8VO CONGRESO NACIONAL

INGENIERÍA INFORMÁTICA / SISTEMAS DE INFORMACIÓN

VIRTUAL CONAIISI 2020

50 UTN ANIVERSARIO FACULTAD REGIONAL SAN FRANCISCO

www.sanfrancisco.utn.edu.ar

05 06 NOV.

RIISIC

confedi Consejo Federal de Decanos de Ingeniería República Argentina

150 ING la Argentina celebra su ingeniería 1870-2020

Octavo Congreso Nacional de Ingeniería Informática/Sistemas de información

CONAIISI

5 y 6 de noviembre de 2020

Universidad Tecnológica Nacional, Facultad Regional San Francisco

Memoria de Trabajos

Claudia Verino, Juan Carlos Calloni, Gabriel Cerutti, Alfonsina E. Andreatta
(Compiladores)

San Francisco, Córdoba - Argentina, Marzo de 2021

Octavo Congreso Nacional de Ingeniería Informática/Sistemas de Información: CONAIISI.
5 y 6 de noviembre de 2020 / Claudia Verino, Juan Carlos Calloni, Gabriel Cerutti, Alfonsina E. Andreatta; compilado por Claudia Verino , Juan Carlos Calloni, Gabriel Cerutti, Alfonsina E. Andreatta. - 1a ed. -
Ciudad Autónoma de Buenos Aires: Universidad Tecnológica Nacional.
Facultad Regional San Francisco, 2021.
Libro digital, PDF

Archivo Digital: descarga
ISBN 978-950-42-0202-8

1. Sistemas de Información. 2. Ingeniería Informática. I. Verino, Caludia, Calloni, Juan Carlos, Cerutti, Gabriel, Andreatta, Alfonsina E. comp.
CDD 004.07

Octavo Congreso Nacional de Ingeniería Informática/Sistemas de información
CONAIISI
5 y 6 de noviembre de 2020
Universidad Tecnológica Nacional, Facultad Regional San Francisco

Memorias de trabajo

Diseño de Tapa: María Laura Vaudagna

ISBN 978-950-42-0202-8

Auspiciantes:

Declarative Specification and Verification of Modern Software Architecture Patterns

Fernando Asteasuain[1,2], Martín Miguel Machuca [1].
[1] Universidad Nacional de Avellaneda {fasteasuain,mmmachuca}@undav.edu.ar
[2]Universidad Abierta Interamericana – Centro de Altos Estudio CAETI

Abstract

In this work we explore FVS as an Architectural Description Language (ADL) with the possibility to perform formal verification of architectural behavior. We modeled and specified a collection of architectural patterns including typical ones such as publish/subscribe or blackboard as well as some more modern ones in emergent technologies such as embedded software or cloud computing. Using a model checker tool we were able to formally verify architectural patterns in a concrete case of study: a server's room monitoring system. The results show the potential of our work in the ADL's domain.

Keywords: Software Architecture, Architectural patterns, Formal Verification.

1. Introduction

The consolidation of Software Architectures [1-2] can be considered one of the most relevant milestones in the Software Engineering community in the past twenty five years. In essence, a software architecture view consists of a high-level representation of the behavior of the system to be developed, exhibiting the most relevant interactions between all the elements of interest. In general, this behavior is depicted using artifacts so called **Components** which communicates with each other by employing **Connectors**. Connectors define the protocol which establishes the communication rules between two or more components [3].

Typical software architecture requirements' transcend functional behavior including concepts as availability, performance, security or usability, just to mention a few of them. For example, it is far from being useful an ATM distributed system where transactions are performed correctly (i.e., functional requirements' are satisfied) but each transactions take 5 seconds to be approved, failing to satisfy performance issues. In this sense, software engineering tools and techniques must be provided in order to reason, explore, model, specify and verify architecture behavior [4,5,6,7,8].

The introduction and usage of architecture patterns [8-13] certainly leverage the potential of architectural elements. In the same way that the widely known object design patterns [14] architecture pattern consists of a template expressing a recurrent solution to common problems and therefore providing a common vocabulary and ontology to communicate and denote architectural behavior. Known architecture patterns are for example *Client/Server, Broadcast, Pipe and Filter, Layered Systems,* and others. In few words, an architecture pattern details the components involved, how they interact, and restrictions about the expected behavior of their combination.

Architectural behavior has been generally addressed by domain specific languages known as ADLs (Arquitecture Description Languages) [11,15,16,17,18]. Although the meaningful advances achieved by these and many other approaches there are still appealing issues that need to be solved and addressed [6-8]. The dynamic and intrinsic nature of architectural behavioral is certainly one of the challenges involved. It is not unusual that relevant architectural behavior only arise while executing the system, for example a server connection that only take place under certain conditions that can be determined exclusively in runtime. Another important factor is granularity. Several code-level relevant events might be needed to build one architectural event. For example, multiple settings and validations may be required to open a socket where a server listens to its requests. Finally, modern systems such as those based on *micro services* [10], *embedded systems* [12] or and BIG DATA systems based on *Cloud Computing* [9] push software architecture's limits to its boundaries creating new and emerging architectures such as Cloud Computing [19] or intensive and dense distributed systems [20-21]. Under these conditions it can be stated that more powerful and expressive ADLs are still required to address these crucial problems. One possible way to achieve this goal is to focus on architectural patterns, since there is a powerful synergy between architectural patterns and ADLs [8,22].

Given this context in this work we explore FVS (Feather Weight Visual Scenarios) [23] as an ADL to model, specify and verify architectural patterns. FVS is a very simple yet powerful specification language based on graphical scenarios originally conceived to model the

expected behavior of reactive systems. Given its flexibility is has been explored in the past as a language to denote the behavior of architectural connectors [24]. We now build on the top of that work expanding FVS architectural behavior coverage including the specification and validation of architectural patterns, including modern ones such as embedded systems patterns, micro services pattern or cloud computing patterns. To validate our proposal we provide in this work the specification of some representative patterns such as Blackboard, publish/subscribe, microservices oriented patterns, and some distributed patterns. We formally verify their behavior employing an architectural relevant case of study using FVS specifications as input into a model checker [25]. We believe the obtained results consolidate FVS as a powerful and expressive ADL's.

The rest of the paper is structured as follows. Section 2 briefly presents the main features of the FVS specification language. Section 3 presents the specification of some architectural patterns whereas Section 4 presents their validation on a case of study. Section 5 discusses some related and future work while Section 6 highlights the conclusions of this paper.

2. Feather Weight Visual Scenarios

In this section we will informally describe the standing features of FVS [23]. The reader is referred to [23] for a formal characterization of the language. FVS is a graphical language based on scenarios. Scenarios are partial order of events, consisting of points, which are labeled with a logic formula expressing the possible events occurring at that point, and arrows connecting them. An arrow between two points indicates precedence of the source with respect to the destination: for instance, in Figure 1-a A-event precedes B-event. We use an abbreviation for a frequent sub-pattern: a certain point represents the next occurrence of an event after another. The abbreviation is a second (open) arrow near the destination point. For example, in Figure 1-b the scenario captures the very next B-event following an A-event, and not any other B-event. Conversely, to represent the previous occurrence of a (source) event, there is a symmetrical notation: an open arrow near the source extreme. For example, in Figure 1-c the scenario captures the immediate previous occurrence of a B-event from the occurrence of the A-event, and not any other B-event. Events labeling an arrow are interpreted as forbidden events between both points. In Figure 1-d A-event precedes B-event such that C-event does not occur between them. FVS features aliasing between points. Scenario in 1-e indicates that a point labeled with A is also labeled with A ^ B. It is worth noticing that A-event is repeated on the labeling of the second point just because of FVS formal syntaxes [23]. Finally, two special points are introduced as delimiters to denote the beginning and the end of an execution. These are shown in Figure 1-f.

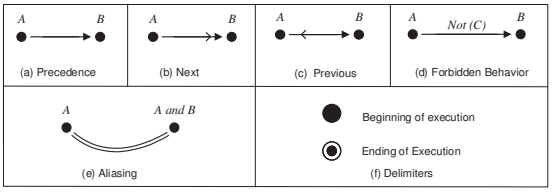

Figure 1. Basic Elements in FVS

We now introduce the concept of FVS rules, a core concept in the language. Roughly speaking, a rule is divided into two parts: a scenario playing the role of an antecedent and at least one scenario playing the role of a consequent. The intuition is that whenever a trace "matches" a given antecedent scenario, then it must also match at least one of the consequents. In other words, rules take the form of an implication: an antecedent scenario and one or more consequent scenarios. Graphically, the antecedent is shown in black, and consequents in grey. Since a rule can feature more than one consequent, elements which do not belong to the antecedent scenario are numbered to identify the consequent they belong to. Two examples are shown in Figure 2 modeling the behavior of a client-server system. The rule in the top of Figure 2 establishes that every request received by a server must be answered, either accepting the request (consequent 1) or denying it (consequent 2). The rule at the bottom of Figure 2 dictates that every granted request must be logged due to auditing requirements.

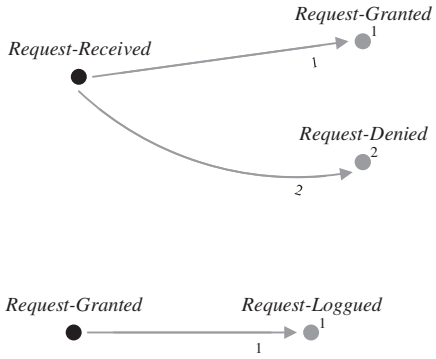

Figure 2. FVS rules' examples

3. Architectural Patterns in FVS

In this section we will show the specification of some representative architectural patterns, namely the blackboard pattern and the publish/subscribe pattern, three architectural patterns for embedded software and a gateway pattern for microservices and cloud-oriented arquitectures. These patterns will be addressed in the next subsections. The specification of these patterns is taken from several approaches in the literature in different shapes and flavors, from formal specification given as temporal logics formulas to specifications indicated in natural language [8-10,12]. This might reflect the

flexibility of our approach since it can be adapted to manifold circumstances. Finally, Section 3.5 presents some reflections about the given description and verification of the behavior of architectural patterns in FVS.

3.1 Publish/Subscribe Pattern

In this pattern three type of components are involved: components who publish information or events (depending on the context of the system), components that receive those pieces of information and a intermediate component playing the role of a intelligent buffer that receive the information and send it to those components who claimed interest in that information. That is, in order to receive the information components must first explicitly show interest in receiving it by subscribing to the intermediate buffer. In this way, components that produce the data are decoupled from the components that consume the data. Three main rules shaping the behavior of this pattern according to the specification given in [8] are the following:

- R1: The Publish/Subscribe instantiation must be unique, acting like a *Singleton* [14].
- R2: Whenever a component sends a data to the intermediate buffer, the information must be received. That is, everything components produce it is eventually published.
- R3: Every time a component who has subscribed to receive some type of information and a component publish one of that type, then the interested component eventually receive that information. In other words, subscribes receive their data.

The FVS rules shown in Figure 3 capture these Publish/Subscribe requirements. In the rule in the top of Figure 3 the *BB-On* event stands for the activation of the Blackboard, which must be unique during all the computation according to *R1*. The rule in the middle says that every component *C* that publish a certain data C_i it is received in the intermediate buffer. Finally, the last rule addresses requirement *R3*: if a component C_n is subscribed to data from a component C_i and information C_i is received, then component C_n receives the information.

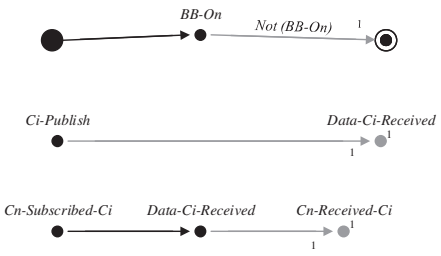

Figure 3. Publish/Subscribe rules in FVS

3.2 Blackboard Pattern

As in the previous case we follow the specification for this pattern from [8]. This pattern is usually used for the task of collaborative problem solving, i.e., a set of components work together to solve an overall, complex problem that usually do not have a feasible or known solution [8]. Three type of components are present in this pattern: the *blackboard* itself, who plays the role of a publish/subscribe, the *knowledge* components that solve the different parts of the main problem and a *control* component, who centralizes and orchestrates the interaction between all the components. The specification detailed in [8] includes the following requirements describing the behavior of this pattern:

- R1: All the requirements given in Section 3.1 for the publish/subscribe pattern since the blackboard component behaves as a publish/subscribe.
- R2: The *Knowledge* components solve all the tasks they receive.
- R3: Every task published by a *Control* component is eventually assigned.
- R4: Every solution published by a *Knowledge* component is eventually received by the *Control* Component.

Figure 4 shows the FVS rules modeling requirements R2 (in the top of the Figure), R3 (the rule in the middle of the figure) and R4 (the rule in the bottom of the Figure). Note that requirements R1 denoting the behavior of a *Publish/Subscribe* are shown in Figure 3.

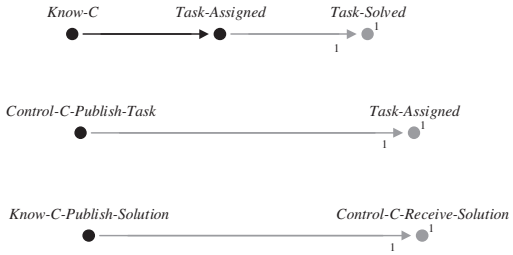

Figure 4. Rules for the Blackboard Pattern

3.3 Hardware Abstraction Layer

This pattern belongs to the *Distribution* category for embedded software architectural patterns identified in [12]. The application context of this pattern is the following: An embedded control system controls hardware devices (e.g. sensors and actuators) and the problem is how the engineer can control different kinds of hardware within application, without needing to know the details of the hardware devices. The solution provided by the pattern is the introduction of a hardware abstraction layer interface containing functions to control all the hardware devices.

The requirement shaping the behavior of this pattern (see Figure 5) is very simple: every hardware request must be handled by the interface.

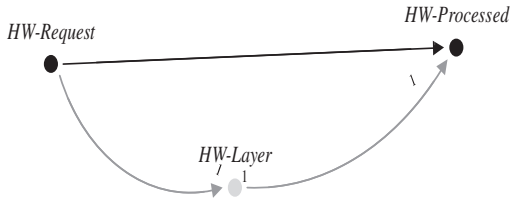

Figure 5. An FVS rule for the HW Abstraction Pattern

3.4 Fixed Process Allocation and Scheduling Hardware Abstraction Layer

This pattern belongs to the *Real Time* category for embedded software architectural patterns identified in [12]. Its application context is the following: An embedded control system requiring several processes. The scheduling is not pre-emptive because there are no priorities between the processes. However, some tasks may need more frequent repetition and some tasks may need more processing time. The pattern allows creating processes with predictable time behavior to address the problem following the next conditions:

1. All the resources needed by a process must be granted before it is started, and no initialization time must be given.
2. Each process is divided into executable codes and for each one their worst-case scenario is calculated.
3. Once a process is started there are no other external interruption allowed until it finishes.

The FVS rules in Figure 6 capture the conditions needed to this particular pattern for embedded systems. The rule in the top of Figure 6 simply states that if a process is started, then it had obtained its resources and no initialization had occurred. The rules in the middle of Figure 6 illustrate the behavior imposed by the second condition. Finally, the rule in the top of Figure 6 tackles the third condition.

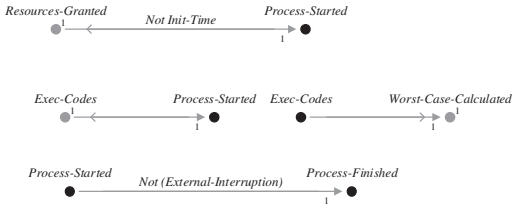

Figure 6. Rules for Fixed Scheduling Pattern

3.5 Virtual TimeStamps Pattern

This pattern belongs to the *Fault Tolerance* category for embedded software architectural patterns identified in [12]. Its application context is the following: *How to know the order of events in a distributed embedded system?*. The pattern provides a solution by adding a *TimeStamp* component synchronized by a clock to tag all the events of interest. Rules in Figure 7 introduce the behavior of this pattern. The rule in the top of the Figure simply adds a timestamp to every event whereas the rule at the bottom synchronizes the timestamp component with the clock of the system.

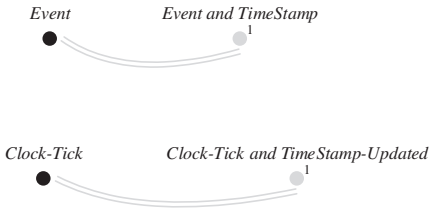

Figure 7. Rules for TimeStamps Pattern

3.6 The API and Cloud-Computing Gateway Pattern

The API-Gateway pattern belongs to the microservices domain and it is fully specified in [10]. As it is explained in [10] the pattern is the entry point of the system that routes the requests to the appropriate microservices, also invoking multiple microservices and aggregating results. It can also be responsible for different tasks such as authentication. Following the specification in [10] we include the following requirements:

- R1: Every request received by any of the microservices of the system must be communicated only by the gateway.
- R2: Every aggregated result transmission must be done through the gateway.
- R3: The gateway must implement the authentication feature.

Rules in Figure 8 introduce the behavior of this pattern. The rule in the top of the Figure 8 claims that every client's request to a micro service must be delivered exclusively by employing the gateway channel (Requirement R1). The second rule focuses on requirement R2: if a result is aggregated then the *API-Gateway* event occurred in the past (i.e., communication took place only by the Gateway). Finally, the rule at the bottom of Figure 8 addresses requirement R3: a micro service will receive a task only if an Authenticated event occurred previously (that is, the *Gateway* performed the authentication controls.)

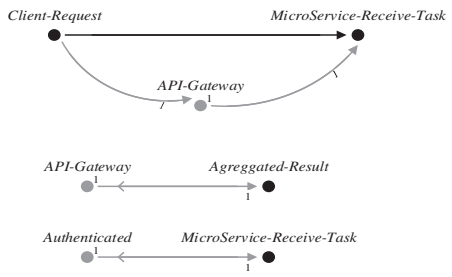

Figure 8. API-Gateway Pattern Behavior

It is worth noticing that these requirements are general enough to match another pattern from the cloud computing domain: the *Cloud component gateway pattern* [9]. By just replacing microservices events for the events of interest in the cloud architecture system this pattern constitute a possible specification of the cloud component gateway pattern.

3.6 Some observations

FVS was able to fully specify pattern behavior in different kind of domains, from traditional architectural patterns such as blackboard or publish/subscribe to more modern ones such as gateways for cloud computing and microservices architectures as well as architectural patterns for embedded software. This may exhibit the flexibility and expressive power of our approach. Nonetheless, a more comprehensive comparison against other approaches is needed to further validate this observation.

4. Case Study: A System for Monitoring a Servers' Room in a University.

In this section we will verify the behavior of some of the previously mentioned patterns in a concrete case of study. Since FVS rules can be translated into Büchi automata [23] they can be used to feed any model checker to verify if the current model of system satisfies their behavior. We employed the LTSA [25] model checker, but any other could have been used as well.

The case of study consists of a system monitoring the servers' room which belongs to the Software Engineering career at the Universidad Nacional de Avellaneda. Temperature and humidity values from the room are read by sensors (three for temperature and three for humidity), and a central system receives this information. In concrete, the server room contains two servers and six sensors monitor them: sensors *BMP280* and *DS18B20* to measure temperature and three sensors *DHT22* for humidity. A *Wemos* board is in charge of reading the values from the sensors whereas the system features a *ThingSpeak* platform. Communication along the system is regulated by the MQTT (Message Queuing Telemetry Transport) protocol [30].

In further versions of the system more sensors are expected to be aggregated and also an internal system to regulate the temperature of the room taking into account the received information.

We employed two of the architectural patterns from Section 3 in the monitoring system: the Hardware Abstraction Layer pattern (Section 3.3) and the publish/subscribe pattern (Section 3.1). The first pattern was used to shape the communication with the sensors while the second one to distribute the information. Figure 9 shows the architecture of the serve's room monitoring example.

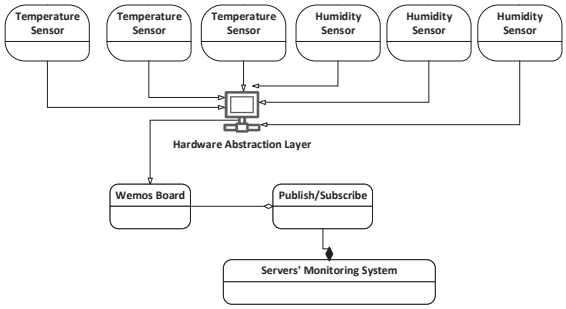

Figure 9. Servers' Monitoring System's Architecture

We build a model of the system and verified the rules modeling both patterns in the LTSA [25] model checker. Regarding performance, LTSA took eight seconds to verify the architectural patterns. After running the model checker we found out that the publish/subscribe pattern behavior was satisfied by the model. However, we found an important bug regarding the Hardware Abstraction Layer pattern, since there was a direct connection from the system to the humidity sensors. This was solved by removing that connection and introducer a proper communication through the gateway.

5. Related and Future Work

We share some objectives and goals with different approaches. Charmy [7] is an appealing framework for architecture's validation and verification. Behavior is denoted using another graphical language called Property Sequence Chart (PSC) [15]. PSC is inspired in UML 2.0 Interaction Sequence diagrams. Contrary to FVS which is exclusively a graphical language, some restrictions in the behavior must be accompanied by natural language. Another distinction is that properties in PSC are described as "negative" or "oppsosite" behavior whereas in FVS behavior is described by using rules.

In [8] a technique to verify some architectural patterns is presented. Specification of software connectors, components, ports and interactions can be thoroughly described by employing a domain specific language. Verification is achieved by introducing a formal and automatic translation to a theorem prover tool. All the specifications follow an operational approach while our technique is based on a declarative perspective.

In [9-10] some modern patterns for embedded and cloud computing patterns are introduced. This kind of work presenting modern challenges for software architectural patterns is clearly an inspiration for others following this line of research. However, in most cases patterns are not formally described. Besides the specification of the pattern, our approach also offers the possibility to formally verify them by employing a model checker tool [25].

Some other approaches presents novel ADL's to shape the behavior of software architectural patterns [19,26-27]. We believe the graphical nature of FVS, the flexibility and expressive power of its notations plus the possibility of introducing verification tasks and synthesis of behavior makes FVS a distinguishable approach.

Regarding future work we would like to undertake an empiric study to compare the flexibility and expressive power of FVS against other ADL's. We would also like to compare performance issues against other approaches verifying architectural specifications like [7-8]. Finally, we also would like to combine FVS with other frameworks, including static architectural analyzers like [28] as well as dynamic analyzers such as [29].

6. Conclusions

In this work we propose FVS as an architectural description language to denote, explore and verify architectural behavior. To validate our proposal we explore the topic of architectural patterns which subsume typical solutions including the interaction of complex components and connectors. In our experimentation we include widely known patterns such as publish/subscribe or blackboard as well other modern patterns in new domains such as embedded software and distributed and cloud computing architectures aiming to cope with new challenges. All of these patterns were modeled and specified in FVS, showing the flexibility and expressive power of our notation. In addition, we were able to formally verify some pattern in a concrete case of study employing a model checker tool.

7. Referencias

[1] Shaw, Mary, and David Garlan. Software architecture. Vol. 101. Englewood Cliffs: prentice Hall, 1996.

[2] Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. ACM SIGSOFT Software engineering notes, 17(4), 40-52.

[3] Medvidovic, N., & Taylor, R. N. (2010, May). Software architecture: foundations, theory, and practice. In 2010 ACM/IEEE 32nd International Conference on Software Engineering (Vol. 2, pp. 471-472). IEEE.

[4] Farshidi, S., Jansen, S., & van der Werf, J. M. (2020). Capturing software architecture knowledge for pattern-driven design. Journal of Systems and Software, 110714.

[5] Bozhukha, L., & Beloborodko, O. (2019). SELECTING THE STRATEGY FOR DESIGNING THE SOFTWARE ARCHITECTURE. System technologies, 6(125), 121-126.

[6] Chondamrongkul, N., Sun, J., & Warren, I. (2019, July). PAT approach to Architecture Behavioural Verification. In SEKE (pp. 187-252).

[7] Pelliccione, P., Inverardi, P., & Muccini, H. (2008). Charmy: A framework for designing and verifying architectural specifications. IEEE Transactions on Software Engineering, 35(3), 325-346.

[8] Marmsoler, D. (2018, April). Hierarchical specification and verification of architectural design patterns. In International Conference on Fundamental Approaches to Software Engineering (pp. 149-168). Springer, Cham.

[9] Fehling, C., Ewald, T., Leymann, F., Pauly, M., Rütschlin, J., & Schumm, D. (2012, June). Capturing cloud computing knowledge and experience in patterns. In 2012 IEEE Fifth international conference on cloud computing (pp. 726-733). IEEE.

[10] Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. SCITEPRESS.

[11] Topaloglu, N. Y., & Capilla, R. (2004, September). Modeling the variability of web services from a pattern point of view. In European Conference on Web Services (pp. 128-138). Springer, Berlin, Heidelberg.

[12] Eloranta, V. P., Hartikainen, V. M., Leppänen, M., Reijonen, V., Haikala, I., Koskimies, K., & Mikkonen, T. (2009). Patterns for distributed embedded control system software architecture. Tampere University of Technology. Report, 2.

[13] Pahl, C., & Barrett, R. (2010). Pattern-based software architecture for service-oriented software systems. e-Informatica Software Engineering Journal.

[14] Gamma, E. (1995). Design patterns: elements of reusable object-oriented software. Pearson Education India.

[15] Autili, M., Inverardi, P., & Pelliccione, P. A scenario based notation for specifying temporal properties. In Proceedings of the SCESM workshop s (pp. 21-28). ACM. (2006).

[16] Cavalcante, E., Oquendo, F., & Batista, T. (2014, August). Architecture-based code generation: from π-ADL architecture descriptions to implementations in the Go language. In European Conference on Software Architecture (pp. 130-145). Springer, Cham.

[17] Cuenot, P., Chen, D., Gérard, S., Lönn, H., Reiser, M. O., Servat, D., ... & Weber, M. (2007). Towards improving dependability of automotive systems by using the EAST-ADL architecture description language. In Architecting

dependable systems IV (pp. 39-65). Springer, Berlin, Heidelberg.

[18] Haider, U., McGregor, J. D., & Bashroush, R. (2019). The ALI Architecture Description Language. ACM SIGSOFT Software Engineering Notes, 43(4), 52-52.

[19] Velte, T., Velte, A., & Elsenpeter, R. (2009). Cloud computing, a practical approach. McGraw-Hill, Inc..

[20] Tanenbaum, A. S., & Van Steen, M. (2007). Distributed systems: principles and paradigms. Prentice-Hall.

[21] Tierney, B., Johnston, W., Lee, J., & Thompson, M. (2000). A data intensive distributed computing architecture for "grid" applications. Future Generation Computer Systems, 16(5), 473-481.

[22] Shaw, M., & Clements, P. (1996). How Should Patterns Influence Architecture Description Languages?. Working paper for DARPA EDCS community.

[23] Fernando Asteasuain and Víctor Braberman. Declaratively building behavior by means of scenario clauses. Requirements Engineering, 22(2):239-274, 2017. doi:10.1007/s00766-015-0242-2

[24] Fernando Asteasuain – Francisco Tarulla. Modelado de Comportamiento de Conectores de Software a través de Lenguajes Declarativos. CONAIISI 2016.

[25] Magee, J., Kramer, J., Chatley, R., Uchitel, S., & Foster, H. (2009). LTSA–Labelled Transition System Analyser.

[26] Zhang, X., Lee, C., & Helal, S. (2019). iPOJO flow: a declarative service workflow architecture for ubiquitous cloud applications. Journal of Ambient Intelligence and Humanized Computing, 10(4), 1483-1494.

[27] Dajsuren, Y. (2019). Defining Architecture Framework for Automotive Systems. In Automotive Systems and Software Engineering (pp. 141-168). Springer, Cham.

[28] Santos, A., Cunha, A., & Macedo, N. (2019, February). Static-time extraction and analysis of the ROS computation graph. In 2019 Third IEEE International Conference on Robotic Computing (IRC) (pp. 62-69). IEEE.

[29] Cavalcante, E., Quilbeuf, J., Traonouez, L. M., Oquendo, F., Batista, T., & Legay, A. (2016, November). Statistical model checking of dynamic software architectures. In European Conference on Software Architecture (pp. 185-200). Springer, Cham.

[30] http://mqtt.org/ MQTT Protocol Specification