSL in the iot domain, in: STAF Worksh. Proc: 4th Worksh. on Model-Driv. Eng. for the Intern. of Things (MDE4IoT), volume 2707 of *CEUR Workshop Proceedings*, 2020. URL: http://ceur-ws.org/Vol-2707/mde4iotpaper1.pdf.

[3] C. Berger, P. Eichhammer, H. P. Reiser, J. Domaschka, F. J. Hauck, G. Habiger, A survey on resilience in the IoT: Taxonomy, classification and discussion of resilience mechanisms, ACM Comp. Surv. (2021). doi:10.1145/3462513, accepted for publication.

[4] W. Nace, P. Koopman, A product family approach to graceful degradation, in: Architecture and Design of Distributed Embedded Systems: IFIP WG10.3/WG10.4/WG10.5 Int. Worksh. on Distr. and Par. Emb. Sys. (DIPES), Springer US, 2001, pp. 131–140. doi:10.1007/978-0-387-35409-5_13.

[5] J. C. Knight, E. A. Strunk, Achieving critical system survivability through software architectures, in: Architecting Dependable Systems II, volume 3069 of *LNCS*, Springer, 2004, pp. 51–78. doi:10.1007/978-3-540-25939-8_3.

[6] H. Chang, A. Hari, S. Mukherjee, T. Lakshman, Bringing the cloud to the edge, in: IEEE Conf. on Comp. Comm. Worksh. (INFOCOM WKSHPS), IEEE, 2014, pp. 346–351.

[7] M. Willocx, I. Bohé, V. Naessens, QoS-by-design in reconfigurable IoT ecosystems, in: 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), IEEE, 2019, pp. 628–632.

[8] M.-K. Yoon, B. Liu, N. Hovakimyan, L. Sha, VirtualDrone: Virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems, in: 8th Int. Conf. on Cyber-Phys. Sys. (ICCPS), ACM, 2017, p. 143–154. doi:10.1145/3055004.3055010.

[9] W. Nace, P. Koopman, A graceful degradation framework for distributed embedded systems, in: Worksh. on Rel. in Emb. Sys. (in Conj. with SRDS), 2001.

[10] Int. Soc. of Automotive Engineers, Architecture analysis and design language annex (AADL), Vol. 1, Annex E: Error model annex, 2006.

[11] J. Delange, P. H. Feiler, Architecture fault modeling with the AADL error-model annex, in: 40th EUROMICRO Conf. on Softw. Eng. and Adv. App. (EUROMICRO-SEAA), IEEE Comp. Soc., 2014, pp. 361–368. doi:10.1109/SEAA.2014.20.

[12] M. Bozzano, A. Cimatti, J. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, Safety, dependability and performance analysis of extended AADL models, Comput. J. 54 (2011) 754–775. doi:10.1093/comjnl/bxq024.

[13] A. Borda, V. Koutavas, Self-adaptive automata, in: 6th Conf. on Formal Meth. in Softw. Eng. (FormaliSE), collocated with ICSE, ACM, 2018, pp. 64–73. doi:10.1145/3193992.3194001.

[14] Microsoft, TypeScript Language Specification v1.8, 2016. URL: http://typescriptlang.org.

[15] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Softw. Eng. 26 (2000) 70–93. doi:10.1109/32.825767.

[16] E. Börger, A. Raschke, Modeling Companion for Software Practitioners, Springer, 2018. doi:10.1007/978-3-662-56641-1.