CRESCO Framework and Checker: Automatic generation of Reflective UML State Machine's C++ Code and Checker

1st Miren Illarramendi
Software and Systems Engineering Research Group
Mondragon Goi Eskola Politeknikoa (MGEP)
Arrasate, Spain
millarramendi@mondragon.edu

2nd Leire Etxeberria
Software and Systems Engineering Research Group
Mondragon Goi Eskola Politeknikoa (MGEP)
Arrasate, Spain
letxeberria@mondragon.edu

3rd Felix Larrinaga
Software and Systems Engineering Research Group
Mondragon Goi Eskola Politeknikoa (MGEP)
Arrasate, Spain
flarrinaga@mondragon.edu

4th Goiuria Sagardui
Software and Systems Engineering Research Group
Mondragon Goi Eskola Politeknikoa (MGEP)
Arrasate, Spain
gsagardui@mondragon.edu

***Abstract*—Software Systems are becoming increasingly complex leading to new Validation & Verification challenges. Model checking and testing techniques are used at development time while runtime verification aims to verify that a system satisfies a given property at runtime. This second technique complements the first one. This paper presents a tool that enables the developers to generate automatically reflective UML State Machine controllers and the Runtime Safety Properties Checker (RSPC) which checks a component-based software system's safety properties defined at design phase. We address embedded systems whose software components are designed by Unified Modelling Language-State Machines (UML-SM) and their internal information can be observed in terms of model elements at runtime. RESCO (REflective State Machines-based observable software COmponents) framework, generates software components that provide this runtime observability. The checker uses software components' internal status information to check system level safety properties. The checker detects when a system safety property is violated and starts a safe adaptation process to prevent the hazardous scenario. Thus, as demonstrated in the evaluated experiment but not shown in the paper due to the space limitation, the safety of the system is enhanced.**

***Index Terms*—Automatic code generation, Runtime monitoring, Robustness, Models@runtime, Software components, UML State-Machines**

I. Introduction

The scope, complexity, and pervasiveness of software systems continue to increase dramatically. The consequences of these systems failing can range from mildly annoying to catastrophic. Software increasingly assumes the responsibility for providing functionality in systems. Therefore, enhancing the safety through software has a significant impact on the system.

Verification and validation techniques applied during development give a certain level of confidence in correctness and are effective at detecting and avoiding anticipated faulty scenarios. However, in embedded software systems where a fault can lead to a critical scenario, we need a way to detect and mitigate hazardous and uncertain scenarios. Runtime verification techniques could be used to maintain safe control in unanticipated circumstances.

Monitoring information related to the internal status of the embedded software can anticipate the detection of failures. This makes it possible to take corrective actions earlier and prevent faulty scenarios. This idea is described as a safety bag in [1] and [2]. The goal is to prevent software systems' hazardous states by means of safety verification at runtime. Thus, we increase their robustness enhancing safety.

Current runtime checking solutions as shown in Figure 1, are specified at different abstraction levels: system, component, class, method or statement. As can be observed, most of the approaches check if the specification is fulfilled at the same level that the monitoring is performed. Thus, for the detection of system level misbehaviour, only system level properties are checked. Nevertheless, component or class level properties can give valuable information in detecting system level problems and undesired emergent behaviour.

Both software and hardware checkers have been studied in [4]. In this work they argue that while hardware checkers are able to detect errors before a change of state (transition) is performed, software checkers detect errors once a change of state has occurred. Hardware checkers, such as the Noninterference Monitoring Architecture checker [5], need additional hardware to collect state information and to assist in checking. As a result, the cost and complexity of the solution is increased.

In addition, most runtime software checkers require modifying the source code of the observed system by instrumented code. However, it is desirable that runtime checkers for testing safety properties of the systems should be isolated from the target system to minimize any interference of the system being

978-1-7281-7735-9/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSREW51248.2020.00032

tested [6].

In this paper, the Runtime Safety Properties Checker (RSPC) is presented. RSPC tries to solve the aforementioned limitations of the current software runtime checkers. The checker is based on the concept of runtime Safety Properties (SP).

Safety Properties assert that the system always stays within some allowed region. Today's rapid development of complex and safety-critical systems requires reliable verification methods. In formal verification, we verify that a system meets a desired property by checking that a mathematical model of the system meets a formal specification that describes the property. The properties asserting that observed behaviour of the system always stays within some allowed set of finite behaviours, in which nothing "bad" happens, have a special interest. For example, we may want to assert that every message received was previously sent. Such properties of systems are called Safety Properties.

Our approach (RSPC) is classified in the target area of Figure 1. Specification is made using system level safety properties and the monitored information is the internal status of the software components of the system. It verifies the system level safety properties based on the internal status information received from each of the systems' software components' observable states at runtime. These properties are checked when the internal status of the system components is going to perform a state transition. In this sense, our software checker shares the advantages of hardware checkers because we can detect the error before a change in the component's state happens. RSPC is developed in an isolated way and therefore without involving any modification in the component to be observed.

The approach has been validated in some academic toy examples and one industrial use case. The latter is a Train Door Controller studied in [7]. The main contributions of RSPC are:

- check system level safety properties by monitoring internal status of the system's software components,
- prevent failures before a change in state of the software components occurs,
- isolate the system's functionality and its own (RSPC) while not interfering with the developer's design and development work.

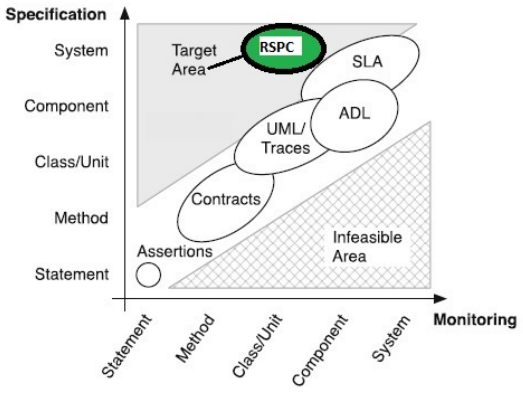

Fig. 1. Abstraction level of specifications vs. runtime monitoring abstraction levels based on [3].

It is worth noting that the RSPC has been developed taking into account a work called RESCO framework [8]. This framework is able to generate reflective software components that provide information about the internal status of the software components in terms of UML-SM elements at runtime. The safety properties that the RSPC checks are defined using the information provided by these software components at runtime. Nonetheless, the RSPC solution can be used independently of RESCO software components. In any case, the software components we are addressing have to fulfill the following conditions:

1) they have to be designed by UML-SMs,
2) they have to provide the internal status of their observed states at runtime,
3) they have to provide the ability to adapt to a safe-mode to avoid any hazardous situation at runtime.

RSPC also requires consistent snapshots of the system and, to this end, for example, the observed system has to be a synchronous one or the system's messages must be causally ordered.

Section II presents the Background to understand the presented work and Section III introduces the RESCO framework and the Runtime Safety Properties Checker. Finally, section IV ends the paper with our Conclusions and Future Lines of action.

II. Background

In this section, we will define some concepts that will help to understand the work presented in this document.

One way to perform runtime verification is to observe the runtime information (traces) sent by the software controller to the externalized runtime checker/adapter. Since correct traces will be finite and predefined in the checker/adapter, when the received trace is not defined as a correct one, the checker/adapter comes to a state that a Trace-Violation has been detected.

One possible solution for this approach is using the information of model elements (current state, event, next state,...) of the UML-SM model of the software component under study. This enables using a common language to design and verify software components at runtime.

In order to maintain the model at runtime, the software component has to be observable by the externalized checking and adapting system. In order to support this characteristic, the software components that are monitored need to have the introspection and reflection ability. Introspection supports runtime monitoring of the program execution with the goal of identifying, locating and analyzing errors [9].

Reflection [10] can be defined as the property by which a component enables observation and control of its own structure and behavior from outside itself. This means that a reflective component provides a meta-model of itself, including

structural and behavioral aspects, which can be handled by an external component. A reflective system is basically structured around a representation of itself or meta-model that is causally connected to the real system.

The software components of the presented solution are generated automatically from the UML-SM model. Even if we apply model checking methods to the models, due to the complexity of the systems there may be residual faults. In this scenario, solutions that support runtime verification are needed.

The faults that can be detected by the solution are:

- random hardware faults such as bit inversions or changing errors,
- random software faults such as heisenbugs [11],
- residual faults not detected when testing,
- unanticipated faults that were not considered in the design and development phase.

Runtime Adaptation is a technique prevalent to long-running, highly available software systems, whereby system characteristics (e.g., its structure, locality ...) are altered dynamically in response to runtime events (e.g., detected hardware faults or software bugs, changes in system loads), while causing limited disruption to the execution of the system [12].

In [13] is proposed an "externalized" runtime adaptation system that is composed of external components that monitors the behaviour of the software component of the running system. These external components are responsible for determining when a software component's behaviour is within the envelope of acceptable system parameters. When the software component's behaviour fall outside of the expected limits, the external components start the adaptation process. To accomplish these tasks, the externalized mechanisms maintain one or more system models, which provide an abstract, global view of the running system, and support reasoning about system problems and repairs.

To accomplish the adaptation process, the externalized modules:

1) maintain the models that provide an abstract, global view of the monitored running software components and the whole system, and
2) support reasoning about system problems and how to repair them.

We define as normal-mode of operation the situations in which all elements of the system are functioning as intended and the software component's behaviour is within the envelope of acceptable system parameters. When the software component's behaviour is not working in the expected limits, the adaptation process starts and the software component is sent to a safe-mode of operation (graceful degradation). The safe-mode operation is an aspect of a fault tolerant software system, where in case of some faults, system functionality is reduced to a smaller set of services/functionalities that can be performed by the system [14].

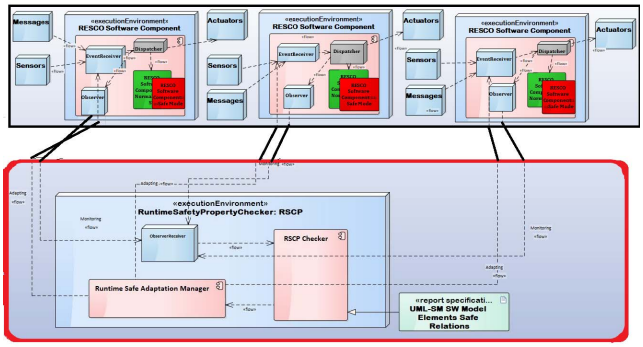

Fig. 2. General Architecture of the Safe Properties Checking System

III. Reflective Code Generator and Runtime Checker

Our approach has the following characteristics: (1) specification of what we want to be observed at runtime is added to the models at design phase; (2) code generation takes this information and generates model-centric code; (3) observed information not only considers the outputs of the software components but also monitors their states and events status;(4) systems composed of CRESCO software components have the ability to detect unsafe scenarios if a component's behaviour deviates from the established one at specific points of time. In the latter case, the runtime checker sends an event to the observed software component and this component changes its operation mode automatically to safe-mode.

In Figure 2, the overall architecture of the solution is shown. The solution uses an externalized checker, the RSCP, which is composed of different modules. The main modules are the *RSCP Checker* and the *Runtime Safe Adaptation Manager*. The former, checks the fulfillment of the system level safety properties based on software component level information in model elements. This information is sent by the software components of the system at runtime and when it detects that one of the safety properties is not fulfilled the *Runtime Safe Adaptation Manager* module starts the adaptation process.

This section will present the process of generating the reflective software components by the RESCO framework (Section III-A) and the RSPC (Section III-B), how the runtime state-based safety properties are specified (Section III-C) and how the safe adaptation process is defined (Section III-D).

A. Generating automatically software components using RESCO

This section presents the detailed steps to follow in the generation of the software components using RESCO framework. The overall process is shown by an SPEM process in figure 3.

1) 1st step: Behaviour design of the software component: The first step is to model the behaviour of software components by UML-SM models using Papyrus [15] modeling tool. In the presented solution, runtime adaptation is one of the

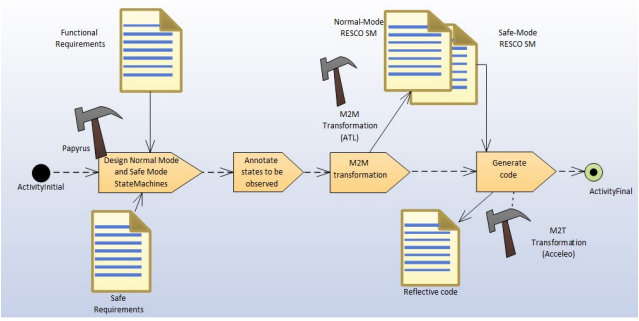

Fig. 3. SPEM diagram of the RESCO methodology

contributions and for that, in this first step the safety engineer has to design also the alternative safe-mode behaviour(s) of the software component. All these models will be transformed by the RESCO M2M transformation rules. In addition, the safety engineer has to define for each unsafe detected modes the initial state in the safe-mode model to be adapted.

2) 2nd step: Automatic generation of the RESCO Model: Once the designer finishes the first step, RESCO framework continues with the work. First, it takes the annotated UML-SMs and performs a M2M transformation. As a result, it generates an instrumented model for each of the designed UML-SM. For doing this, we defined some platform-independent transformation rules.

Our approach is based on a platform-independent model instrumentation process. As in [16], our approach uses M2M transformation techniques for creating an instrumented version of the user-defined model supporting introspection, checking and adapting activities at runtime. Thus, without having to instrument the code, we are able to generate applications providing advanced capabilities such as component introspection by themselves.

To formalize our approach, we considered only the computations that occur in actions and conditions attached to transitions.

Figure 4 shows how a transition chain between two states is instrumented in order to provide debugging and observation ability at runtime. The left side of the figure shows what happens when, being the software component in *S1*, *EvA* arrives. The right side shows the equivalent version of the transition after model instrumentation. The new model introduces a choice point and a composite state that will get the observed information and share/log it. Certain solutions to instrument the models follow the approach described in Figure 4. In our case, we follow this approach but the composite state is shared by all the transitions. We do not have to implement different Observer States for each of the transitions, we need not to add explicitly the instrumented model in each use case's transitions. Once a state is annotated as observed state, this behaviour is added by construction and shared in all the observed transitions.

Summarizing, this is the overall behaviour of RESCO-SMs: when an event is sent to the state machine based software

TABLE I
RESCO OBSERVED DATA

Data	Description
Component Name	Identification of the current component
Current State	Identification of the current state
Next State	Identification of the next/target state
Father State	Identification of the father state
Event Id	Identification of the current event

component, the dispatcher analyzes the current status and calculates if a transition has to be performed. If the transition is going to be performed, and the *current*, *next* or *parent* state is annotated as *Observable*, the current state information is observed and sent to the externalized checker. Having this observed information at runtime, we can localize bugs analyzing execution traces in model terms.

3) 3rd step: C++ reflective UML-SM based software components generation (CRESCO): In this section we will present the RESCO approach for C++: CRESCO framework. As we have mentioned, the RESCO metamodel is platform independent.

In order to generate an application with *Observable* software components in terms of model elements at runtime, CRESCO framework includes: (1) M2T transformations of the elements of the design package part of the RESCO metamodel into C++ code by the Acceleo [17] tool, and (2) an implementation in C++ of the runtime infrastructure.

Following this solution, table I provides the information available from the RESCO software components at runtime. The runtime checker will receive this information at runtime from the states annotated as *observable* in order to check the correctness of the system.

B. Process for defining Safe Properties and generating the RSPC

Before showing the process and in order to be more clear in the next explanations, we are going to define some terms:

- Safety Requirement (SR) will be allocated to the system and it may be satisfied by a safe property or a set of safe properties.
- A State-Based Safe Property (SP) is a specification of correct compound state of the system. System level safe properties are defined based on the possible states in

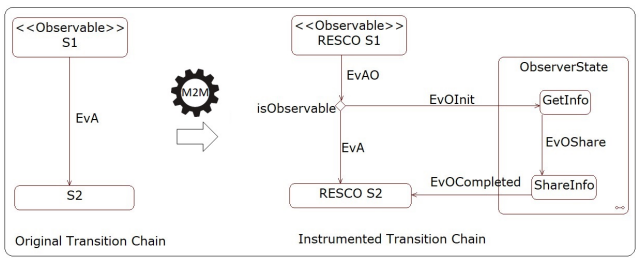

Fig. 4. Model Instrumentation: Transformation Rule of the runtime package of RESCO metamodel.

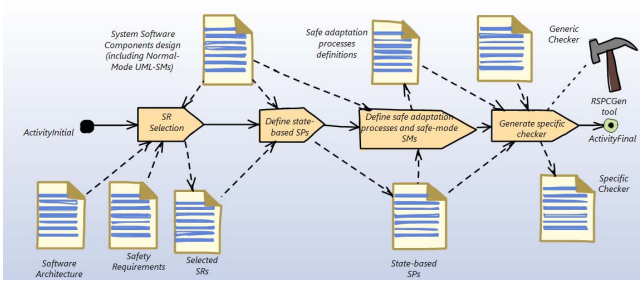

Fig. 5. Process for generating state-based safe properties checker

which each of the software components of the system can coexist at runtime.

The process to generate the RSPC is embedded in a typical design process for developing dependable systems. After designing the system and obtaining the system architecture with the decomposition of software components, together with a first design of the software components including their behaviour (UML-SM diagrams), the process for defining state-based safety properties starts. This process has four steps (see Figure 5):

1) Step1: Select Safety Requirement to be used at runtime verification. Not all safe requirements are verifiable in terms of internal states of system's components; those that can be verified in this way have to be selected. The result is a list of safe requirements (SR_i).
2) Step2: Define state-based Safe Property based on selected SR. The information of the States involved in the SPs could be used to automatically annotated the states that must be observed.
3) Step3: Define $safeAdapt$ processes to be launched in case safe properties are not fulfilled at runtime (a process for each safe property).
4) Step4: Generate the checker: Runtime Safe Properties Checker Generator (RSPCGen) tool transforms the safe properties to RSPC Code (checker, in C++) automatically. RSPCGen uses a generic checker as a basis and adds the specific state-based safe properties to the RSPC Checker module and safe adaptation processes to the Runtime Safe Adaptation Manager module.

It can be argued that the RSPC is a generic solution. It is true that it has a use case specific part in which the safety properties and the adaptation process have to be defined. Nevertheless, using the templates and formalism to define those specific parts, we may say that the process itself is generic and the definition of the specific parts have been also become as generic as possible.

a) RSPC behaviour at runtime: The RSPC starts after getting an initial consistent snapshot of the component-based software system. Next, the checker waits until it receives an update from any of the system's software components. The observer of these software components sends their internal status information before performing a state transition. The RSPC checker compares this information with the system level safe properties. If system safe properties are fulfilled, the system status information is updated and the RSPC waits for new updates.

In the event of the safe properties not being fulfilled, the RSPC Checker module notifies that circumstance to the Runtime Safe Adaptation Manager module. Next, this manager starts the predefined safe adaptation process. This process sends predefined events to the software components that are involved in the safe adaptation process. Finally, these software components are adapted to the safe-mode UML-SMs and the system continues in this mode until its reparation.

C. Specification using Safe Properties

We need to prove that the composite implementation of the system guarantees system level properties at runtime.

In our case, a safe property specifies properties related to the internal behaviour of the software components that are part of the system in terms of their UML-SM model (active states). That is what we call a state-based safe property.

In our approach a system (Sys) may be composed of subsystems (that could be further decomposed) and primitive components (C) that cannot be further decomposed. Furthermore, the primitive components have a behaviour specified using a state machine. A system (Sys) is composed of at least one primitive Component (C), i.e., $Sys = \{C_1, C_2, ...C_{nc}\}$ where nc is the total number of primitive components (Cs) in the system (Sys). Accordingly, a C in our context is state-based. Each of these Components has a set of states (S), i.e., $C_i = \{S_1, S_2, ...Sns_{C_i}\}$ where ns_{C_i} is the number of states that comprise the i-th C.

Let us denote the active state of a component (C) at a discrete time point as $C_i.S_j$, the state S_j being any of the states the C component ($S_j \in C_i$) may have.

Safety requirement will be allocated to the system. And a safety requirement may be satisfied by a safe property or a set of safe properties. A safe property will be related to the states of the components involved in the property. Next grammar is used to specify what to check. The relevant syntax of this grammar has been summarized:

$$
\begin{array}{rcl}
\textit{safe property} & := & \textit{constraint} \mid \textit{timedConstraint}; \\
\textit{constraint} & := & \textit{condition} \ \textbf{implies} \ \textit{condition}; \\
\textit{condition} & := & \textit{activeState} \mid \textbf{not} \ \textit{condition} \mid (\textit{condition}) \mid \\
 & & \textit{condition} \ \textbf{or} \ \textit{condition} \mid \\
 & & \textit{condition} \ \textbf{and} \ \textit{condition}; \\
\textit{timedConstraint} & := & \textit{constraint} \ \textbf{in} \ \textit{timeUnits}; \\
\textit{activeState} & := & ComponentName.StateName;
\end{array}
$$

D. Safe Adaptation Process definition

When the RSPC Checker module detects that one of the safe properties is not being fulfilled, it sends this information to the safeAdaptationProcess Manager module. This manager has a table that was created in Step 3 of the RSCP generation process. The information of the table is organized as presented in Table II.

Based on the information that is available in this table, the specific safe adaptation process for the not fulfilled SP will start.

IV. Conclusion and Future Work

This paper presents a reflective UML State Machines controllers' automatically C++ code generator and a runtime checker for these controllers. This checker considers system level specific safety properties to be verified at runtime.

The tooling generated and presented in this document have been empirically evaluated and the performance of the checker (in terms of execution time and percentage of CPU usage) using a different number of system level safe properties have been also checked. We also evaluated if the checker is an effective tool for failsafe operation at runtime. Some experiments were carried out by injecting random and unconditional faults into the software components. This information was not included in this work because of the page number limitation.

The main conclusions are that the checker is able to detect all errors that impact on the safe properties at runtime, thereby ensuring the safe behaviour of the system. As it uses components' internal information, it has the ability to prevent faulty scenarios before having changed the system output signals. This is a benefit compared with software monitors that can only check the output signals.

Our last conclusion is that the process to implement and generate reflective UML State Machine's based software controllers and the RSPC is cost-effective as the system level RSPC and the controllers themselves are generated automatically. The safety engineer simply has to define the safe properties (following the defined grammar to this end) and the safe adaptation processes. The rest of the process is automatic. The software developer of the software components only considers the functional aspects of the system.

Having evaluated the experiments and results, the benefits and novelty of the RSPC could be summarized as follows:

- RSPC is a generic solution able to be used in different use cases. The specific part of each use case is added by using templates and formalism defined to this end.
- RSPC uses software component level internal information in model element terms to check the correctness at system level. The solution follows the models@runtime approach and fits in the target area shown in Figure 1.
- The solution is based on a runtime adaptation approach. Not only does it detect the errors or unexpected circumstances faster than other existent approaches, but it also follows a runtime enforcement strategy, thereby avoiding transitions on system's software components to hazardous states.

As future research lines, we might consider expanding the empirical evaluation by using other realistic use cases in different industrial domains and projects.

TABLE II
Safe Adaptation Process Information

Safe Property	Id of the not fulfilled SP
Involved SW component	Id of the SW component(s) to be updated
Safe Mode State Machines	Id of the SM(s) to be updated (safe mode UML-SM Id)
Initial State	Id of the initial state of the safe UML-SM

Acknowledgment

The project has been developed by the Software and Systems Engineering Research Group of MGEP and supported by the Department of Education, Universities and Research of the Basque Government under the projects Ikerketa Taldeak (IT1326-19) and by the European H2020 research and innovation programme, ECSEL Joint Undertaking, and National Funding Authorities under the project Productive 4.0 with grant agreement no. GAP-737459 - 999978918.

References

[1] IEC, *61508:Functional safety of electrical/electronic/programmable electronic safety related systems*, 2010.

[2] M. Brini, P. Crubill, B. Lussier, and W. Schn, "Complementary methods for designing safety necessities for a safety-bag component in experimental autonomous vehicles," in *Proceedings 12th National Conference on Software and Hardware Architectures for Robots Control*, 2017.

[3] M. Pezzé and J. Wuttke, "Model-driven generation of runtime checks for system properties," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 1, pp. 1–19, Feb. 2016. [Online]. Available: http://dx.doi.org/10.1007/s10009-014-0325-2

[4] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, Dec. 2004.

[5] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi, "A noninterference monitoring and replay mechanism for real-time software testing and debugging," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 897–916, 1990.

[6] A. Kane, T. Fuhrman, and P. Koopman, "Monitor based oracles for cyber-physical system testing: Practical experience report," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 148–155.

[7] E. Gomez-Martinez, R. J. Rodrguez, C. B. Earle, L. Elorza, and M. Illarramendi, "A methodology for model-based verification of safety contracts and performance requirements," *Journal of Risk And Reliability*, 2016.

[8] M. Illarramendi, L. Etxeberria, X. Elkorobarrutia, and G. Sagardui, "Runtime observable and adaptable uml state machines: models@ run. time approach," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2019, pp. 1818–1827.

[9] M. James, P. Springer, and H. Zima, "Adaptive fault tolerance for many-core based space-borne computing," in *European Conference on Parallel Processing*. Springer, 2010, pp. 260–274.

[10] P. Maes, "Concepts and experiments in computational reflection," in *ACM Sigplan Notices*, vol. 22, no. 12. ACM, 1987, pp. 147–155.

[11] M. Grottke and K. S. Trivedi, "A classification of software faults," *Journal of Reliability Engineering Association of Japan*, vol. 27, no. 7, pp. 425–438, 2005.

[12] I. Cassar and A. Francalanza, "Runtime adaptation for actor systems," in *Runtime Verification*. Springer, 2015, pp. 38–54.

[13] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *Proceedings of the first workshop on Self-healing systems*. ACM, 2002, pp. 27–32.

[14] R. Dhall, "Designing graceful degradation in software systems," in *Proceedings of the Second International Conference on Research in Intelligent and Computing in Engineering*, vol. 10, 2017, pp. 171–179.

[15] Papyrus. (2019) Papyrus. [Online]. Available: https://eclipse.org/papyrus/

[16] M. Bagherzadeh, N. Hili, and J. Dingel, "Model-level, platform-independent debugging in the context of the model-driven development of real-time systems," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 419–430.

[17] A, "Acceleo," https://www.eclipse.org/acceleo/, Tech. Rep., 2016.