



11°
Congreso
TUCUMÁN
2023

20
23



CoNaII SI

XI Congreso Nacional
de Ingeniería en Informática
y Sistemas de Información



XI Congreso Nacional de Ingeniería en Informática y Sistemas de Información.

CoNaISI 2023

Universidad Tecnológica Nacional.
Facultad Regional Tucumán

XI Congreso Nacional de Ingeniería en Informática y Sistemas de Información :
CoNalSI 2023 / Gabriela Ivana Aparicio ... [et al.] ; Compilación de José Augusto
Nasrallah ; Gabriela Ivana Aparicio ; Editado por Fernando Cejas. - 1a ed - Ciudad
Autónoma de Buenos Aires : edUTecNe, 2024.
Libro digital, PDF

Archivo Digital: descarga y online
ISBN 978-987-8992-38-9

1. Ingeniería. 2. Sistemas de Información. 3. Tecnología Informática. I. Aparicio, Gabriela
Ivana, comp. II. Nasrallah, José Augusto, comp. III. Cejas, Fernando, ed.
CDD 629.801

Diseño de Tapa e interior: Fernando Cejas



Universidad Tecnológica Nacional – República Argentina
Rector: Ing. Rubén Soro
Vicerrector: Ing. Haroldo Avetta
Secretaría Cultura y Extensión Universitaria: Ing. Federico Olivo Aneiros



Universidad Tecnológica Nacional – Facultad Regional Tucumán
Decano: Mgter Ing Walter Fabián Soria
Vicedecano: Ing. Juan Esteban Campos
Secretario de Vinculación Tecnológica: Ing. Mario Augusto Madariaga
Secretario de relaciones institucionales: Ing. Jorge Molina



edUTecNe – Editorial de la Universidad Tecnológica Nacional
Coordinador General a cargo: Fernando Cejas
Dirección General: Mg. Claudio Véliz
Dirección de Cultura y Comunicación: Ing. Pablo Lassave

Queda hecho el depósito que marca la Ley N° 11.723

© edUTecNe, 2024

Sarmiento 440, Piso 6 (C1041AAJ)

Buenos Aires, República Argentina

Publicado Argentina – Published in Argentina

ISBN 978-987-8992-38-9



Reservados todos los derechos. No se permite la reproducción total o parcial de esta obra, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio (electrónico, mecánico, fotocopia, grabación u otros) sin autorización previa y por escrito de los titulares del copyright. La infracción de dichos derechos puede constituir un delito contra la propiedad intelectual.

Making Self-Adaptive Systems More Verifiable

Fernando Asteasuain^{1,2}

¹Universidad Nacional de Avellaneda - fasteasuain@undav.edu.ar

²Universidad Abierta Interamericana - Centro de Altos Estudios CAETI

Abstract

Property Adaptation patterns were proposed as a technique to empower the application of formal verification mechanisms to detect and modify behavior in Self-Adaptive Systems. In this work the graphical and declarative language FVS (Feather weight Visual Scenarios) is proposed as a flexible and expressive formalism to specify all the adaption patterns. FVS distinguishable features allow performing the verification process in Self-Adaptive systems in a simple and straightforward manner. This work provides a full specification of the Adaptation patterns in FVS.

Keywords: Formal Verification, Self-Adaptive System.

1. Introduction

Self-Adaptive Systems (SAS) [1-3,10-11] development represents an immense challenge for the Software Engineering community. Not only behavior must be specified, analyzed, implemented and verified but also the system must be able to adapt to different and possible sudden unforeseen changes in the environment in which the system is running. SAS have been successfully applied in several and varied emergent domains such as UAV's (Unmanned Aerial Vehicles), nano satellites, multi-agent systems, fault-tolerant computing, dependable computing, distributed systems, autonomous computing, self managing systems, autonomous communications, adaptable user interfaces, machine learning, economic and financial systems [1-3,8-14].

In order to be able to dynamically adapt to changes in the environment, SAS must be able to detect these changes, establish how to reconfigure goals and objectives, and perform the necessary modifications in the system to satisfy the new set of requirements. For example, consider a mission for a typical UAV to take some high quality photos of landscapes in a given zone. Up to this point, all the UAV's systems, goals and objectives are focused on taking the required photographs [11-12]. Suppose now that an alert is issued stating that a tourist got lost in that zone. If the UAV's is managed as SAS then it could be reconfigured and change its goals and

objectives to try to detect the presence of the missing person. It is crucial to perform these changes dynamically while the UAV is flying, rather than sent it back to the base to recompile its software. Other example is presented in [1-3] in the health care domain, where a Body Sensor Network (BSN) system captures vital signs of patients for detecting emergencies. This system must be able detect any subtle modification in the environment (the patient health's status) and introduce the necessary updates such as incorporating a new oxygen sensor or any other health care strategy.

From the architectural point of view SAS employ a layered and loop-based cycle where changes are detected, modifications applied and a new configuration is launched [1-3,11-12].

In order to detect changes in the environment some approaches have leveraged on the use of formal verification techniques such as model checkers [1-3,11-12]. Model checking [14] is a formal technique which automatically can establish whereas a given model of the system (typically given as an automaton) satisfies a given behavioral property (typically expressed in temporal logics or other similar formalisms). For example, a model checker can be used to detect a possible deadlock in the specification of a distributed system or to verify that very request is answered in a client-server software implementation. This is usually achieved by composing an automaton modeling the desired property with the automaton of the system and checking if an error state is reachable. Other interesting formal approaches aim to detect changes through the use of inductive logics [15]. In this case, inductive logics help the user to find errors in the specification by automatically finding what is missing in the system (in terms of logical variables) to satisfy its goals.

If goals, objectives and requirements are formally specified verification techniques can be applied to detect changes in SAS with verve [1-3,13-14]. In [1-3] a novel technique is presented to ease the way to specify and detect changes in SAS system through the use of model checking. This technique presents the so called **Property Adaptation** patterns, which can be seen as an extension of the Property Specification pattern [8]. The introduction of the Property Specification pattern was a huge milestone in formal verification since it allows expressing behavioral properties in a simple way. One of the most known patterns is the **Response** pattern, which targets a very

common behavior: the occurrence of a stimuli event S must be followed by the occurrence of a response event R . Work by Dwyer, Avrunin and Corbett [8] details a proper translation of each pattern into several formalisms such as temporal logics or graphical formal languages. In this way, the person in charge of the specification can rely on the usage of the patterns to circumvent the details of a complex formal language. Following this initial work on patterns, the Adaptation patterns aim to simplify the phase of modifying a certain property, an activity which is more than crucial in SAS. Adaptation patterns address typical modifications such as removing or adding events. In this way, not only the specification is more amenable but also the application of the required modifications to adapt to the new systems conditions.

The utilization of graphical languages has also been proposed as a way to ease the behavioral specification processes [16-20]. In particular, the so called Feather Weight Visual Scenarios (FVS) graphical language was found to be more flexible and expressive than other formalisms such as temporal logics or other automata-based notations [16-17] when specifying specification patterns. **Taking this into consideration in this work we explore FVS as the specification language to denote all the Property Adaptation patterns.** Results show that the FVS language is able not only to denote all of them but also exhibits some advantages due to its flexibility and expressive power. It is worth mentioning that a proper empiric evaluation assessing the proposal's applicability to property modeling and verification is an objective to be addressed in the near future.

The rest of this work is structured as follows. Section 2 briefly introduces the FVS language, highlighting its most representative features. Section 3 details the Property Adaption patterns specification in FVS. Section 4 and 5 discuss related and future work whereas Section 6 presents some final conclusions.

2. Feather Weight Visual Scenarios

In this section we will informally describe the standing features of FVS [17]. The reader is referred to [17] 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 $A \wedge B$. 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.

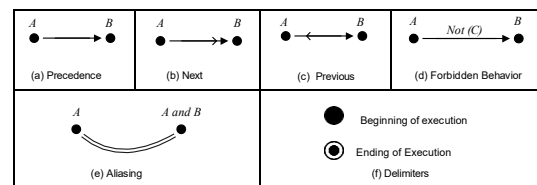


Figure 1. Basic Elements in FVS

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. An example is shown in Figure 2 modeling the behavior of a known software architectural pattern [9]. 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 the application, without needing to know the details of the hardware devices. The solution provided by this 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 2) is very simple: every hardware request must be handled by the interface before it is processed.

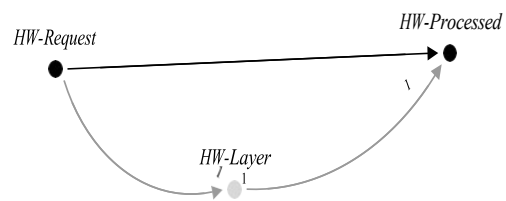


Figure 2. FVS rule example

3. Property Adaptation Patterns in FVS

In this section we show how all the adaptations patterns are modeled in FVS. Sub sections 3.1 to 3.5 address each of the five adaptations patterns introduced in [1-3]. Finally, sub section 3.6 presents some final remarks regarding the FVS specification for the adaptation patterns.

3.1 Updating a Time Guard Adaptation Pattern

This pattern addresses a very common modification that arises when specifying requirements: some conditions changes and “guards” must be updated and modified. Usually, a “guard” represents a Boolean expression indicating that certain actions must be taken in case that the guard is evaluated to true. A common example is updating a time guard: the amount of seconds to trigger a timeout alarm was previously five seconds and now must be triggered earlier, with three seconds without obtaining a response. Given this, to modify a guard and adapt the system to the new behavior this pattern simply proposes to modify the conditions that trigger some actions. Figure 3 sketches a possible instantiation of this pattern in a graph-based notation similar to the one introduced in [1]. Figure 3 show two automata: the one in the top stands for the original version whereas the automaton in the bottom of the Figure shows the adapted version. In this case, to leave the state S1 and enter state S2 the guard is updated from five seconds to three seconds, where S1 stands for the “normal” or regular state and S2 can be considered the state where a timeout is triggered.

In systems where real time is not present in the specification one possible solution is to discretize time [4-7]. For example, introducing events such as “OneSecondDelay”, “TwoSecondDelay” or simply a “Tick” event standing for the classical advance of a clock variable. In this work we followed this classical approach to model time.

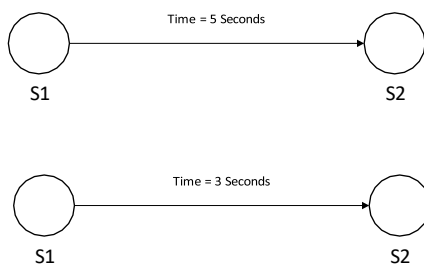


Figure 3. Update Guard Pattern Example.

We now exhibit an FVS specification for the Update a time guard Adaptation Pattern. The two rules specified in Figure 4 target the equivalent behavior to that shown in Figure 3: the three seconds delay replaces the five seconds delay to trigger a timeout. Once again, the rule at the top of the Figure 4 represent the original behavior whereas the rule at the bottom of Figure 4 stands for the modified behavior. We employed some syntactic sugar to simplify

the specification. In particular, we denote the “RegularState” event to name any other state excepting the one in which a timeout is triggered.

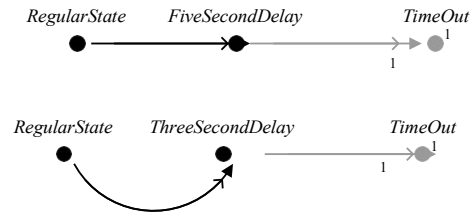


Figure 4. Update Guard Pattern Example in FVS.

As it can be seen, we simply remove the *FiveSecondDelay* event to introduce the *ThreeSecondDelay* event. In other cases, the guard condition to be removed might involve more complex (involving more events for example) which can be a menace to obtain a proper requirements modification. Suppose that the condition to update the triggering of an timeout now involves analyzing the connection status. For example, the new condition to trigger an timeout is expressed as: a three second delay and a *No Connection* status. Since logical events representing simultaneous occurrences of an event can be labeled at any point in FVS these kinds of updates represent no threat at all. Figure 5 shows the new condition including the *No Connection* status.



Figure 5. A more complex guard modification in FVS.

3.2 Updating an Event Adaptation Pattern

This pattern can be seen as a special case for the Updating a Time Guard pattern, where the time guard consists of a occurrence (or not occurrence) of a single event. In simpler words, this adaptation pattern changes one event for another one. The example shown in Figure 4 is a possible example, where the *FiveSeconds* event is replaced in the requirement for the *ThreeSeconds* event. Just to illustrate a more complex example suppose that due to changes in the environment circumventing the system instead of triggering a time out in a regular state, an alarm must be raised when a timeout occurs while the system is in working on a degraded mode. In this case we employed the Update an Event pattern twice: to modify regular state for degraded state, and a timeout for a raise alarm event (See Figure 5).



Figure 5. Update an Event Pattern Example in FVS.

3.3 Adding a response to a response chain Adaptation Pattern

This pattern deals with a typical generalization of the response pattern [8]. In the canonical response pattern only two events are involved: a stimuli S and a response R . The intent of the pattern in natural language can be depicted as: every occurrence of a stimuli S must be followed by a response R . This pattern is known to be one of the most applied pattern in any behavioral specification for a given system [8]. The response chain pattern deals with those cases that more than two events are present: for example, when the response consists of an occurrence of two or more events, or when a sequence of events take part of the stimuli instead of a single event. This adaptation pattern simply handles adding one or more occurrences of events into the chain. In the example shown in Figure 6 a new response $R3$ is added to the previous response chain with $R1$ and $R2$ events.

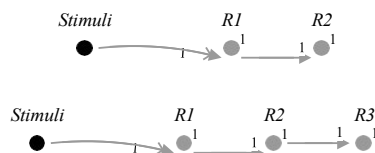


Figure 6. Adding a Response Event Pattern specified in FVS.

Now we illustrate a further example of this pattern just to pinpoint FVS flexibility. Suppose that besides adding a new response $R3$, the new requirement states that the responses in the chain ($R1$, $R2$ and $R3$) can occur in any order. That is, given the stimuli all of them must occur, but in any order and not just $R1$ followed by $R2$ and $R3$. For example, the sequence $R3$, $R1$ and $R2$ is a valid behavior. Figure 7 shows an FVS rule to capture this modification.

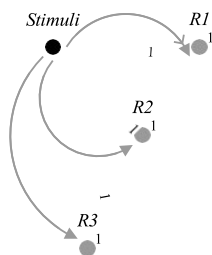


Figure 7. Flexibility in the Response Event Pattern

3.4 Removing a response to a response chain Adaptation Pattern

This pattern is the dual behavior of the previous pattern. This adaptation pattern handles removing one or more events from a sequence of events representing a compound stimuli or response. For example, in the FVS rule in figure 8 the response event $R2$ is removed from the response chain. The new behavior is the following: given the occurrence of a stimuli S two responses must occur in any order: $R1$ and $R3$.

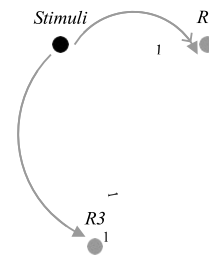


Figure 8. Removing a Response Event Pattern in FVS.

3.5 Splitting the response Chain Adaptation Pattern

This pattern can be seen as a special case for the previous pattern. It simply addresses the case where no special order between the responses is specified, so any of them can occur in any order. As it was previously exemplified (see Figure 7) in FVS this pattern can be straightforward modeled by simply relaxing the precedence relationship between the events composing the response (or the stimuli).

3.6 Analyzing FVS specification for the Property Adaptation Patterns

FVS flexible and expressive notation is capable of denoting all the adaptation patterns in a smooth and straightforward way. For example, the splitting a response pattern can be simply modeled as a variant of the Adding/removing a Response chain pattern. In addition, more complex situations denoting two or more events can be naturally modeled since we can denote logical formulas to abstract the occurrence of events. As it was shown in Section 3.1 a more complex time guard condition was seamlessly specified (see Figure 5).

Although an empiric evaluation is needed to formally assess FVS's usability and applicability to model property adaptation patterns we can state that the presented results are promising and encouraging to continue this line of research.

4. Related Work

Property Adaption patterns are introduced in [1-3]. Behavioral properties are expressed employing future MTL[21] and the UPPALL model checker [22] is used as the verification tool. Properties are translated into automata using the Property Specification patterns and the Property Adaption patterns and the composed with the automata representing the systems' behavior. Then, the UPPALL tool checks if the system satisfies the property or not. If not, the systems requirements' are updated to comply with the self-adaption philosophy. The software architecture relies on a MAPE_K feedback loop style [23]. There is human intervention in this self-adaptive loop, since it is a human who, taking notice of the adaptation patterns, make the necessary transformation on the behavioral properties to reflect the new conditions for the

system. In this context, a more flexible and graphical notation such as FVS can be more naturally employed. In [16] FVS was shown as a more easy formalism to adapt reason and modify specification patterns than other formalisms like temporal logics or automata-based notations. It is necessary an empiric validation over the adaptation patterns to conclude that is also the case for them. This is indeed an interesting future line of research to explore.

Other remarkable approaches relying over self-adaptive systems are [24-25]. D'Ippolito et al. [24] presents a multi-tier layered architecture framework where behavioral properties are built using control synthesis. According to Carwehl et.al [1], these approaches may not be flexible enough to support adaptation of individual properties.

Other similar approaches also employ automata-based properties and model checkers to update properties and system's goals [26-27]. In these cases, properties are expressed using structured natural languages or formalisms like the ones shown in the Property Specification patterns [8]. On the contrary, FVS rely on a declarative flavor of behavioral specification.

Finally, it is interesting to mention probabilistic and quantitative approaches [28-30]. It would be attractive to explore if FVS could be extended to denote probabilistic model checking.

5. Future Work

We certainly would like to make a throughout empiric validation of FVS as a formalism to specify adaptation patterns. A possible step in this direction would be to compare the size of the automata involved and performance of the model checking verification against other approaches like [1-3].

We also would like to apply FVS specification of the adaptation patterns in a complex enough case of study. One possibility is to analyze the Body Sensor network (SBN) system introduced in [1-3]. This system can be used to monitor a patient's vital signs to determine if the patient is in a emergency state. Another example to be explored could be satellite and robotic missions as the one shown in [20].

We can mention two additional research lines to continue this work. For one side, we are working on a theoretical proof to exhibit that FVS's specification of the adaptation patterns is sound and complete. On the other side, we are interested in applying FVS in the behavioral synthesis for self-adaptive systems domain [24].

6. Conclusions

In this work we present the specification of all the Property Adaptation patterns in the graphical and

declarative framework FVS. Adaptation patterns were proposed as a novel technique to simplify the application of formal verification when modifying behavior in Self-Adaptive Systems.

The FVS specification of the entire adaptation patterns shows that the flexibility and expressive power of this formalism come together with neat benefits such as easing the process of exploring, analyzing and modifying a given property. Further continuation of this line of work could greatly improve the verification process for Self-Adaptive Systems.

References

- [1] Carwehl, M., Vogel, T., Rodrigues, G. N., & Grunske, L. (2023). Runtime Verification of Self-Adaptive Systems with Changing Requirements. arXiv preprint arXiv:2303.16530.
- [2] Vogel, T., Carwehl, M., Rodrigues, G. N., & Grunske, L. (2023). A property specification pattern catalog for real-time system verification with UPPAAL. *Information and Software Technology*, 154, 107100.
- [3] Rodrigues, A., Caldas, R. D., Rodrigues, G. N., Vogel, T., & Pelliccione, P. (2018, May). A learning approach to enhance assurances for real-time self-adaptive systems. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems* (pp. 206-216).
- [4] Jacod, J., & Protter, P. (2011). *Discretization of processes* (Vol. 67). Springer Science & Business Media.
- [5] Asarin, E., Maler, O., & Pnueli, A. (1998). On discretization of delays in timed automata and digital circuits. In *CONCUR'98 Concurrency Theory: 9th International Conference Nice, France, September 8–11, 1998 Proceedings* 9 (pp. 470-484). Springer Berlin Heidelberg.
- [6] Bouyer, P., Fahrenberg, U., Larsen, K. G., Markey, N., Ouaknine, J., & Worrell, J. (2018). Model checking real-time systems. *Handbook of model checking*, 1001-1046.
- [7] Waez, M. T. B., Dingel, J., & Rudie, K. (2013). A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9, 1-26.
- [8] Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1998, March). Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice* (pp. 7-15).
- [9] 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.
- [10] Zudaire, S. A., Nahabedian, L., & Uchitel, S. (2022). Assured mission adaptation of UAVs. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 16(3-4), 1-27.
- [11] Zudaire, S., Gorostiaga, F., Sánchez, C., Schneider, G., & Uchitel, S. (2021, May). Assumption monitoring using runtime verification for UAV temporal task plan executions. In *2021 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 6824-6830). IEEE.

- [12] Bonnet, J., Gleizes, M. P., Kaddoum, E., Rainjonneau, S., & Flandin, G. (2015, September). Multi-satellite mission planning using a self-adaptive multi-agent system. In 2015 IEEE 9th International Conference on Self-adaptive and Self-organizing Systems (pp. 11-20). IEEE.
- [13] Macías-Escrivá, F. D., Haber, R., Del Toro, R., & Hernandez, V. (2013). Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18), 7267-7279.
- [14] Clarke, E. M., Henzinger, T. A., Veith, H., & Bloem, R. (Eds.). (2018). *Handbook of model checking* (Vol. 10, pp. 978-3). Cham: Springer.
- [15] Alrajeh, D., Kramer, J., Russo, A., & Uchitel, S. (2012). Elaborating requirements using model checking and inductive learning. *IEEE Transactions on Software Engineering*, 39(3), 361-383.
- [16] Asteasuain, F., & Braberman, V. (2017). Declaratively building behavior by means of scenario clauses. *Requirements Engineering*, 22(2), 239-274.
- [17] Asteasuain, F., & Braberman, V. (2015). Specification patterns: formal and easy. *International Journal of Software Engineering and Knowledge Engineering*, 25(04), 669-700.
- [18] Holzmann, G. J. (2002). The logic of bugs. *ACM SIGSOFT Software Engineering Notes*, 27(6), 81-87.
- [19] Autili, M., Inverardi, P., & Pelliccione, P. (2007). Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14, 293-340.
- [20] Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., & Berger, T. (2019). Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 47(10), 2208-2224.
- [21] Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4), 255-299.
- [22] Larsen, K. G., Pettersson, P., & Yi, W. (1997). UPPAAL in a nutshell. *International journal on software tools for technology transfer*, 1, 134-152.
- [23] Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1), 41-50.
- [24] D'Ippolito, N., Braberman, V., Kramer, J., Magee, J., Sykes, D., & Uchitel, S. (2014, May). Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 688-699).
- [25] Borda, A., Pasquale, L., Koutavas, V., & Nuseibeh, B. (2018, May). Compositional verification of self-adaptive cyber-physical systems. In *Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems* (pp. 1-11).
- [26] Perez, I., Mavridou, A., Pressburger, T., Goodloe, A., & Giannakopoulou, D. (2022, March). Automated translation of natural language requirements to runtime monitors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 387-395). Cham: Springer International Publishing.
- [27] Mallozzi, P., Castellano, E., Pelliccione, P., Schneider, G., & Tei, K. (2019, May). A runtime monitoring framework to enforce invariants on reinforcement learning agents exploring complex environments. In 2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE) (pp. 5-12). IEEE.
- [28] Cailliau, A., & Lamsweerde, A. V. (2019). Runtime monitoring and resolution of probabilistic obstacles to system goals. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 14(1), 1-40.
- [29] Calinescu, R., Weyns, D., Gerasimou, S., Ifikhar, M. U., Habli, I., & Kelly, T. (2017). Engineering trustworthy self-adaptive software with dynamic assurance cases. *IEEE Transactions on Software Engineering*, 44(11), 1039-1069.
- [30] Solano, G. F., Caldas, R. D., Rodrigues, G. N., Vogel, T., & Pelliccione, P. (2019, May). Taming uncertainty in the assurance process of self-adaptive systems: a goal-oriented approach. In 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) (pp. 89-99). IEEE.