EPiC Series in Computing
Volume 64, 2019, Pages 98–108
Proceedings of 28th International Conference on Software Engineering and Data Engineering

On Architectural Decay Prediction in Real-Time Software Systems

Aziz Fellah and Ajay Bandi

Northwest Missouri State University
School of Computer Science and Information Systems
Maryville, MO 64468 USA
afellah@nwmissouri.edu ajay@nwmissouri.edu

Abstract

As the number of software applications including the widespread of real-time and embedded systems are constantly increasing and tend to grow in complexity, the architecture tends to decay over the years, leading to the occurrence of a spectrum of defects and bad smells (*i.e.*, instances of architectural decay) that are manifested and sustained over time in a software system's life cycle. Thus, the implemented system is not compliant to the specified architecture and such architectural decay becomes an increasing challenge for the developers. We propose a set of constructive architecture views at different levels of granularity, which monitor and ensure that the modifications made by developers at the implementation level are in compliance with those of the different architectural timed-event elements of real-time systems. Thus, we investigated a set of orthogonal architectural decay paradigms timed-event component decay, timed-event interface decay, timed-event connector decay and timed-event port decay. All of this has led to predicting, forecasting, and detecting architectural decay with a greater degree of structure, abstraction techniques, architecture reconstruction; and hence offered a series of potential effectiveness and enhancement in gaining a deeper understanding of implementation-level bad smells in real-time systems. Furthermore, to support this research towards an effective architectural decay prediction and detection geared towards real-time and embedded systems, we investigated and evaluated the effect of our approach through a real-time Internet of Things (IoT) case study.

1 Introduction

Software systems continue to evolve and tend to grow in complexity over the years, requiring long-term sustainability and maintainability at different levels of granularity from the source code up to the systems' design and architecture. Architecture evolution may be manifested in addressing new emerging requirements, changing technology, capturing and restructuring a component or for any possible and prominent reasons. However, the dynamic interactions and complex aspects of any successful software system at different levels of granularity may eventually lead to architectural decay and violation. Such architectural degeneration manifest themselves in a variety of symptoms that need to be diagnosed and removed. However, the existing studies and literature have heavily focused on the actual code level to detect architectural anomalies and problems rather than on the architectural level itself. Studies have shown that developers who often maintain the software and make changes have no in-depth but only limited knowledge of the available architecture design of the system. Consequently, an important aspect of maintaining a software is understanding not only the code but also the architecture. Thus, we should assess a multifaceted overall operation through a planned series of phases, from the initial architecture to the target implementation, which in turn can ultimately lead to a new version of the system. Empirical evidence has been shown in research that architectural-level decay is a phenomenon that has a negative impact on software systems by causing more decay than the code-level decay.

The two most predominant architectural phenomena that have a negative impact on software systems are erosion and drift. Both architectural erosion and drift are caused by variety of factors. Typically, the erosion problem is caused by the divergence of the software architecture implementation from the software intended's architecture, the technological changes (*i.e.*, OS, hardware, new platforms, programming languages), code complexity, design decisions, time pressure, conflicting requirements, and unintended cyclic dependency among components. In contrast, drift arise through different degradation symptoms caused by the anomalies between the design principles and the architecture styles, multiple responsibilities of a component (*i.e.*, interface, circular dependencies), low-coupling components, violation of modularity principles, and coincidental cohesion. Gradual interweaving of both erosion and drift tend to manifest in software modules and may contribute to the degeneration of the indented software architecture. In fact, architectural erosion and drift can impair each other and software modules and their interactions can become the hub of both erosion and drift symptoms. Thus, translating to a significant decrease in productivity.

In the rest of this paper, we collectively refer to the pair, architectural erosion and drift, as *architectural decay*. In addition, we refer to *architectural smells* as indicators and symptoms of architectural decay. The consequences of the architecture smells (*i.e.*, instances architectural decay) are a spectrum of defects that are manifested in either requirements, design, architecture, or implementation. Thus, risking further decay in a system. A catalog [9] illustrated a number of commonly occurring architectural bad smells such as duplicate components, ambiguous interfaces, cyclic dependencies, and scattered functionality. Instances of such decay negatively affect maintainability, readability, testability, extensibility, reusability and result in side effects induced by code changes and modifications [14]. Over the years, such accumulation of architectural decay would inevitably shorten the life time and evolution of any software system. In addition, there are several other metrics that contribute to architectural decay and violations such as the execution of unreachable (dead) code and code clones which are scattered across the system, hacking, lack of knowledge and deadline constraints in real-time systems. There has been relatively few studies focusing on foundations and techniques to support architecture violation perspectives and provide "standardized" architectural-decay metrics.

In this paper and with no comprehensive literature on architectural decay for real-time and embedded systems, we attempt to pave the way and lay some foundations of architectural decay for real-time systems, an area of research that has not received much attention and could be investigated in various directions. In this work, we are not claiming that we developed a general and conclusive architectural decay framework for real-time and embedded systems, but, and this paper will add values to existing approaches. The paper describes strategies and knowledge needed as well as the rational behind during architectural and code decay of real-time systems. We will shed light on what architects and developers should emphasize when faced with the challenging timed constraint tasks of gaining an understanding of the architecture and debugging real-time source code that should be aligned with that of the architects and designers of the original code. Importantly, the focus of this contribution is on eight set of orthogonal architectural decay paradigms for real-time systems that we introduce and refer to as *timed-event component decay*, *timed-event interface decay*, *timed-event design decay*, *timed-event connector decay*, *timed-event constraint decay*, *timed-event port decay*, *timed-event decay*, and *timed-event tools decay.* The main reason for performing such a partition is to ensure that we do not carry and propagate any "architectural bad smells" to the code level which itself contains several sub-levels and steps. Moreover, we ensure a constructive compliance of the implemented real-time system with the conformance of the architecture phase. We also envision intermediate states of the real-time architecture as being represented by interfaces, ports, connectors, clocks, protocols of interaction, tools, and formal domain descriptive languages. Given the conference's page limitations of the paper, the focus of our contribution in this paper will be on four orthogonal architectural decay paradigms that we introduced earlier. That is, *timed-event component decay*, *timed-event interface decay*, *timed-event connector decay*, and *timed-event port decay.* We refer to the occurrence of smells identified in the previous mentioned architectural decays as *component*, *interface decay*, *connector* and *port smells.* Such architectural decay paradigms would help developers with comprehending systems functionality, understanding code, interweaving abstractions and theories, and building a mental model about a piece of software as well as using effective tools to support architectural, design and implementation decay activities.

With no role in the coding decay aspect, timed-event component smell tasks are grounded on a set of autonomous functional block units that we refer to as timed-event component (TeCmp). The coordination and interaction between TeCmp is fully delegated to a special class of component that we refer to as timed-event connector (TeCnn). These connectors have no relevant role in the computation aspect, but mediate, coordinate, and control interactions among TeCmp at different level of granularity.

The structure of the paper is as follows. In Section 2 we highlight some related and prior work that appeared in literature. Similarly, in Sections 3, 4, and 5 we describe in some details architectural decay prevention and d throughout timed-event component (TeCmp), timed-event interface (TePrt) and timed-event connector (TeCnn), and timed-event port (TePrt), respectively. All four components, TeCmp, TeInt, TeCnn, and TePrt are intrinsically linked and neither of them can be explored in isolation. The characteristics of the IoT irrigation case study system are summarized in section 6. We conclude the paper with some potential discussions in section 7.

2 Related Work

It is not uncommon for developers to inherit and maintain code and therefore they must detect and correct not only the implementation smells in the code but also comprehend the overall architecture decay and smells of the original system. Software maintenance and reuse of large and complex systems are costly, difficult time-consuming and effort-intensive task for developers Software architectural problems and decay (*i.e.,* architectural erosion, decay or smells) are found at multiple levels within almost every software system. These problems are recurring issues in software projects which are constantly maintained during their lifetimes to meet rapidly changing requirements. In this section, we only highlight and outline some prior work and research challenges that have emerged over the years to capture the concept of architectural decay. Consequently, several architectural-level decay approaches have been investigated in a substantial number of empirical studies and research work [8]. Architectural design decisions and problems arising from architectural erosion has been formally defined and treated in [21].

This work introduced RecovAr, a technique that uses the project's available history artifacts and code repository to recover the architectural design decisions embodied in the system. Yuan et al. [27] proposed the concept of source code abridgment, an approach to display the source code in a small space, without compromising the ability of a developer to understand the main functionality of the code. A set of algorithms to automatically detect instances of multiple architectural smell types and their relationships has been developed in [14]. Moreover, the authors described an empirical study and provided consequences of how architectural smells manifest themselves in a system's implementation. As part of related work, a survey of broad techniques and tools to prevent architectural erosion, detect, and repair different types of code-smell have been studied in [22]. Other foundational research and code evolution experiences in the area of architectural decay have been both theoretically and practically addressed in literature, see for example [21], [27], [14], [19], [3], [20], [11], [24], [12]. In order to prevent architectural violations and extend the life time of software systems, a set of measures have been investigated in the literature [19], [4], [3], [25] to monitor the architectural decay process. For example, locate the decay points. Then after pinpointing the violations, perform the reverse process by rolling back (*i.e.*, refactoring) software decay by removing architecture violations and replacing them by implementing decisions that are consistent with the intended architecture. However, coupling the work of the late references with the time dimension shows little or no evidence of working with real-time systems. Pruning dead and complex code and eliminating cycles are also examples of refactoring. Other techniques and frameworks to identify and address architectural decay problems have been illustrated and evaluated in research literature. Particularly we mention, Domain-Specific Languages (DSL) as an alternative to the general purpose languages being used in specifying rules to detect architectural degradation. In a broad sense, DSL is specific to a particular domain and provides a framework to support built-in abstraction. It is an architectural specification language used during the design phase of a software development. Other versions of DSL ideally suited to automated industrial projects such as Structured Control Language (SCL)[10], Dependency Constraint Language (DCL) [23], LogEn [6], and .QL [2]. Several other language-based architectural tools [15], [13], [17], have been customized to tackle software architecture complexity for specific domains including Lattix LDM (Lightweight Dependency Models) [20], IESE SAVE [5], and constructive compliance checking during development and evolution [19].

3 Architectural Decay Prediction and Detection via Timed-Event Component

Architecture decay are found at multiple levels of granularity within software systems. For our work, we want to forecast and prevent this decay, and furthermore identify and correct emerging architectural bad smells. Thus, we propose an architecture structure, which monitors and ensure that the modifications made by developers at the implementation level are in compliance with those of the different architectural elements. In the ontological sense and at the architectural level, timed-event component (TeCmp) express what we believe to know about components of the software system and their interrelationships; and it also captures a common understanding of the application domain. On the other hand and at the semantic level, TeCmp define and disseminate the meanings and aspects of behavior which are expressed in terms of operations and coding affiliated with models, entities, attributes, and tools which are in turn associated through the language at our disposition. In addition, abstraction, modularity and modeling are key factors that enable the prediction, detection, and forecasting of architectural decay. We propose a multitude number of layered abstraction views and models which mimic not only common modeling architectural designs but also improving maintainability and promoting evolution in real-time software systems. In the context of this paper, this high layer of abstraction consists of several constructs such as timed-event components, ports, timed-event connectors, configurations, and interfaces. Importantly, our focus is still on TeCmp and TeCnn. That is, we explicitly express TeCmp and TeCnn, two distinguished component classes, at the implementation level by formally modeling the functionality of TeCnn units and the interaction protocols of TeCnn as timed-event finite automata [7].

The correctness of real-time and embedded systems depends not only on the logical correctness of the computation, but also on the time at which these computations had occurred. Furthermore, the structural decomposition of such systems is embodied in their various components and relationship to each other. Thus, there is a need to promote a software space of design alternatives by putting these pieces together, namely, a collection of application-specific interfaces, ports, timed-event components, port-connectors, and a set of real-time constraints. More specifically, interfaces explicitly describe the services that TeCmp provide and the services that they require from other TeCmp, including their compliance with executions. Ports are the access points in TeCmp where its interfaces and services can be accessed or where it can access another TeCmp interface and services. TeCmp can be atomic and/or composed of layered interactions between a collection of TeCmp that interact with each other to provide new functionalities. TeCnn play a primary role in mediating interactions among TeCmp by providing architectural interaction using different techniques such as queries. Furthermore, they provide different type of services such as data transfer, communication protocols and control transfer. Configurations are a set of association between TeCmp and TeCnn. We assume TeCnn can have at least one TeCmp coupled at each of its ports performing operation requests (*i.e.,* data and control).

Abstraction and modularity are key factors in timed component-based framework that enable the development of re-usable software. They have become an integral part of almost every software system development and are likely to facilitate software verification. We start with various and rigorous levels of abstractions and structures that are refined at each stage of the development before mapping them to programming. For instance in timed-event components and connectors (TeC&C) model, TeCmp architectural abstractions expose a high-level of the structure of the system, including TeCmp logical abstractions. Also, TeCmp functional abstractions reflect the functionality, encapsulates details, and verifies functional properties. On the other hand, TeCnn data- and control-flow abstractions propose categorization spaces of data types and control the flow of imposed conditions. TeCnn communication and synchronization abstraction styles support for instance, protocols and enforce synchronous, asynchronous requests. TeCmp and TeCnn timing abstractions and properties address several issues of real-time systems throughout modeling formalism.

Component Architectural's Abstraction

Abstraction can take many forms and dimensions to serve various purposes in software development. In the context of this work, we propose two different levels of abstraction. A horizontal abstraction that studies component architecture at a very high level of abstraction such as TeCmp's functionality, ports, interfaces, and TeCnn. Figure 1 views a prism rectangle box with special components, TeCmp, ports, TeCnn, interfaces and a "time event clock".

4 Architectural Decay Prediction and Detection via Timed-Event Interface

We define three types of interaction decay interfaces, *get-interface decay*, *put-interface decay*, and *syn-interface decay* where get-interfaces are required and put-interfaces are provided in-

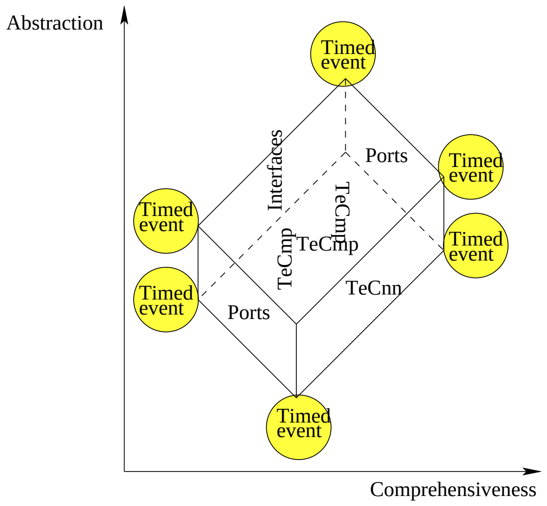

Figure 1: Architectural dimension views through TeCmp, TeCnn, ports and interfaces.

terfaces by TeCmp. However, there may be complicated by synchronization decay constraints between two or more interfaces of a single TeCmp, then we complement TeCmp with a third type of interface that we refer to as *syn-interface*. Two TeCmp, $\mathcal{C}_1$ and $\mathcal{C}_2$, may interact synchronously through syn-interface. Figure 2 shows a timed-event component system with of three timed-event components, C_1, C_2, and C_3 that communicate through their respective ports, interfaces, and TConn.

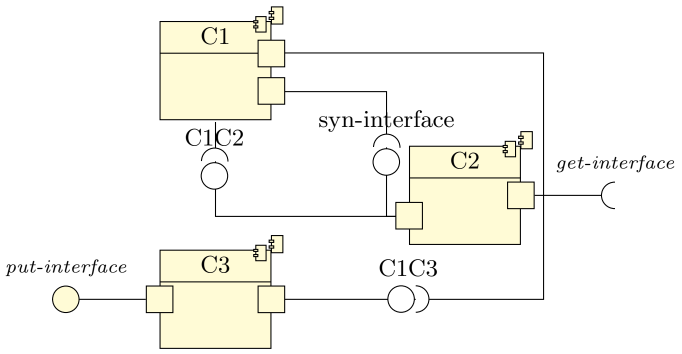

Figure 2: A Time-event component-based system with three composed TeCmp, C_1, C_2, and C_3 communicating via encapsulated ports/TeCnn, and interfaces.

Now, we can define the following relations between TeCmp. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two TeCmp, we define the following TeCmp decay relationships:

1) TeCmp *Inheritance Decay*

We say two TeCmp, $\mathcal{C}_2$ and $\mathcal{C}_1$, have an inheritance decay relation if $\mathcal{C}_2$ inherits all the properties of $\mathcal{C}_1$. In addition, $\mathcal{C}_1$ may have more interaction interfaces and all the inherited interaction interfaces of $\mathcal{C}_2$ work exactly the same way as those of $\mathcal{C}_1$.

2) TeCmp *Association Decay*
We say two TeCmp, $\mathcal{C}_1$ and $\mathcal{C}_2$, have an association decay relation if they have at least on interaction interface.

3) TeCmp *Aggregation Decay*
We say two TeCmp, $\mathcal{C}_1$ and $\mathcal{C}_2$, have an aggregation decay relation if $\mathcal{C}_1$ is a subset of $\mathcal{C}_2$. In addition, a single TeCmp can be aggregated by several TeCmp. The aggregate TeCmp has all the interaction interfaces of its TeCmp.

4) TeCmp *Composition Decay*
A composition is the combination of two or more TeCmp at different levels of abstraction to achieve modularity and decomposition of TeCmp using various programming languages or composition tools as defined by the TeCmp infrastructure. Let $\mathcal{C}_1, \mathcal{C}'_1, \mathcal{C}_2, \mathcal{C}'_2$ be four TeCmp. Let the operators $\equiv$ and $\times$ be the equivalence operator and composition decay operator in the semantic context, respectively. Then, if $\mathcal{C}_1 \equiv \mathcal{C}'_1$ and $\mathcal{C}_2 \equiv \mathcal{C}'_2$ implies $\mathcal{C}_1 \times \mathcal{C}_2 \equiv \mathcal{C}'_1 \times \mathcal{C}'_2$.

5) TeCmp *Encapsulation Decay*
We say that TeCmp $\mathcal{C}_1$ exhibits functional encapsulation decay if $\mathcal{C}_1$ hides its details while exposing a well-defined interface through its ports. Furthermore, embedded TeCmp may occur at different levels of abstraction and could potentially foresee what we call recursive encapsulation decay, a fundamental scheme that should be avoided in the architecture.

5 Architectural Decay Prediction and Detection via Dual Timed-Event Connector and Port

Time Dimension Decay
We leverage the time logic and dimension structures of [26], [1] to describe real-time interactive or concurrent systems in this work. Importantly, we consider that the time dependent behavior of any TeCmp is an important aspect of the system's requirements, enforced by the component itself and coordinated by TeCnn. To develop a uniform timing framework, we consider the absolute time which could be modeled using a global clock. The concept of time is very important in real-time and embedded systems. Therefore, we predict that time is an important architectural and design element that most likely to undergo decay. Moreover, we need to predict and detect decay involving issues and problems attributed to different architectural elements. That is, timed-event connector (TeCnn) and timed-event port (TePrt).

Let $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\} \subseteq$ TeCmp be a finite set of timed-event component instances where $|\mathcal{C}| = n$. Let $\Omega = (\mathcal{T}, \mathcal{E})$, where $\mathcal{T} = \{t_1, t_2, \ldots, t_k\}$ is a set of points in the time domain and $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ is a set of events in the event domain. For convenient, we assume $|\mathcal{T}| = |\mathcal{E}| = k$. Let $\prec$ be a strict partial order precedence relation over $\mathcal{T}$. Let $\mathcal{C}_1(e_1, t_1)$, $\mathcal{C}_2(e_2, t_2)$, and $\mathcal{C}_3(e_3, t_3)$ indicate that $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ are being active on the occurrence of event e_i at time t_i, respectively where i=$i = 1 \ldots, n$. We define the timed-event dimension structure over TeCmp as a tuple in the form $\mathcal{C}(\mathcal{E}, \mathcal{T})$ that satisfies the following properties:

(*i)* For all $e \in \mathcal{E}$, if $\mathcal{C}_1(e, t_1) \prec \mathcal{C}_2(e, t_2)$ and $\mathcal{C}_2(e, t_2) \prec \mathcal{C}_3(e, t_3)$ then $\mathcal{C}_1(e, t_1) \prec \mathcal{C}_3(e, t_3)$.

(*ii)* For all $e \in \mathcal{E}$ and $t \in \mathcal{T}$, $\mathcal{C}_i(e_i, t_i) \not\prec \mathcal{C}_i(e_1, t_i)$, $i = 1, \ldots, n$.

(*iii)* For all $e \in \mathcal{E}$ and $t \in \mathcal{T}$, if $\mathcal{C}_1(e_1, t_1) \prec \mathcal{C}_2(e_2, t_2)$ then $\mathcal{C}_2(e_2, t_2) \not\prec \mathcal{C}_1(e_1, t_1)$.

(*vi)* For all $\mathcal{C}_i(e, t)$ and $\mathcal{C}_j(e, t)$, if $\mathcal{C}_i(e, t) \not\prec \mathcal{C}_j(e, t)$ then $\mathcal{C}_i$ and $\mathcal{C}_j$ are interpreted as being concurrent, for all $e \in \mathcal{E}$ and $t \in \mathcal{T}$, and where $i, j = 1, \ldots, n$.

The external view of the port model is based on the pipe-and-filter architectural style with consists of a set of *data* and *control port* groups. In addition and for various purpose, we assume there is one extra internal group ports that we refer to as *special ports*. The data port group is explicitly divided into input and output data ports. Similarly, the control port group is explicitly divided into input and output control ports. Both the data and control ports are provided by default for each port. However, other types of variables such as monitoring and controlling ports can be an intrinsic part of the internal port and this depending on the application domain. In the context of this paper, we define a port signature $\mathcal{S}$ as follows:

Definition 5.1. *A timed-event port signature is a quintuple $\mathcal{S}$ = (Event, Type, Data, Control, Time), where $Event = \{In, Out, Spec\}$ and In, Out, Spec are the set of input, output, and special ports respectively; Type is a finite set of type names, Data and Control are sets of data and control values, respectively. Time is a set of point structure of time, modeled by a global clock. Moreover, $(In \cap Out \cap Spec) = \emptyset$, and the set of data and control values is disjoint.*

Now, borrowing from the syntax and semantics of components and connectors views [16, 18], we formalize the structure of the timed-event component and connector model (TeC&C) model by not focusing on the interfaces defined for the ports, but rather on the relation between the different pieces of the TeC&C model.

Definition 5.2. *A timed-event component and connector* TeC&C *model is a sextuple structure $\mathbb{CC}$ = ($\mathcal{C}$, $\widehat{\mathcal{C}}$, $\mathcal{P}$, $\mathcal{S}$, δ_p, δ_t) where*

(*i*) $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\} \subseteq$ TeCmp *is a finite set of timed event component instances where $|\mathcal{C}| = n$.*

(*ii*) $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ *is a finite set port instances where $|\mathcal{P}| = m$*

(*iii*) $\mathcal{S} = \{s_1, s_2, \ldots, s_m\} \subseteq$ *port signatures is a finite set of port signature instances where $|\mathcal{S}| = |\mathcal{P}| = m$.*

(*iv*) $\widehat{\mathcal{C}} = \{\widehat{\mathcal{C}}_1, \widehat{\mathcal{C}}_2, \ldots, \widehat{\mathcal{C}}_q\} \subseteq$ TeCnn *is a finite set of connector instances which are used to capture pathways of events (data transfer flow and control flow) between $\mathcal{C}_i$, $i = 1 \ldots n$. ($|\widehat{\mathcal{C}}| << |\mathcal{C}|$).*

(*v*) δ_p*: $\mathcal{C} \times \mathcal{P} \to \mathcal{C} \times \mathcal{P}$. That is, $\delta_p(\mathcal{C}_i, p_j) \subseteq \mathcal{P}$, for all $i = 1 \ldots n$ and $j = 1 \ldots m$.*

(*vi*) δ_t*: $\mathcal{P} \times \mathcal{S} \to \mathcal{P} \times \mathcal{S}$. That is, $\delta_t(p_j, s_j) \in (\mathcal{P} \times \mathcal{S})$, for all $j = 1 \ldots m$.*

6 IoT Case Study

We describe a case study that has been conducted and implemented on an IoT irrigation embedded system which controls a water solenoid valve for controlling a drip irrigation system using Arduino and Raspberry Pi infrastructure. For the experiment, we selected and expand the recent IoT project of three of our graduate students as the basis of our case study by running a variety of experiments to test the proposed theoretical work. The experiment was tested on several events such as moisture, temperature, and humidity. The system is able to deliver water to the plants based on the moisture of the soil, temperature and humidity of the day which are obtained through DHT sensors. Importantly, we use a real-time clock that allows the system to set the start of the irrigation system based on the moisture and temperature levels. Furthermore, the system can also start and stop at the specified time intervals to control the water management. In the experiment, the IoT system is controlled by the real-time status of the soil moisture, atmospheric conditions, and on the real time clock to adjust the irrigation scheduling through time intervals. In this IoT-based system a strong emphasis is put on timed-event components of the system and empirical evaluation.

In analogical mapping, our abstract model domain of study, timed-event component-based can be mapped into the real-time target irrigation domain. That is, soil moisture, temperature and humidity sensors send real data to the microcontroller which is considered as the central TeC&C architectural information gateway. The microcontroller can be monitored and operated via WiFi using a Web browser, or managed by the user through a mobile application. The TeCmp sprinkler controller ensures uniform distribution of water to all parts of that plant and it is monitored by the microcontroller. In addition, the TeCmp sprinkler may be switched off and on once the soil moisture sensor has reached the appropriate threshold value. We may consider that DHT moisture, temperature, humidity sensors are equipped with some ports communicating with various TeCmp. The coordination and interaction between various TeCmp is fully delegated to a special class of component that we referred to as TeCnn. These connectors have no relevant role in the irrigation aspect, but mediate, coordinate, and control interactions among various TeCmp phontons. In addition, the data of sensors is displayed in a graphical format, analyzed and visualized by the end-user. Due to the conference's page limit, the authors may also be contacted for a full and detailed version of this case study.

7 Conclusion

Our work revealed an apparent lack of foundations in the literature that relate to architectural decay for real-time and embedded systems. An area of research that has not received much attention and could be investigated in various directions. We investigated a set of orthogonal timed-event architectural decay paradigms, timed-event component decay, timed-event interface decay, timed-event connector decay, and timed-event port decay, which led to predicting, detecting, and forecasting architectural decay with a greater degree of structure, abstraction techniques, architecture reconstruction; and hence offered a series of potential effectiveness and enhancement in gaining a deeper understanding of architectural bad smells in real-time systems. First, we mainly rely on architectural levels and time dimensions which have been explicitly targeted in our work and how they are clearly manifested in a real-time systems's implementation. We have examined the relationships between architectural timed-event component, timed-event interface, timed-event connectors, and timed-event ports. Such refinement and analysis from the architectural-level view to the implementation-level view is significantly represented in the source code. Furthermore, we have performed an empirical IoT irrigation case study in order to complement and provide a qualitative base and characterization of our approach to architectural decay and software evolution.

References

[1] T. Ben-Nun, A. S. Jakobovit, and T. Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Procedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS), Montreal, Canada*, 2018.

[2] O. de Moor. Keynote address: .ql for source code analysis. In *Proceedings of the 7th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 3–14, 2007.

[3] L. de Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[4] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[5] S. DuszynskiJens, K. Knodel, and M. Lindvall. Save: Software architecture visualization and evaluation. In *Proceedings of the 13th Euromicro Conference on Software Maintenance and Reengineering, CSMR*, pages 167–176, 2009.

[6] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 391–400, 2018.

[7] A. Fellah. Timed event systems and automata. 2011.

[8] J. Garcia, E. Kouroshfar, and S. Malek. Architectural decay prediction from evolutionary history of software. Technical report, University of California, Irvine, CA, 2018.

[9] J. Garcia, D. Popescu, and G. Edwardsand N. Medvidovic. Toward a catalogue of architectural bad smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems, QoSA '09, Springer-Verlag, East Stroudsburg, PA USA*, pages 146–162, 2009.

[10] D. Hou and H. J. Hoover. Using scl to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[11] Semmle Inc. Semmle's on-demand analytics of software assets.

[12] J. Knodel, D. Muthig, M. Naab, and M. Lindvall. Static evaluation of software architectures. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 167–176, 2006.

[13] P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, and A. Tang. The road ahead for architectural languages. *IEEE Software*, prePrint:1, 01 2014.

[14] D. Minh Le, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural decay in open-source software. In *Proceedings 2018 IEEE International Conference on Software Architecture (ICSA)*, pages 176–17609, 2018.

[15] J. Lenhard, M. Blom, and S. Herold. Exploring the suitability of source code metrics for indicating architectural inconsistencies. *Software Quality*, 27(1):241–274, 2019.

[16] S. Maoz, N. Pomerantz, and B. Rumpe. Synthesis of component and connector models from crosscutting views. *B. Meyer and L. Baresi and M. Mezini, editors, (ESEC/SIGSOFT FSE)*, pages 444–454, 2010.

[17] N. Mendonça, M. T. Valente, L. Passos, and R. Diniz. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89, 09/10 2010.

[18] S. Maoz nd N. Pomerantz, J. O. Ringert, and R. Shalom. Why is my component and connector views specification unsatisfiable. In *Procedings of ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, pages 134–144, 2017.

[19] L. Passos, R. Terra, R. Diniz, M.T. Valente, and N. Mendona. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82 – 89, 2010.

[20] N. Sangal, E. Jordan, E. Sinha, and V. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications IEEE International Conference on Source Code Analysis and Manipulation (ACM)*, pages 167–176, 2005.

[21] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic. Recovering architectural design decisions. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 95–9509, 2018.

[22] L. Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[23] R. Terra and M. T. Valente. A dependency constraintlanguage to manage object-oriented software architectures. *Software: Practice and Experience*, 32(12):1073–1094, 2009.

[24] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *Proceedings of the 16th European Conference on Software*

Maintenance and Reengineering (CSMR), pages 167–176, 2012.

[25] R. Terra, M.T. Valente†, K. Czarnecki, and R. S. Bigonha. Recommending refactorings to reverse software architecture erosion. In *Proceedings of the 16th Euromicro Conference on Software Maintenance and Reengineering, (CSMR)*, 2012.

[26] S. Yu. The time dimension of computation models. *Where Mathematics, Computer Science, Linguistics and Biology Meet*, pages 162–172, 2001.

[27] B. Yuan, V. Murali, and C. Jermaine. Abridging source code. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, volume 57, page 16 pages, 2017.