MONDRAGON UNIBERTSITATEA

GOI ESKOLA POLITEKNIKOA

FACULTY OF ENGINEERING

PHD THESIS

Runtime Observable and Adaptable UML State Machine-Based Software Components Generation and Verification: Models@Run.Time Approach

Author:

MIREN ILLARRAMENDI REZABAL

Supervisors:

Dr. XABIER ELKOROBARRUTIA LETONA

Dr. LEIRE ETXEBERRIA ELORZA

Computer and Electronics Department

Faculty of Engineering

Mondragon Unibertsitatea

Arrasate

September 2019

“I think it’s very important to get more women into computing.
My slogan is: Computing is too important to be left to men”
Karen Sparck Jones

Ama eta Aita.
Haritz, Txomin eta Amaia.
Nere bi anayak eta familiya dana.

Indarra ateratzeko adorea eman didaten guztiak.
Bereziki, niretzat bide ardatz izan diren emakumeak.
Zientzian ERE geure lekua daukagulako.

Eskerrak bihotz-bihotzez, inoiz bueltatu ezingo dizuedan guztiagatik.
Miren

Declaration

Hereby I declare that this document is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Miren Illarramendi Rezabal
Arrasate, September 2019

Acknowledgments

Kostata, baina azkenean bai, dokumentu hau idazteraino iristsi naiz. Oraindik gogoratzen dut, honetan murgiltzeko erabakia bera ere ez zela erraza izan. Bi umeren ama, gaztaroaren bizitasuna galtzen hasita,... eta orain honetan sartu behar dezu, Miren? Baina bai, zergatik ez! Eta aurrera pausoak poliki poliki ematen hastea erabaki nuen. Ez dut gezurrik esango, ez da erraza izan, baina behin tematurik eta benetan gustatzen zaizun horretan sartuta, aurrera egitea beste aukerarik ez dut izan. Ta oso gustora. Ez naiz ezertaz damutzen.

Baina guzti hau horrela izan dadin, bidean lagundu didaten lagun asko izan ditut. Lehenengo eta behin, bide osoa nirekin egin duten nire bi zuzendariak: Leire eta Elkoro. Beti sentitu izan dut zuen konfidantza eta asko lagundu didazue bidean. Aurrera egin behar nuen bakoitzean, zuen ekarpenak ezinbestekoak izan dira. Beti modu eraikitzailean egin izan dizkidatzue zuen kritikak eta gauzak ondoegi ez doazenean asko eskertzen da. Bidea egiteko bi makuluak izan zarete. Mila esker.

Baina eskolan, izan ditut beste laguntzaile asko ere. Esker mila Goiuria, artikuluak idazteko garaian egindako ekarpenengatik. Astegun edo asteburu, zeure denbora pertsonala laguntzeko jarri izan duzu eta asko ikasi dut. Txema, Joseba, Aitor, Felix, Urtzi,...ta beste taldekideok ere ezin ahaztu. Beti irrifar batekin animoak eta indarra emanez. Urtzi, gutxi falta zaizu ta segi aurrera. Etxekoek ere eskertuko dute.

Mondragon Goi Eskola Politeknikoa osatzen dugun familia ere eskertu nahi dut. Aukera izan dut nire lana gustatzen zaidan horretan egiten jarduteko eta hori "familiak" nigan sinetsi duelako izan da. Eskerrik asko.

Eta azkenik, nire familia eder eta zoragarria: amatxo, aitatxo (zauden lekuan zaudela, zure indarra beti dago gurekin), Haritz, Txomin ta Amaia. Bereziak zarete niretzat, ta nire petralaldi guztiak zuek ordaindu izan dituzuelako. Ta zuen ondoan behar izan dudan guztietan egon ezin izan arren, beti ulertu izan duzue. Bide honetan, beti egon izan zaretelako nirekin.

Hau zuek guztion lana ere bada.

Abstract

Cyber-Physical Systems (CPSs) are embedded computing systems in which computation interacts closely with the physical world through sensors and actuators. CPSs are used to control context aware systems. These types of systems are complex systems that will have different configurations and their control strategy can be configured depending the environmental data and current situation of the context. Therefore, in current industrial environments, the software of embedded and Cyber-Physical systems have to cope with increasing complexity, uncertain scenarios and safe requirements at runtime.

The UML State Machine is a powerful formalism to model the logical behaviour of these types of systems, and in Model Driven Engineering (MDE) we can generate code automatically from these models. MDE aims to overcome the complexity of software construction by allowing developers to work at the high-level models of software systems instead of low-level codes. However, determining and evaluating the runtime behaviour and performance of models of CPSs using commercial MDE tools is a challenging task. Such tools provide little support to observe at model-level the execution of the code generated from the model, and to collect the runtime information necessary to, for example, check whether defined safe properties are met or not.

One solution to address these requirements is having the software components information in model terms at runtime (models@run.time). Work on models@run.time seeks to extend the applicability of models produced in MDE approaches to the runtime environment. Having the model at runtime is the first step towards the runtime verification. Runtime verification can be performed using the information of model elements (current state, event, next state,etc.)

This thesis aims at advancing the current practice on generating automatically Unified Modeling Language - State Machine (UML-SM) based software components that are able to provide their internal information in model terms at runtime. Regarding automation, we propose a tool supported methodology to automatically generate these software components. As for runtime monitoring, verification and adaptation, we propose an externalized runtime module that is able to monitor and verify the

correctness of the software components based on their internal status in model terms at component and system level. In addition, if an error is detected, the runtime adaptation module is activated and the safe adaptation process starts in the involved software components. All things considered, the overall safe level of the software components and CPSs is enhanced.

Laburpena

Sistema Ziber-Fisikoak, konputazio sistema txertatuez osatuta daude. Konputazio sistema txertatu hauek, mundu birtuala mundu fisikoarekin uztartzeko gaitasuna eskaintzen dute. Sistema ziberfisikoak orokorrean sistema konplexuak izan ohi dira eta inguruan gertazen denaren araberako konfigurazio desberdinak izan ohi dituzte. Gaur egungo industria ingurunetan, sistema hauek daramaten kontroleko softwarea asko handitu da eta beren konplexutasunak ere gorakada handia izan du: aurrez ezagunak ez diren baldintza eta inguruetan lan egin beharra dute askotan, denbora errealeko eskakizunak eta segurtasun eskakizunak ere beteaz.

UML State Machine formalismoa, goian aipaturiko sistema mota horien portaera logikoa modelizatzeko erabiltzen den formalismo indartsu bat da. Formalismo honen baitan eta Model Driven Engineering (MDE) enfokea jarraituaz, sistema modelatzeko erabilitako grafikoetatik sisteman txertatua izango den kodea automatikoki sor genezake. MDEk softwarea sortzeko orduan izan genezakeen konplexutasuna gainditu nahi du, garatzailei software-sistemen goi-mailako ereduetan lan egiteko aukera emanez. Hala ere, MDE-an oinarrituriko tresna komertzialak erabiliaz, zaila izaten da berauen bidez sorturiko kodearen errendimendua eta portaera sistema exekuzioan dagoenean ebaluatzea. Tresna horiek laguntza gutxi eskaintzen dute modelotatik sortutako kodea exekutatzen ari denean sisteman zer gertatzen ari denaren informazioa modeloaren terminoetan jasotzeko. Beraz, exekuzio denboran, oso zaila izaten da sistemaren portaera egokia den edo ez aztertzea modelo mailako informazio hori erabiliaz.

Eskakizun horiek kudeatzeko modu bat, software modeloaren informazioa denbora errealean izatea da (models@run.time enfokea). Model@run.time enfokearen helburu nagusietako bat, MDE enfokearekin garapen fasean sortutako modeloak exekuzio denboran (runtime-en) erabilgarri izatean datza. Exekuzio denboran egiaztapen edo testing-a egin ahal izateko lehen urratsa, testeatu nahi den software horren modeloa exekuzio denboran eskuragarri izatea da. Honela, exekuzio denborako egiaztapen edo berifikazioak softwarea modelatzeko erabili ditugun elementu berberak erabiliaz egin

daitke (egungo egoera, gertaera, hurrengo egoera, eta abar).

Tesi honen helburutako bat UML-State Machine modeloetan oinarritutako eta exekuzio denboran beren barne egoeraren informazioa modeloko elementu bidez probestu ahalko duten software osagaiak modu automatikoan sortzea da. Automatizazioari dagokionez, lehenik eta behin, software-osagai horiek automatikoki sortzen dituzten tresnak eskaintzen dituen metodologia proposatzen dugu. Bigarrenik, UML-SM oinarritutako software osagaiak automatikoki sortuko dituen herraminta bera proposatzen dugu. Exekuzio denboran eguneraketen jarraipenari, egiaztatzeari eta egokitzeari dagokionez, barne egoera UML-SM modelo terminoetan eskaintzen duten software osagaiak egiaztatzeko eta egokitzeko gai den kanpo exekuzio modulo bat proposatzen dugu. Honela, errore bat detektatzen bada, exekuzio garaian egokitze modulua aktibatuko da egokitzapen prozesu segurua martxan jarriaz. Honek, dagokion software osagaiari abixua bidaliko dio egokitzapena egin dezan. Gauza guztiak kontuan hartuta, software osagaien eta CPSen segurtasun maila orokorra hobetua izango da.

Resumen

Los sistemas cyber-físicos (CPSs) son sistemas de computación embebidos en los que la computación interactúa estrechamente con el mundo físico a través de sensores y actuadores. Los CPS se utilizan para controlar sistemas que proveen conocimiento del contexto. Este tipo de sistemas son sistemas complejos que suelen tener diferentes configuraciones y su estrategia de control puede configurarse en función de los datos del entorno y de la situación actual del contexto. Por lo tanto, en los entornos industriales actuales, el software de los sistemas embebidos tiene que hacer frente a la creciente complejidad, los escenarios inciertos y los requisitos de seguridad en tiempo de ejecución.

Las máquinas de estado UML son un formalismo muy utilizado en industria para modelar el comportamiento lógico de este tipo de sistemas, y siguiendo el enfoque Model Driven Engineering (MDE) podemos generar código automáticamente a partir de estos modelos. El objetivo de MDE es superar la complejidad de la construcción de software permitiendo a los desarrolladores trabajar en los modelos de alto nivel de los sistemas de software en lugar de tener que codificar el control mediante lenguajes de programación de bajo nivel. Sin embargo, determinar y evaluar el comportamiento y el rendimiento en tiempo de ejecución de estos modelos generados mediante herramientas comerciales de MDE es una tarea difícil. Estas herramientas proporcionan poco apoyo para observar a nivel de modelo la ejecución del código generado a partir del modelo. Por lo tanto, no son muy adecuadas para poder recopilar la información de tiempo de ejecución necesaria para, por ejemplo, comprobar si se cumplen o no las restricciones definidas.

Un enfoque para gestionar estos requisitos, es tener la información de los componentes de software en términos de modelo en tiempo de ejecución (models@run.time). El trabajo en models@run.time busca ampliar la aplicabilidad de los modelos producidos en fase de desarrollo mediante el enfoque MDE al entorno de tiempo de ejecución. Tener el modelo en tiempo de ejecución es el primer paso para poder llevar a cabo la verificación en tiempo de ejecución. Así, esta verificación se podrá realizar utilizando la información de los elementos del modelo (estado actual, evento, siguiente estado,

etc.).

El objetivo de esta tesis es avanzar en la práctica actual de generar automáticamente componentes software basados en Unified Modeling Language - State Machine (UML-SM) que sean capaces de proporcionar información interna en términos de modelos en tiempo de ejecución. En cuanto a la automatización, en primer lugar, proponemos una metodología soportada por herramientas para generar automáticamente estos componentes de software. En segundo lugar, proponemos un marco de trabajo de generación de componentes de software basado en UML-SM. En cuanto a la monitorización, verificación y adaptación en tiempo de ejecución, proponemos un módulo de tiempo de ejecución externalizado que es capaz de monitorizar y verificar la validez de los componentes del software en función de su estado interno en términos de modelo. Además, si se detecta un error, se activa el módulo de adaptación en tiempo de ejecución y se inicia el proceso de adaptación seguro en el componente de software correspondiente. Teniendo en cuenta todo esto, el nivel de seguridad global de los componentes del software y de los CPS se ve mejorado.

Contents

List of Figures

List of Tables

Acronyms

ACTL Action-based Computation Tree Logic

CHP Combined Heat and Power

CoMA Conformance Monitoring by Abstract State Machines

CPS Cyber-Physical System

CRESCO C++ REflective State-Machines based observable software COmponents

FI Fault Injection

FTM Fault Tolerance Mechanism

ICE Internet Communications Engine

LTL Linear Temporal Logic

M2M Model To Model

M2T Model To Text

MDD Model-Driven Development

MDE Model-Driven Engineering

PM Process Mining

PSSM Precise Semantic of UML State Machines

RA Runtime Adaptation

RE Runtime Enforcement

RESCO REflective State-Machines based observable software COmponents

RMVA Runtime Monitoring Verification and Adaptation

RMVAGen Runtime Monitoring Verification and Adaptation Generator

RSPC Runtime Safe Properties Checker

RSPCGen Runtime Safe Properties Checker Generator

RT Real Time

RV Runtime Verification

SC-LTL Syntactically Cosafe Linear Temporal Logic

SCUS Software Components Under Study

SFI Software fault injection

SoCPS System of CPS

SP Safe Properties

SR Safe Requirements

SRS Safe Requirements Specification

TCMS Train Control and Monitoring System

TMR Triple Modular Redundancy

UML Unified Modeling Language

UML-SM Unified Modeling Language - State Machine

Part I

Foundation and Context

Chapter 1

Introduction

Contents

This chapter introduces the main motivation and scope of the research carried out by the Ph.D. candidate and the problems that have been tackled. The selected research methodology is introduced. The main technical contributions are summarized and the publications for each of the technical contribution are highlighted. In addition, the accomplished research activities are described.

1.1 Motivation and Scope of the Research

Cyber-Physical Systems (CPSs) integrate digital cyber computations with physical processes. These CPSs are composed of embedded systems and networks that monitor and control physical processes by means of sensors and actuators [DLV12]. As stated in [AGJ+14], CPSs include embedded systems, which are inherently self-adaptive, because they are meant to observe and/or influence the environment they are embedded in. On the other hand, System of CPSs (SoCPSs) are systems where each component of the overall system is a CPS [E+14].

In our live, we are surrounded by CPSs and SoCPSs due to an increasing number of intelligent systems that involve safety, life and business-critical requirements in domains such as transportation, healthcare or home equipment. These systems directly interfere with our physical world which makes their safe, dependable and resilient operation one of their primary requirements.

In recent years, software components have gained importance as controller part of the CPSs. Control and safety features that were previously added mechanically or by hardware are now carried out by software. This has led to the control software taking more responsibility and needing mechanisms to enhance correct and safe behaviour. Furthermore, every component of a CPS is a potential point of failure. This is true not only for embedded systems but also for purely software systems such as distributed and cyber applications.

These systems require extra tasks in the development process to ensure dependability. For instance, verification and validation of software are of vital importance in order to give a certain level of confidence in the correctness of these systems. V&V techniques are effective at detecting and avoiding faulty scenarios.

In addition, mechanisms such as Fault Tolerance Mechanisms (FTMs) are used to enhance the CPSs and SoCPSs safety level. Those solutions usually need redundancy on hardware and/or software diversity which increases the final cost of the solution. However, CPS are often resource-limited because of the industrial cost requirements. Thus, developers have to deal with additional restrictions such as memory size, power and processing capacity of those CPSs. Moreover, we need a cost effective 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 CPSs at runtime can anticipate the occurrence of failures. This makes it possible to take corrective actions earlier and prevent faulty scenarios. This idea is described as a safety bag in [IEC10] and [BCLS17]. The goal is to prevent software systems' hazardous states by means of safety verification at runtime. Thus, we increase their robustness enhancing reliability.

One way to take corrective actions at runtime is by Runtime Adaptation (RA). The adaptation could be predefined or dynamic at runtime. The former could be the first step towards enhancing the overall safe operation of such systems. As stated in [CK10], the use of runtime techniques such as quantitative verification and model checking are a way to obtain dependable self-adaptive software.

Moreover, the scope, complexity, and pervasiveness of CPSs continue to increase dramatically. The inclusion of safety mechanism also increase the complexity of these type of systems. In order to manage this complexity, on the one hand, it could be a good practice to separate tasks (separation of concerns for dependable software design [JK10]) to be performed by software engineers (more focused on the logic and behaviour of the CPSs) and safety engineers (concerned about system correctness and reliability).

On the other hand, Model-Driven Engineering (MDE) aims to overcome the complexity of software construction by allowing developers to work at the high-level models of software systems instead of low-level code [AVW07]. MDE provides methodologies for software development that help in developing software components with a high degree of reliability. This approach allows us to develop software based on functional behavioural models, test its functionality at model level and also automatically generate the source code. Nevertheless, generating the code automatically is not enough. There is still a need to provide low-cost mechanisms to ensure correct and safe behaviour at runtime.

Work on models@run.time seeks to extend the applicability of models produced in the development phases to the runtime environment. Having the model at runtime is the first step towards runtime verification; and having the mechanism to adapt the model at runtime which automatically implies a software change is the next step once an unexpected situation or error is detected. Those mechanisms could contribute to enhance the safety and availability of the system.

Determining and evaluating the runtime behaviour and performance of models of CPSs using commercial MDE tools is a challenging task. Such tools provide little support to observe at model-level the execution of the code generated, and to provide

this observed information at runtime in order to verify, for example, whether defined constraints are met or not [AHJ+16].

Recent approaches recognize the need to produce, manage, and maintain software models all along the software's life time to assist the realization and validation of system adaptations while the system executes[CEG+14]. In addition, works such as [CK13] point out that the information in model terms is very useful to check the correct behaviour of the software components at runtime.

In contrast, in traditional Runtime Verification the states of an observed execution usually do not completely reflect the system's or software components' state but only contain the value of certain variables of interest.

Considering all the issues identified above, some challenges and work to be done in the area of Monitoring, Verification and Adaptation of Models at Runtime have been identified. Some of these include:

1. *How to increase the safe behaviour level of resource-limited CPSs and SoCPSs that interfere with the physical world not increasing the complexity of their development process*.

2. *How to provide low cost mechanisms based on model terms in order to enhance the safe-behaviour/reliability of CPSs and SoCPS.* In addition, this challenge could be decomposed in the following sub-challenges:

   a) How to obtain runtime information in model terms that reflects the internal status of the resource-limited CPSs.

   b) How to check the behaviour in model terms of CPSs and SoCPSs at runtime in order to enhance the safe behaviour of those systems.

   c) How to perform an adaptation process in model terms at runtime when an unexpected situation or error is detected.

1.2 Research Methodology

The selected research method is an iterative model named Design and Creation [VK04]. The methodology is composed by five phases, which are also named process steps. Each process step has an output that can be understood as the result of the activity related to the process step. Figure 1.1 depicts an overview of the methodology. The process steps are described bellow:

- **Awareness of Problem:** It is the first step, where an interesting problem is detected. The awareness of the problem might come from sources such as new developments

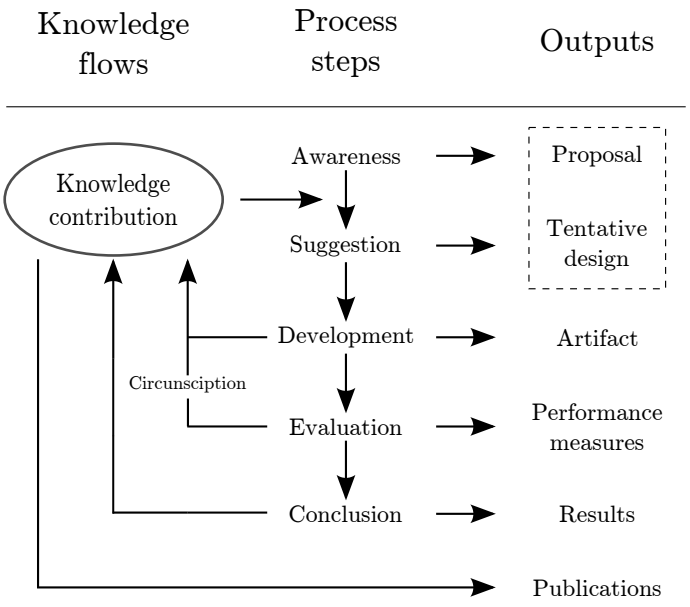


Figure 1.1: General overview of the research methodology

in the industries or reading in an allied discipline. The output of this phase is a formal or an informal proposal.

- **Suggestion:** The second step is related to the suggestion; in this phase, a creative step with novel functionalities is envisioned. A tentative design is suggested as an output and likely, the performance of a first version or a prototype of the design could be shown.

- **Development:** The tentative design is further developed and implemented in the third phase, following different techniques for its implementation depending on the artifact to be created. As output, a novel artifact is provided.

- **Evaluation:** The developed artifact is evaluated according to a certain criteria in the evaluation phase. This phase contains analytic sub-phases where hypotheses are tested. The output of this phase will be a set of fault detection and performance measures.

- **Conclusion:** It is the end of a research cycle or a research effort. It is the last phase of the iterative model and the results from the design and creation model meet the requirements specified in the previous steps. Results are consolidated and the obtained knowledge is detailed and disseminated.

1.3 Technical Contributions

The solution developed in the PhD is a models@run.time approach to enhance the safe behaviour of CPSs and SoCPSs with limited resources through introspection, verification and adaptation ability at runtime of software components. The REflective State-Machines based observable software COmponents (RESCO) methodology and framework are the solutions that enable the automatic generation of these software components with introspection and adaptation ability at runtime. In addition, an externalized runtime checker and adaptation system have been developed. The main contributions of this thesis can be summarized as follows:

1. A methodology supported by a framework, REflective State-Machines based observable software COmponents (RESCO), that is able to generate software components modeled by Unified Modeling Language - State Machine (UML-SM) that provide their internal status information in model terms at runtime.

2. RESCO framework: Automatic generation of software components with internal status information observation ability in UML-SM model terms (current state, event, next state,...). The software engineer focuses on the design of the functional behaviour of the software component, whereas the internal observability ability in model terms is added automatically. The software engineer is not involved in changing the model or source code to provide this information at runtime and thus, can focus exclusively on modelling the behaviour of the software components by UML-SMs. Additional infrastructure for having internal status information and adaptation ability at runtime is automatically added by the framework. As a result, this information could be used to increase the safe behaviour of the CPSs without increasing the complexity of the development process. Software components generated by this framework especially address resource-limited systems.

3. Runtime Verification. An external monitor and verification system is used to check the internal status of the UML-SM based software components in model terms before a transition in their state and a change in the output signal is performed. This allows us to detect faults before the failure happens, increasing the resilience against faults. Different types of monitor and verification systems have been considered: (1) monitor and verification systems that check the correct behaviour of each software component and (2) monitor and verification systems able to check the rules governing the relations between different software components in component based systems based on component level model based information. In

the latter, system-wide rules or safe properties, based on software components' internal state status information in model terms, are defined by the safety engineer.

4. Runtime Adapter. An externalized runtime adaptation module has been developed. In the solution, the adaptation is triggered by unexpected events or when a fault is detected at runtime. This allows to protect the system against unsafe situations and scenarios ensuring that the software component performs safe actions.

1.4 Publications

Different peer-reviewed publications were published at conferences during the Ph.D. Notice that some of them were directly related to the results of the thesis whereas others show results from broader research projects partially related to the scope of this work.

The conference publication papers are ranked by a raking systems supported by the Spanish Informatics Scientific Society (SCIE (www.scie.es)).[1] The journal publications are scored within the Journal Citation Report (JCR) system as well as their quartile.

1.4.1 Journal Articles

Journal Articles directly related to the work

By the time this is delivered a journal article to be sent to review at the IEEE Transactions on Engineering Management - Special Issue "Smart Services and Software Platforms" is being written.

Journal Articles indirectly related to the work

Two journal articles were published: one at Journal of Risk and Reliability and another at DYNA Ingeniería e Industria. They are listed below in chronological order:

- Elena Gómez-Martínez, Ricardo J. Rodríguez, Clara Benac Earley, Leire Etxeberria Elorza and Miren Illarramendi Rezabal. "A Methodology for Model-based Verification of Safety Contracts and Performance Requirements" in Journal of Risk and Reliability: Proceedings of the Institution of Mechanical Engineers, Part O, 2018, pp. 227-247 **JCR:** 1.373. **Q2**.

[1] http://gii-grin-scie-rating.scie.es/

- Felix Larrinaga Barrenechea, Iñigo Aldalur Ceberio, Miren Illarramendi Rezabal, Mikel Iturbe Urretxa, Txema Perez Lazare,Gorka Unamuno Eguren, Jon Salvidea Campuzano, Inaxio Lazkanoiturburu. "Análisis de arquitecturas tecnológicas para el nuevo paradigma de la industria 4.0: Analysis of technological architectures for the new paradigm of the industry 4.0" in Dyna Ingeniería e Industria, 2019, pp. 267-271 http://dx.doi.org/10.6036/8837 **JCR:** 0.5. **Q4**.

1.4.2 International Conferences

A total of 5 publications were achieved at international conferences, including SEAA, ICRE and SAC.

By the time this dissertation was submitted a conference paper was sent to review at the 30th International Symposium on Software Reliability Engineering (ISSRE 2019).

International Conference directly related to the work

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Goiuria Sagardui. "Runtime Contracts Checker: Increasing Robustness of Component-Based Software Systems" 3rd International Conference on Reliability Engineering (ICRE 2018). Barcelona. 24-26 November, 2018

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Goiuria Sagardui. "Runtime observable and adaptable UML State Machines: Models@run.time approach" 34th ACM/SIGAPP Symposium On Applied Computing (SAC), 2019, **Ranking_SCIE: A-**

International Conference indirectly related to the work

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia. "Reuse in safety critical systems : educational use case" 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Santander. 4-6 September, 2013, **Ranking_SCIE: B-**

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia. "Reuse in safety critical systems : educational use case first experiences" EUROMICRO DSD/SEAA, Euromicro Conference series on Software Engineering and Advanced Applications (SEAA) and Euromicro Conference on Digital System Design (DSD). Verona. 27-29 Agosto. Pp. 417-422, 2014, **Ranking_SCIE: B-**

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia. "Reuse in Safety Critical Systems: Educational Use Case Final Results" Proceedings 41st Euromicro Conference on Software Engineering and Advanced Applications. Funchal. 26-28 August. Pp.290 - 297, 2015, **Ranking_SCIE: B-**

1.4.3 Workshops, Fast Abstracts, Posters and National Conferences

In addition to international conferences, two national conference papers were published. Furthermore, a workshop paper at SAFECOMP-DECSOS 2017, a poster at the Safety Critical Systems Symposium and a Fast Abstract at SAFECOMP-2018 were published.

Directly related to the work

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Goiuria Sagardui. "Increasing dependability in Safety Critical CPSs using Reflective Statechart" DECSOS Workshop in Computer Safety, Reliability, and Security. SAFECOMP 2017. Trento, September, 2017

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Goiuria Sagardui. "Increasing dependability in Cyber-Physical Systems using Reflective Statecharts based Software Components" Evolution of System Safety. Proceedings of the Twenty-sixth Safety-Critical Systems Symposium. York, UK. 6-8 February,2018

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Goiuria Sagardui. "Models adaptation at runtime: enhancing the safety of software systems in uncertain scenarios" Fast Abstract Session in 37th International Conference on Computer Safety, Reliability, and Security (SAFECOMP). Vasteras, Sweden. 19-21 September,2018

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Goiuria Sagardui. "Exekuzio Denboran barne egoera ikusi eta aldatzea ahalbideratzen duten UML Egoera Makinak: Models@run.time" IkerGazte 2019. Baiona, Euskal Herria. May, 2019.

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Jose Maria Perez, Felix Larrinaga, Goiuria Sagardui. "MDE based IoT Service to enhance the safety of controllers at runtime"II International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems. Co-located with STAF 2019. Eindhoven, The Netherlands. July, 2019.

Indirectly related to the work

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia. "SafeCer: safety certification of software intensive systems" XIV Congreso de confiabilidad, libro de ponencias Madrid, 27 y 28 de noviembre de 2012. Madrid: Asociación Española para la Calidad

- Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia. "Reuse in safety critical systems : TCMS railway use case first experiences" Actas de las V Jornadas de Computación Empotrada. Valladolid. 17-19 Septiembre. Valladolid, 2014

1.5 Other Related Activities

In addition to attending the conferences of the aforementioned conference publications, the Ph.D. student has accomplished other activities that helped her in her training as a researcher. These activities have included a participation in competitions, service to the community and participation in European projects.

1.5.1 Competitions

The work developed during this dissertation was also disseminated in the following forums:

- Runtime Verification Competition: Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, Goiuria Sagardui. "Runtime Verfication of CRESCO software components: enhancing the safety of software systems". RV Benchmark Challenge 2018 (Open category). Evaluation took place during the 18th International Conference on Runtime Verification.

1.5.2 Service

Researchers are often involved in peer-reviewing articles or organizing conferences and workshops. As part of his training, the Ph.D. student has been involved in the following activities as a service to the research community:

- Reviewer for the Journal DYNA Ingeniería.

- Proceedings chair and Program committee member of the IEEE 13th International Workshop On Electronics, Control, Measurement, Signals and their Application in Mechatronics, ECMSM'17. [2]

[2] http://ecmsm2017.mondragon.edu/en

- Program committee of ACM-W: Informática para tod@s 2016-2018-2019.
- Reviewer at Jornadas Sarteco (2013-1018)

1.5.3 Project Proposals Preparation

Writing proposals for funding is one of researchers' key activities. The Ph.D. student has been involved in the proposal of an European project that is related to this dissertation. The project name was the "Productive 4.0" project, part of the ECSEL call. The project was funded by the European commission.

At this moment, in this project we are working on the proof of concept of what may be a Safety Manager in the Arrowhead framework [Del17]. This preliminary definition is based on the work presented in this document.

1.6 Document Structure

The thesis is structured as follows: the first part of the thesis corresponds to the Foundation and Context. Chapter 1 introduces the main motivation of the thesis, the employed research methodology, the contributions, the achieved publications and the activities accomplished by the Ph.D. student. Basic background as well as terminology used during the rest of the document is provided in Chapter 2. Chapter 3 gives an overview of the state of the art and highlights the most relevant studies related to this thesis. The theoretical framework is explained in Chapter 4, including the research objectives, the research hypotheses, an overview of the proposed solutions and the employed case studies.

The second part corresponds to Reflective UML-SM Software Components Generation. Chapter 5 provides the method we propose and develop for generating reflective UML-SM based software components automatically.

The third part corresponds to runtime monitoring, verification and adaptation. First, our proposal for the Software Components Level Runtime Verification and Adaptation module is presented in Chapter 6. Second, we propose a Software System Level Runtime Verification system in Chapter 7.

To conclude, in the final remarks part in Chapter 8, we summarize the contributions of the thesis, we validate the hypotheses, we discuss the main limitations of the proposed solution and we provide a set of lessons learned. Furthermore, we propose and discuss future research.

Chapter 2

Technical Background

Contents

The goal of this chapter is to make the reader familiar with the areas covered by the thesis. First, background related to the type of the systems addressed in the dissertation is provided in Section 2.1. Second, basic definitions to better understand the work are presented in Section 2.2 and in Section 2.3 we present fault injection techniques. Next, in section 2.4 verification and validation and Fault Tolerance Mechanism (FTM) are presented. Section 2.5 presents Runtime Verification (RV), Runtime Enforcement (RE) and Runtime Adaptation (RA). Lastly, background related to the concept of Models@Runtime is shown in Section 2.6.

2.1 Application Domain

The increased connectivity of embedded systems and sensors has led to the emergence of CPSs, systems of collaborative computational elements controlling a physical process. Areas such as avionics, smart grids, medical devices, traffic control, automotive are examples of domains where CPS is growing at an exponential pace.

These CPSs increasingly require cooperation from human users or operators and other embedded systems and CPSs that have been developed by third parties which may not do what they are expected to do [Gar10]. Not all the interaction possibilities and scenarios can be anticipated and there will be non-controlled and uncertain scenarios. This dissertation proposes a framework to generate automatically software components with reflection and introspection ability for embedded systems, CPSs and System of CPSs (SoCPSs). Thus, these software components will help detecting non-controlled, uncertain or erroneous scenarios based on their internal information provided at runtime and therefore, safe behaviour of these systems will be enhanced.

In the next subsections we are going to explain the basic characteristics of these kind of systems.

2.1.1 Cyber-Physical System (CPS)

Cyber-Physical Systems (CPSs) are computer-enabled mechanism interacting networks of physical and computational components with feedback loops where physical processes affect computations and vice versa. CPS will provide the basics of our critical infrastructure, in terms of emerging and future smart services, and improve our quality of life in many aspects [Wol09, Alh17].

Cyber-Physical Systems can be described in terms of five dimensions that are built upon each other as they evolve towards increasing openness, complexity and intelligence:

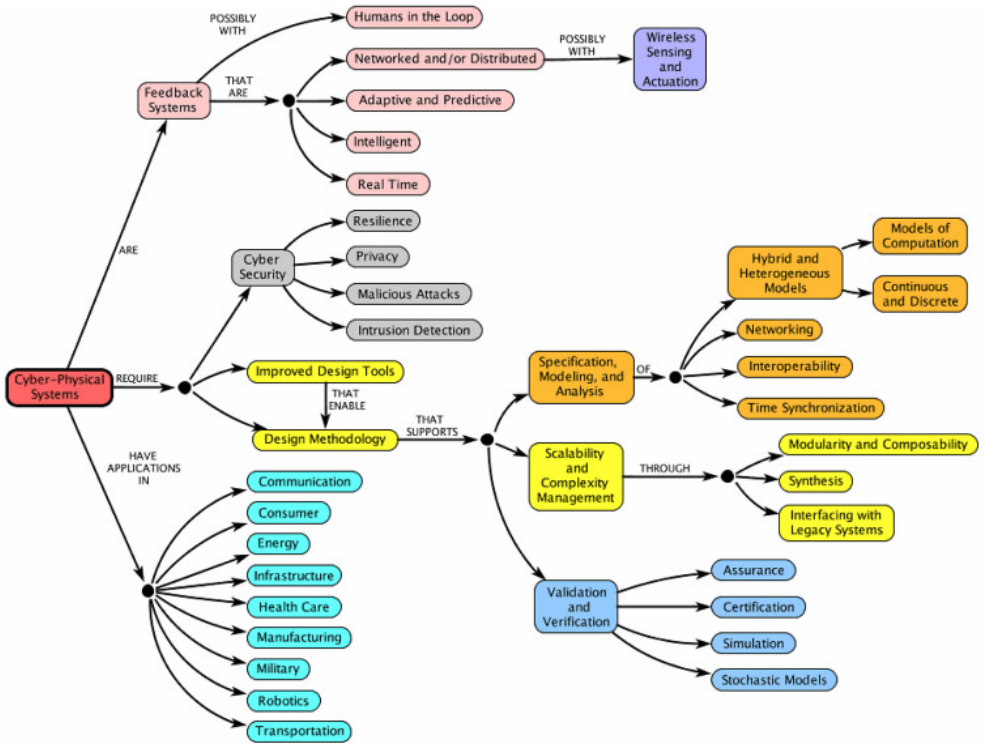


Figure 2.1: Cyber-physical systems. A concept map [ABL+17].

Figure 2.1 presents a Cyber-Physical Systems concept map.

- merging of the physical and virtual worlds,
- systems of systems with dynamically adaptive system boundaries,
- context-adaptive systems with autonomous systems; active real-time control,
- cooperative systems with distributed and changing control,
- extensive human-system cooperation.

2.1.2 Embedded Systems

An embedded system is a system that, like a CPS, incorporates a digital computing element (e.g., control software) and some physical components that interact with the physical environment [FLPV13].

The embedded nature allows the computational elements to interact directly (i) with a physical computing platform on which it is executed on and (ii) with its physical surroundings. In other words, computational logic may obtain input from sensors that

measure physical parameters, execute physical instructions of a computing platform, and provide the output to actuators that effect change in physical parameters and affect the physical behavior of the system that is controlling [MZ16].

An embedded system has a much more limited scale than a CPS; an embedded system is commonly considered at the level of a device, incorporating a limited functionality designed for a specific task with associated control logic and hardware elements [RI16].

Comparing to embedded systems, CPSs' provide bigger functionality and networking interoperability, growing importance and reliance on software and the number of non-functional constraints such as robustness or scalability [NYA+13]

2.1.3 Dependability, reliability and safety concepts and their relevance in CPSs and Embedded Systems

In this section we are going to define the concepts of dependability, reliability and safety and the interpretation we make of them in our work.

- Dependability [Som07]: the dependability of a system reflects the user's degree of trust in that system. Dependability is a non-functional requirement. Redundancy and diversity techniques are used for reaching dependability. But adding diversity and redundancy adds complexity and this can increase the chances of error. Principal dimensions of dependability are: Availability, Reliability, Safety, Security and others.

- Reliability [IEE08]: is the ability of a system or component to perform its required functions under stated conditions for a specified period of time.

- Safety [Som07]: is a judgment of how likely it is that the system will cause damage to people or its environment.

Having all these definitions into account, we can argue that the solution presented in this work does not address dependability because it does not address some of the dimensions such as security or availability. Regarding reliability, the solution addresses improving correctness of software systems and components.

The present work addresses error and uncertain situation's detection in their early stages in order to avoid hazardous scenarios. We can conclude that it is focused on improving the safe and undamaged status of software systems.

All things considered, we have defined (our interpretation) two different roles when developing software components based on our solution:

- Software Engineer: person in charge of the functional and behavioural design of software components and systems.
- Safety Engineer: person in charge of the safety related requirements.

Relevance of Reliability and Safety in CPSs and Embedded Systems The major problem of achieving reliable operations for CPS's open and networked control systems is approached using a system engineering process to gain an understanding of the problem domain. Air traffic control systems, railway signaling, automatic car braking systems, defense systems, nuclear power stations and medical equipment (increasingly including home medical electronics) are some of complex systems in use, on which life and property depend. These systems do work well because of the expertise and diligence of professional safety engineers, regulators and other practitioners who work to minimise both the likelihood of accidents, and their consequences [Clu].

In these systems the implemented functionality must not only be reliable, but must also meet real-time constraints. As a consequence, their development process is more complex and its cost increases. FTMs are commonly used mechanisms to decrease the probability of having accidents. Nevertheless, fault tolerance cannot be tackled just as a software problem due to the nature of CPSs, which includes close coordination among hardware, software and physical objects [Alh17].

There are other industrial domains that are not safety-critical but where reliability and safety are important. For instance, home appliances or electronic toys for children must avoid hazardous situations. The techniques and methods to increase the reliability and safety are usually expensive solutions. Nonetheless, these types of industries need low-cost mechanisms for the development of reliable, safe and cost-effective systems.

2.2 Faults, Errors and System Failures

First of all Fault, Error and Failure concepts are described in the following [Sto96]:

- **Fault** is a defect within the system. A faulty system is the one with defects such as malfunction of a hardware component, mistake within a piece of software or a defect in the design of the system.
- **Error** is a deviation from the required operation of the system or subsystem. The presence of a fault may lead to an error. An error is the mechanism by which the fault becomes apparent.
- **System Failure** occurs when the system fails to perform its required function.

Different authors define dissimilar fault classifications. Some of them divide faults in the next categories [Wea08]:

- Random faults: are due to physical causes and only apply to the simple hardware components within a system. This type of fault is caused by effects such as corrosion, thermal stressing and wear-out. Statistical data gathered on large numbers of similar devices may enable predictions to be made concerning the probability of a component failing within a given period of time. All physical components are subject to failure, and thus all systems are subject to random faults. Example of random faults: random bit flip, hardware component malfunction, ...

- Systematic faults: are produced by human errors during system development and operation. They can be created in any stage of the system's life including specification, design, manufacturing, operation, maintenance and decommissioning. After a systematic fault has been created, it will always appear, when the circumstances are exactly the same, until it is removed. These faults are not random and are thus not usually susceptible to statistical analysis. It is therefore more difficult to predict their effect on the reliability of the system.

Other possible classification could be made based on their duration [Dub13]:

- Permanent faults: remain active until a corrective action is taken. These faults are usually caused by some physical defects in the hardware, such as shorts in a circuit, broken interconnections, or stuck cells in a memory. All design faults and most random faults (even when their effects may not be visible all time) belong to this group.

- Transient or Temporary faults: remain active for a short period of time. A transient fault that becomes active periodically is an intermittent fault. Because of their short duration, transient faults are often detected through the errors that result from their propagation. Some hardware faults like the effects of a noise spike, or an alpha particle hitting a memory device are in this group. The errors they produce may remain in the system unless some action is taken to remove them.

Another classification which has been extensively used in this domain is the one described by Laprie [Lap92]. He defines three main groups of faults that are shown in Figure 2.2.

- Faults defined in terms of their "Nature":
  - Accidental Faults: Faults that occur randomly,

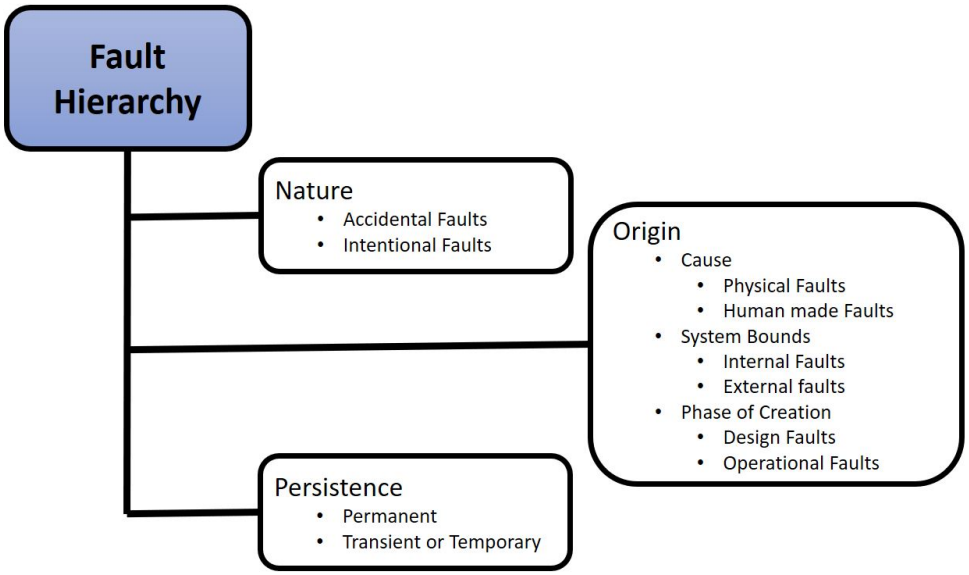


Figure 2.2: Laprie Fault Hierarchy

- Intentional Faults: Faults that occur deliberately,

■ Faults defined in terms of their "Origin":

- Cause:
  - Physical faults: Result of adverse physical phenomena,
  - Human made faults: Caused by human imperfections,
- System Bounds:
  - Internal faults: Occur within a system,
  - External faults: Result of environmental interference,
- Phase of Creation:
  - Design faults: Human made internal fault,
  - Operational faults: Occur during use of the system,

■ Faults defined in terms of their "Persistence":

- Permanent Faults: Internal or External, unrelated to specific system conditions, of indefinite duration,
- Temporary Faults: Present in system for limited time,
- Transient Faults: Temporary external physical fault.

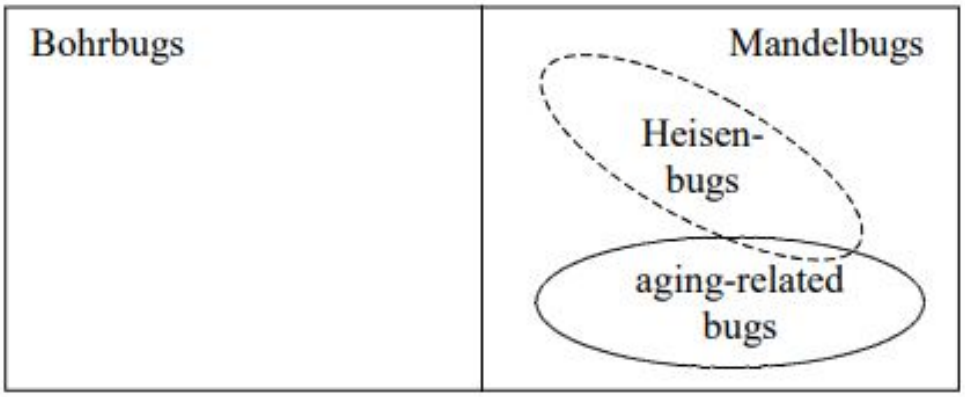


Figure 2.3: Venn diagram of software fault types [GT05].

Certain type of systems that we are addressing have to consider the Functional Safety Standard [IEC10]. This standard classifies the faults according to their:

- Causes: random (hardware) faults, systematic faults (including software faults)
- Effects: safe failures, dangerous failures
- Detectability: detected (revealed by online diagnostics), undetected (revealed by functional tests or upon a real demand for activation)

Our work is mainly focused on software faults. That is why we also considered the fault classification suggested by [GT05]. They classify the software faults in four different groups: Bohrbugs, Mandelbugs, Heisenbugs and Aging-related bugs. A Venn diagram showing the relationships between the four fault categories is depicted in figure 2.3.

- Bohrbug: A fault that is easily isolated and that manifests consistently under a well-defined set of conditions, because its activation and error propagation lack "complexity" as set out in the definition of Mandelbug. Complementary antonym of Mandelbug.
- Mandelbug: A fault whose activation and/or error propagation are complex, where "complexity" can take two forms:
  - The activation and/or error propagation depend on interactions between conditions occurring inside the application and conditions that accrue within the system internal environment of the application.
  - There is a time lag between fault activation and failure occurrence, e.g., because several different error states have to be traversed in the error propagation.

Typically, a Mandelbug is difficult to isolate, and/or the failures caused by it are not systematically reproducible. Complementary antonym of Bohrbug.

- Heisenbug: A fault that stops causing a failure or that manifests differently when one attempts to probe or isolate it. Sub-type of Mandelbug.

- Aging-related bug: A fault that leads to the accumulation of errors either inside the running application or in its system-internal environment, resulting in an increased failure rate and/or degraded performance. Sub-type of Mandelbug.

There are other type of errors such as operation errors or maintenance errors that we are not going to consider them in this work.

2.3 Fault Injection techniques

Software fault injection (SFI) is an acknowledged method for assessing the reliability of software systems. The solution presented in this work has been evaluated by implementing some experiments (based on industrial use cases) and in order to simulate faults and errors that would have the evaluated systems in real scenarios we have used these SFI techniques. When an injected fault causes a system failure, this can indicate insufficient fault tolerance mechanisms. Fault Injection (FI) campaigns have been used to test system's robustness.

Fault injectors can be custom-built hardware or software and they can support different fault types, fault locations, fault times, and appropriate hardware semantics or software structure.

Choosing between hardware and software fault injection depends on the type of faults you are interested in and the effort required to create them. For example, if you are interested in stuck-at faults (faults that force a permanent value onto a point in a circuit), a hardware injector is preferable because you can control the location of the fault. The injection of permanent faults using software methods either incurs a high overhead or is impossible, depending on the fault. However, if you are interested in data corruption, the software approach might suffice. Some faults, such as bit-flips in memory cells, can be injected by either method. In a case like this, additional requirements, such as cost, accuracy, intrusiveness, and repeatability may guide the choice of approach.

2.4 Verification & Validation and Fault Tolerance Mechanisms

2.4.1 Verification & Validation

The main objective in Verification & Validation (V&V) of software requirements, design specifications and source code is to identify and resolve software problems and high-risk issues early in the software life-cycle. These V&V techniques are used to detect mainly systematic faults. However, it is not always possible to detect all types of faults during this phase. Residual or remaining software faults are included in this category. We can define them as systematic faults that remain after the validation and verification of design and development phase.

In order to detect this type of faults and others defined as unanticipated faults (not considered in the design and development phase and dependent of the final environment of the CPS) we will need Fault Tolerance Mechanism (FTM) or other techniques such as Runtime Verification.

2.4.2 Fault Tolerance Mechanisms

The goal of a fault tolerant system is to provide safety, liveness and avoid system failures even if faults are present. If a fault occurs, the system hides its effects. Safety alone is not sufficient because it does not guarantee that the system does anything useful. The system has to be both, safe and alive, even in presence of faults.

The fault tolerance process is that set of activities whose goal is to remove errors and their effects from the computational state before a failure occurs. The process consists of [Pul01]:

- Error detection: in which an erroneous state is identified;
- Error diagnosis: in which the damage caused by the error is assessed and the cause of the error is determined;
- Error containment/isolation: in which further damages are prevented (or the error is prevented from propagating);
- Error recovery: in which the erroneous state is substituted with an error-free state.

This process plays an increasing role for technical processes in order to improve reliability, availability, maintenance and life-time of software components. Having as an objective to improve the safe behaviour of the addressed systems by using these

fault tolerance mechanisms, one of the first step would be to reduce the number of faults and errors. Whatever the reason for a fault, we would like the software to be able to recognize that there is an error and to avoid and recover from it before it becomes a failure. There are four steps for doing this: (1) noticing that there is an error; (2) when possible, diagnosing exactly which of the software components is affected by the error; (3) finding an alternative way of achieving a safe intended behaviour; and (4) executing the alternative way identified in the previous step.

Error detection is one of the most important aspect of fault tolerance because a system cannot tolerate a problem of which it is not aware. The starting point of any fault tolerance activity is error detection. Error detection mechanisms are often referred to as "failure/fault detection", and the presence of an error in the output of a component is declared as failure of the component.

An error is observable if there is information about its existence available at the system interface. The information that indicates the existence of a fault is a symptom. A symptom may be a directly observed error or failure, or it may be a change in system behavior while still the system meets its specifications.

A system is fault tolerant if it can mask the presence of faults in the system by using redundancy [JJ94]. Redundancy is the key to support fault tolerance; there can be no fault tolerance without redundancy. Redundancy is defined as those parts of the system that are not needed for the correct functioning of the system when the system has not faults. There are different types of redundancy: hardware, software, information and time [Avi76, PG05]:

- Hardware: Hardware redundancy duplicates entire components and compares the results on equality. A majority voter takes at least two matching results out of three, whereas a comparator only detects the fault but cannot decide which result is right.
- Software: Software redundancy executes the same piece of software programmed in diverse implementations and compares the outcomes.
- Information redundancy: Information redundancy adds extra nonfunctional relevant information to the original data like checksum or duplication to data to detect faults. The fault coverage depends on the quality of the error detection correction code.
- Time: Time redundancy executes the task several times. It is the additional time that is used to deliver the service of the system (e.g., multiple execution of the operation)

Hardware redundancy may be partial (redundancy of processors, hard disks, network adapters,...) as well as complete with replication or software diversity. In

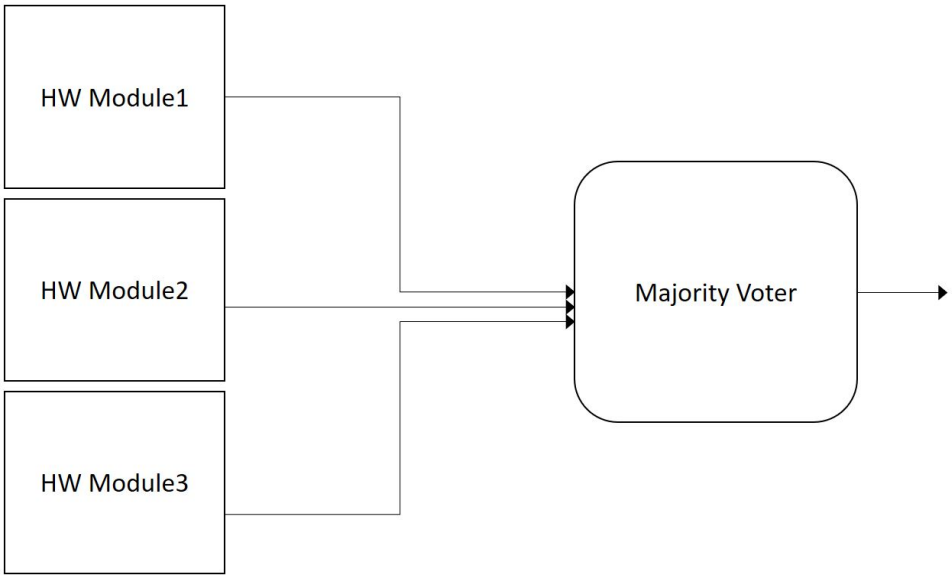


Figure 2.4: Hardware redundancy example: Triple Modular Redundancy.

Figure 2.4 a typical Triple Modular Redundancy (TMR) architecture is shown. TMR is a fault-tolerant form of N-modular redundancy, in which three systems perform a process and that result is processed by a majority-voting system to produce a single output. If any one of the three systems fails, the other two systems can correct and mask the fault.

Software diversity is usually used to prevent software or hardware failures caused by design faults. In this case, different versions of the software are developed and the software for each version must be developed by different teams. To prevent non evident failures, the voting scheme must be used [BJRT06].

The incorporation of redundancy in a software system requires a structured and disciplined approach, otherwise it may increase the complexity of the system and consequently it may decrease, rather than increase the system's robustness.

2.5 Runtime Verification, Enforcement & Adaptation

In the next subsections we will present the different runtime techniques that have been considered in the solution developed in the thesis.

2.5.1 Runtime Verification

Rather than replicate a piece of hardware or a piece of software, another approach to error detection is *dynamic or runtime verification*. Dynamic or runtime verification is

performed during the execution of software, at runtime, and dynamically checks its behaviour.

Runtime Verification (RV) consist of extracting information from a running system and using it to detect, and possibly react to, observed behaviors satisfying or violating certain properties [DGH+16]. Runtime verification avoids the complexity of traditional formal verification techniques (like model checking and theorem proving) by analyzing only one or a few execution traces and working directly with the actual system.

2.5.2 Runtime Enforcement

Runtime Enforcement (RE) consists of monitoring the controller in order to ensure that the system behaviour is always in agreement with the correct specification [CFAI18]. Before performing a transition to another state, this technique checks the current status of the system and in case it identifies something wrong, it does not conclude the transition. The runtime monitor should therefore be capable of anticipating incorrect behaviour before the transition actually happens. One way to perform runtime enforcement is to observe the runtime information (traces) sent by the software controller to an externalized runtime verification module. Since correct traces will be finite and predefined in the verification module, when the received trace is not defined as a correct one, the verification module comes to a state that trace-violation has been detected.

2.5.3 Runtime Adaptation

Once an error is detected at runtime, one possible solution is to adapt it to ensure a safe situation. This technique is known as Runtime Adaptation and it is a way to enhance the safe operation of a system. Nevertheless, the adaptation process itself has to be safe, too.

Runtime Adaptation (RA) is a technique prevalent to long-running, highly available software systems, whereby system characteristics (e.g., its structure, locality etc.) 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 [CF15].

Regarding the concepts of dynamic adaptation in general, in [TAFJ07] they distinguish between dynamic behavior adaptation and dynamic reconfiguration. Dynamic adaptation of behavior enables unconstrained adaptation by modifying the actual behavior, examples are artificial neural networks and evolutionary algorithms. This provides the highest flexibility, but it is unpredictable, not verifiable and therefore not

applicable for safety-critical systems. Dynamic reconfiguration provides constrained adaptation by defining a set of behavior variants at design time and selecting one of these predefined variants at run time. Compared to dynamic behavior adaptation, dynamic reconfiguration is less flexible, but the resulting adaptation behavior is predictable.

Other definitions for runtime adaptation are given in [GAM17]. In this case, they define two main dynamic software adaptation approaches using runtime models of the software but from a different point of view compared to the previous definitions given in [TAFJ07]. In this case, they define planned and unplanned adaptations. Planned adaptation is proactive in which manual or automated decisions are made to dynamically change the software system at runtime. The unplanned adaptation is triggered by unexpected events or when a fault is detected at runtime and reactive decisions are needed to dynamically adapt the system to avoid system failures.

In [GS02] 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. In figure 2.5, the architecture of the solution is shown.

When designing adaptive systems, we need to design different modes of operation to evolve the system at runtime. 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 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 [Dha17].

To accomplish the adaptation process, the externalized components (1) maintain the model of the monitored running software component and (2) support reasoning about system problems.

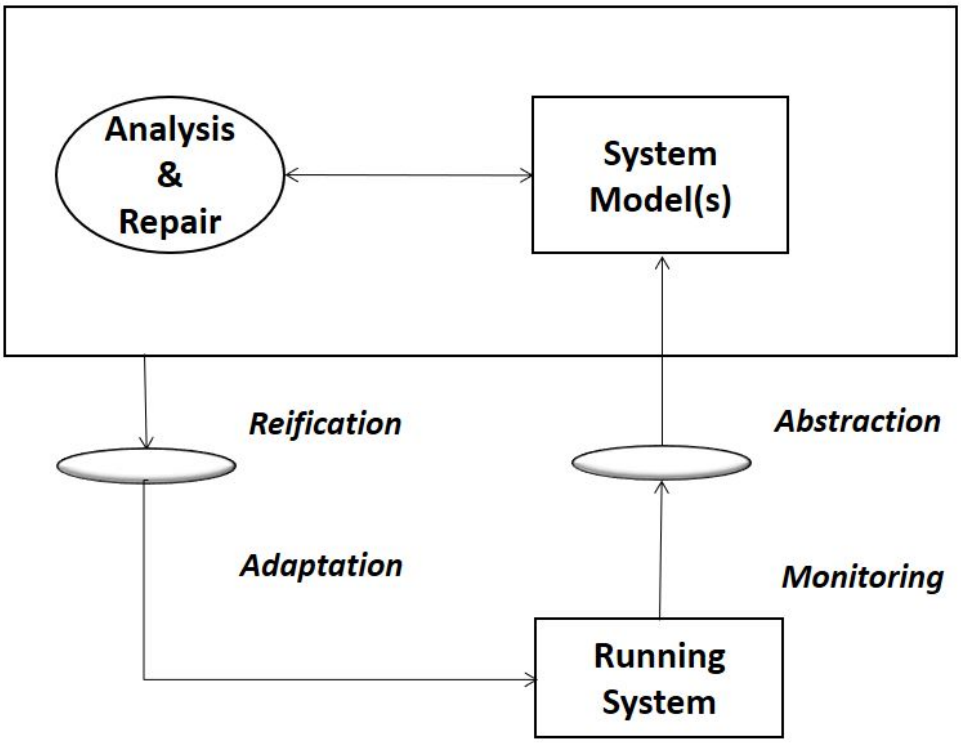


Figure 2.5: Model-based Adaptation

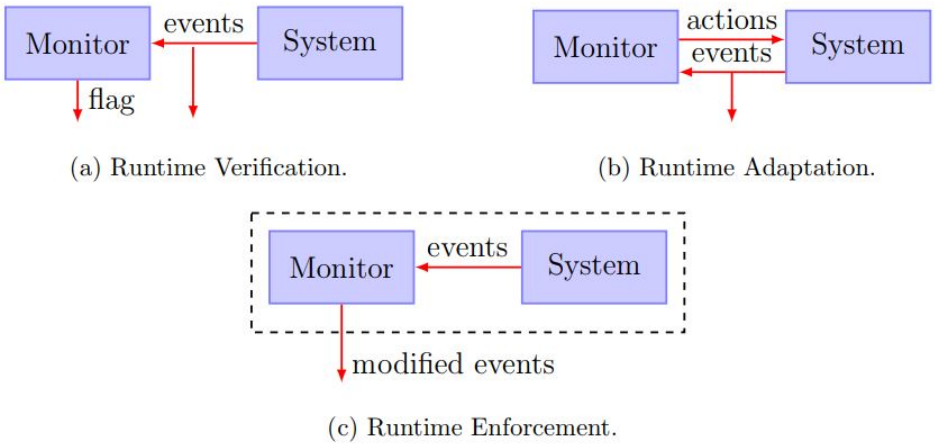
Adaptive Fault Tolerance

A system that remains dependable when facing changes is called resilient. The fast evolution of systems, including safety critical systems, requires that fault tolerance mechanisms remain consistent with their assumptions and the non-functional requirements of the application. A change event may impose the adaptation of a Fault Tolerance Mechanism (FTM) in order to provide an appropriate response to the new system's assumptions. Consequently, system resilience should rely on adaptive fault tolerant computing [EFL17].

2.5.4 Runtime techniques comparison

In figure 2.6, the differences between the presented different techniques (Runtime Verification (RV), Runtime Adaptation (RA) and Runtime Enforcement (RE)) are shown.

As we can see, in Runtime Verification (RV) monitors adopt a passive role which is to receive system events and detect violations. On the other hand, monitors in Runtime Adaptation (RA) are not passive and in this case, they execute adaptation actions once



(a) Runtime Verification. (b) Runtime Adaptation.

(c) Runtime Enforcement.

Figure 2.6: Distinguishing between Runtime Verification, Adaptation and Enforcement [CFAI18]

they detect a wrong behaviour. In the last approach, Runtime Enforcement (RE), the system behaviour is kept in line with the correctness requirement. For doing that, the incorrect behaviour is avoided by the monitor which analyses the transitions of the system before executing them. Thus, the incorrect event is suppressed by inserting events that avoid the incorrect situation. This contrasts with RA, where the monitors may allow a violation to occur but then execute remedial actions to mitigate the effects of the violation.

2.6 Models@Runtime: Basic Concepts

2.6.1 Reflection and Introspection

Reflection [Mae87a] can be defined as the property by which a component enables observation of its own structure and behavior from outside. A reflective system is basically structured around a representation of itself or a model that is causally connected to the real system.

A reflective program is one that reasons about itself. A fully reflective procedural architecture [Smi84] is one in which a process can access and manipulate a full, explicit, causally connected representation of its own state.

As stated in [RKFTF03], a reflective component provides a model of itself, including structural and behavioral aspects, which can be handled by an external component. This information is used as an input to perform appropriate actions for implementing non-functional properties (concerning, for instance, fault-tolerance or security

strategies). The reflective systems that we consider are thus structured in two different levels of computation: the base-level, in charge of the execution of the application (functional) software and the meta-level, responsible for the implementation of observation and control (non-functional) software. The meta-level software has a runtime view of the behaviour and structure of its base-level and changes/adaptations are programmed at meta-model.

Generally, reflection is about inspecting (observing and therefore reasoning about) and changing the internal representation (structural reflection) and also reasoning about the normalization (behavioral reflection) of a system. To be more specific, the inspection of the internal representation is typically called introspection and changing the internal representation and also reasoning about the normalization is well-known as intercession and the mechanism that enable these manipulations is called reification [Jam12].

Introspection supports runtime monitoring of the program execution with the goal of identifying, locating and analyzing errors [JSZ10].

The computational reflection as a concept is originated by Brian Cantwell Smith [Smi82] and it was elaborated by Pattie Maes [Mae87b] and Gregor Kiczales [KDRB91] and his colleagues in programming languages. Afterward, the emerging new paradigms and the need for distributed and transparent systems by embracing the component-based concepts, the computational reflection found its way all the way down to distributed systems and their underlying middleware infrastructures to make them highly adaptive. As a result, a number of reflective component model specifications (Fractal) and a middleware implementation (Julia) had emerged. Finally, by arising the need for dynamic software evolution in self-* systems, the need for explicitly maintaining the architectural description which causally connected to runtime model was raised. "Causally connected" means that any changes made to a process's self-representation are immediately reflected in its actual state and behavior.

These concepts have been extensively explored by Walter Cazzola, Peyman Oreizy, Nenad Medvidovic, Richard Taylor, Jeff Magee, Jeff Kramer and their colleagues in [OMT98].

A runtime model provides a view on a running software system that is used for monitoring, analyzing or adapting the system through a causal connection between the model and the system. Most approaches, such as [GCH+04, MZ16], employ one causally connected runtime model that reflects a running system. As shown in figure 2.7 Runtime Models are divided into two top categories, Reflection and Adaptation Models, based on the way they are used at runtime.

As an extension of the work in the area of reflective components a new research

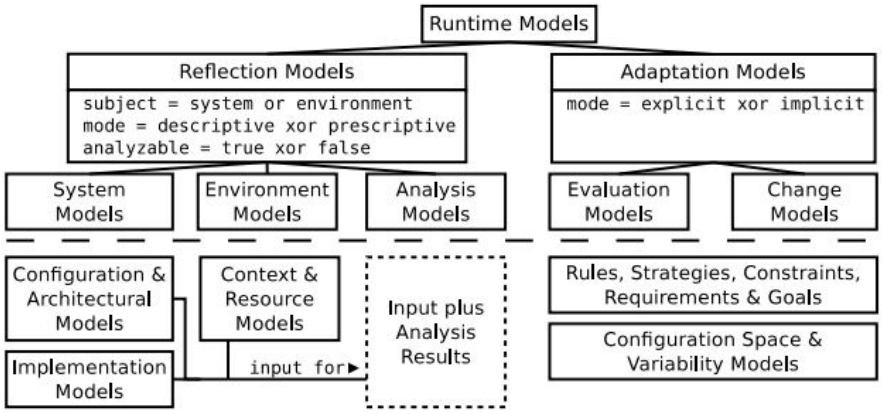


Figure 2.7: Categories of Runtime Models for Self-Adaptive Software Systems [VSG10]

line focused on Models@runtime started at the beginning of the last century. The first edition of the workshop Models@run.time was celebrated in 2006. This approach aims to gain of the benefits that provides the MDE techniques.

2.6.2 Model-Driven Engineering (MDE) & Models@Runtime

Model-Driven Engineering (MDE) aims at facilitating the development of complex, reliable and highly reusable systems by using models as the primary artifacts of the software development process [BSAN17].

The models@runtime pattern [MBJ+09] applies these MDE ideas to self-adaptive systems: The system thus maintains a domain-specific model of its state (tailored for a given adaptation) and any change made to this model is automatically synchronised with the running system. A models@runtime environment keeps a model in synchrony with a running system, this way a reasoning engine adapts the system by modifying this model. Existing models@runtime environments typically fail to let the user control not only which are the elements or concepts that form the model but also how the model is synchronised with the running system [FCSS17].

Chapter 3

State of the Art

Contents

In this chapter, we give an overview of the state of the art and highlight the most relevant studies related to this thesis. Furthermore, a critical analysis of the state of the art in runtime verification and adaptation and the Models@run.time approach is performed, which aims at finding research opportunities.

3.1 Model Driven Approach: Models@run.time

Event-driven architecture is a commonly used pattern to develop reactive control systems, and Unified Modeling Language - State Machines (UML-SMs) constitute a widely used formalism to design the behaviour of such systems. Following the Model-Driven Engineering (MDE) approach, we are able to design and develop the control system by models, but then, although the final code can be obtained automatically assisted by a code generation tool, at runtime the model itself is lost. In the next subsections, we are going to present different topics that address the Models@run.time approach. First, in subsection 3.1.1 solutions that adds the ability to trace the UML-SM based software components are presented. Then, in subsection 3.1.2 different solutions for Reflective State Machines with introspection and reflection ability at runtime are shown. Last, subsection 3.1.3 presents different tools that generate code automatically from models.

3.1.1 Tracing UML State Machines

Some research lines are open seeking how to maintain the models also at runtime but still more mature results are needed. In the next paragraphs, some of the current works are presented.

Mazak et al. propose an execution-based model profiling as a continuous process to improve prescriptive models at design-time through runtime information in [MWPB16]. In order to have runtime information in model terms, they defined a observation language which determines the runtime changes to be logged. The code generator provides the appropriate logging line to that. In their solution, they instrument the source code and not the model. Source code instrumentation technique is used to modify source code to insert appropriate code (usually conditionally compiled so you can turn it off). Programmers implement instrumentation in the form of code instructions that monitor different aspects of a system. In contrast to this technique, model instrumentation technique specifies which elements of the model will be monitored at runtime and it automatizes the source code instrumentation. The latter technique is more reliable as it is an automatic process and is less likely to insert faults.

Table 3.1: Approaches to generate runtime traces from UML-State Machines

Approaches	Instrumentation (Code or Model)	Model persists at runtime
[MWPB16]	Code	Yes
[DGJ+16]	Code	Yes
[ALG15]	Code	No
[SSL+14]	Code	Yes
[BHD17]	Model	Yes

In the same vein, in [DGJ+16] Das et al. present their solution based on instrumenting the code (not the model) to monitor real-time embedded systems at runtime. They combine the use of MDE, runtime monitoring, and animation for the development and analysis of components in real-time embedded systems.

The solution presented in [ALG15] defines a textual language for trace specification of state machines. As the aforementioned works, this approach is also based on code instrumentation. They define different commands to trace different specifications at runtime but the solution does not provide information at model level and the logged information is not related to model elements: it is not a model centered solution.

In [SSL+14] Saadatmand et al. propose a solution for runtime verification of state machines based on model-based testing. Their solution does not generate automatically the final code and, as the previous solutions, they instrument the code manually to have the software components information in model terms at runtime.

In [BHD17] they present a platform-independent model-level debugger. The solution is focused on real-time embedded systems. This approach relies on model transformation to instrument the model to be debugged. When instrumenting the model, they add a new object for each transition that debugs/traces the execution. The presented solution is applicable to models expressed in UML for Real-Time (UML-RT).

In the Table 3.1 the summary with the main characteristics is shown. We can conclude that only one of the solutions ([BHD17]) is able to instrument the model and not the source code and most of them provide the information in model terms at runtime.

3.1.2 Reflective UML State Machines

The translation to code of UML-SMs have suffered an extensive evolution. First, we are going to have a brief review of this evolution in order to have a better understanding of how reflective ones can be obtained.

In [Sam02] there is an initial survey about how state machine models have been

transformed to code when using procedural languages. Basically two approaches have been used (and are still in use). The first one consist of a switch sentence nested within another, being the first one for state selection and the second for current processing event, and thus the desired activity can be launched. The second one organizes the activities in a matrix-like structure according to states and transitions.

With object orientation, new design patterns were created aiming at better code structuring or easier modification: In [Gam95] it can be found the well known State Patterns; [Ada03] is about an anthology of finite state patterns.

[Sam02] introduced the so called Quantum Programming aimed to C++ programming language. The author extended the idiom of double-switch so it could be used with hierarchical state machines, but lacking some elements of the state machine specification. [PM03] improved it fixing some alteration of the state machine semantic and adding support for concurrent regions.

[AT99] also extended the State Pattern of [Gam95] defining the so called Helper Object Pattern in order to support state hierarchy for Java language, but also lacking support for concurrent regions and other elements of the state machine specification.

Boost statechart library [Dön07] is a library aiming at transforming from UML-SM to executable C++ code and vice versa. This solution, adds the option to extract the model from the code and it has the possibility to design hierarchical state machines as well as models with concurrent regions. One of its drawback is that it makes an extensive use of C++ templates becoming impractical for big sized state machines.

All the mentioned solutions were not considering reflective state machines. [FR98] created the Reflective State Pattern for finite state machine aiming at reflecting the state structure of a component and changing its behaviour at runtime for tolerating faults. [VGB00] was perhaps the first that proposed to reflect in code not only state but also transitions.

But to our knowledge, it is [Bar06] who in first place created a framework for state machine based component that implicitly supported runtime introspection of a UML-SM based software component at runtime. Based on this work, [BHB09], [Bar08] and [EMS$^+$08] propose a component model that carries models at runtime, focusing on UML-SM. The frameworks of [BHB09] and [EMS$^+$08] provide a runtime state-based component model. In addition, [EMS$^+$08] defined a framework for Java that supports runtime modification of the behaviour of a state machine based software component.

Table 3.2 shows the summary of the evolution of UML State Machines. We can conclude that only one of the solutions ([BHB09]) has the reflection ability but it does not address resource-limited solutions (it is implemented in Java). On the other

Table 3.2: Evolution of UML State Machines

Approach	Language	Resource-Limited Solution	Hierarchical SM	Concurrent Regions	Reflection
[Sam02]	C++	Yes	Yes	No	No
[PM03]	C	Yes	Yes	Yes	No
[AT99]	Java	No	Yes	No	No
[Dön07]	C++	Yes	Yes	Yes	No
[BHB09]	Java	No	Yes	Yes	Yes

hand, there are three solutions that could address resource-limited solutions ([Sam02], [PM03] and [Dön07]) but they have not reflection ability.

3.1.3 Automatic UML-SM to Code Generation Tools

There are many commercial tools that transform state machine specification to code, but this is their only aim. Moreover, the transformation rules are quite tool and version specific.

Table 3.3 shows the different solutions to generate code and their characteristics. One of the main conclusion is that there is a lack of solutions able to generate UML-SM based software components that provide runtime information in model terms.

Analyzing this table, we can divide it in two big families. On the one hand, we have generic frameworks that are able to generate code from UML-State Machines for different programming languages. They address different types of solutions and are not specialized in resource limited systems (EA, Yakindu and IBM Raphsody). On the other hand, there are other frameworks that address embedded systems (SinelaboreRT, QP/C++, Papyrus-RT and Pham). The last solutions are able to generate code for C++. We have to add that none of them is able to provide the UML-SM model at runtime.

Table 3.3: Tools and methods for code generation

Tools	**UML-SM Code Generators**	**Model at Runtime**	**Addressed Systems**
EA [Sys15]	Java, C++,...	No	Not resource limited systems
SinelaboreRT [Mue18]	C++/Python/...	No	Embedded Systems
Quantum (QP/C++) [Qua18]	C++	No	Embedded Systems
Yakindu [ite18]	C++/C/Java	No	Different types of solutions
Papyrus-RT [ecl18]	C++	No	Embedded Systems
IBM Raphsody [IBM]	C++, Java,...	No	Different types of solutions
Pham [PRGL17]	C++	No	Embedded Systems

3.2 Runtime Verification and Adaptation

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 CPSs where a fault can lead to a critical scenario, we need a way to detect and mitigate hazardous and uncertain scenarios. Runtime verification and adaptation techniques could be used to maintain safe control in unanticipated circumstances.

In the following subsections, we are going to present different topics that address Runtime Verification and Adaptation techniques. First, in subsection 3.2.1 differences and details about hardware and software runtime verification system are shown. Then, in subsection 3.2.2 different software monitor solutions that enables runtime verification are presented. After that, in subsection 3.2.3 we will present different abstraction levels for doing runtime verification. Another research topic is the generation of the runtime verification system itself. Subsection 3.2.4 tackles this topic and presents different tools for generating runtime verification systems. Lastly, subsection 3.2.5 is focused on showing different solutions for Runtime Adaptation (RA).

3.2.1 Hardware and Software Runtime Verification

Both runtime software and hardware verification systems have been studied in [DGR04]. While hardware verification systems detect errors at bus level, before change in state happens and preventing a catastrophic failures before it occurs, software verification systems detect errors once a change in state has occurred. Hardware verification systems, such as the Noninterference Monitoring Architecture [TFCB90], need additional hardware to collect state information and to assist in checking.

In addition, most runtime software verification systems require modifying the source code of the observed system by instrumented code. However, it is desirable that runtime verification systems for testing safe properties of the systems should be isolated from the target system to minimize any disruption of the system being tested [KFK14].

3.2.2 Software Monitors for Runtime Verification

In this subsection we are going to present different runtime software monitors that are used for Runtime Verification.

In [ea11a], they defined a generic software monitoring model and analyzed different existing monitors. Depending on the programming language used and the formalism used to express the properties, different implementations of monitors have

been proposed [MGE], among others CoMA [ea11b], RV-MONITOR [ea16] and AspectC++ [ea02], BEE++ [BGL93], DN-Rover [DJC94], HiFi [AS98] etc.

CoMA (Conformance Monitoring by Abstract State Machines) [ea11b] is a specification-based approach and its supporting tool for runtime monitoring of Java software. Based on the information obtained from code execution and model simulation, the conformance of the concrete implementation is checked with respect to its formal specification given in terms of Abstract State Machines.

RV-MONITOR [ea16], is a software analysis and development framework that aims to reduce the gap between specification and implementation by allowing them together to form a system. In this case, monitors are synthesized from specifications and integrated into the original system to check its behavior during execution.

Kieker framework is presented in [vHRH$^+$09]. This framework is able to monitor software runtime behaviour. Its flexible architecture allows developers to replace or add framework components, including monitoring probes, analysis components, as well as monitoring record types shared by logging and analysis. As a non-intrusive instrumentation technique, Kieker employs, but is not restricted to, aspect-oriented programming. The missing point of this solution is that the runtime information is not expressed in model terms: a bridge between development models and runtime models has not been built.

[ZOKR06] presents a Model-based runtime Verification Framework for Self-optimizing Systems. This framework works at model level and both models and properties must be known ahead of time. The framework is based on real-time UML state machines and the properties as well as assertions and invariants to be checked are limited to time-annotated Action-based Computation Tree Logics (ACTLs) and Linear Temporal Logics (LTLs).

[PM05] presents a runtime Verification Framework for concurrent monitoring of applications specified by UML-SM. The framework integrates two aspects of verification, temporal Syntactically Cosafe Linear Temporal Logics (SC-LTLs) and implementation errors, with the corresponding error-handling by introducing the concept of exception events as error indicator events. A runtime verifier module is integrated in the framework and this module is the one that sends the exception events to the control system in order to detect errors.

Table 3.4 shows the summary of some of the studied software monitors. The information shown in the table is classified in terms of:

- Externalized RV: runtime verification system is not integrated with the software component that is verifying.

Table 3.4: Software Monitors for Runtime Verification

SW Monitor	Externalized RV	Model based formalism for RV	Programming Language
CoMA [ea11b]	Yes	Abstract SM and Simulator	Java
RV-MONITOR [ea16]	No	Finite SM and LTL (Lineal Temporal Logic)	Java, C
Kieker [vHRH+09]	Yes	No model based solution	Java
RV-Self Optimizing Systems [ZOKR06]	Yes	LTL (modal temporal logic)	Java
Pinter [PM05]	No	UML-SM and LTL	Java

- Model Based Formalism for RV: Formalism that is using the runtime verification system to check the correctness of the software component.
- Programming Language: Programming language which is used to implement the runtime verification system.

We can summarize that there are externalized and integrated solutions and most of them are based on model based formalism.

3.2.3 Runtime Verification: Specification Level vs. Monitoring Level

Current runtime verification solutions as shown in Figure 3.1, are specified at different abstraction levels: system, component, class, method or statement. As it can be observed, most of the approaches check if the specification is fulfilled at the same level that monitoring is performed. This is the case of all the solutions presented in 3.2.2: specifications and the verification systems checking properties are at the component or class level. Thus, for the detection of system level misbehavior, 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.

As far as we know, there are only two works that consider this topic. In [PW16], they present a solution called LuMiNous which checks system level specifications by components' level information. The contribution of this framework relies on the translation of high-level specifications into runtime monitors but, in this case, their

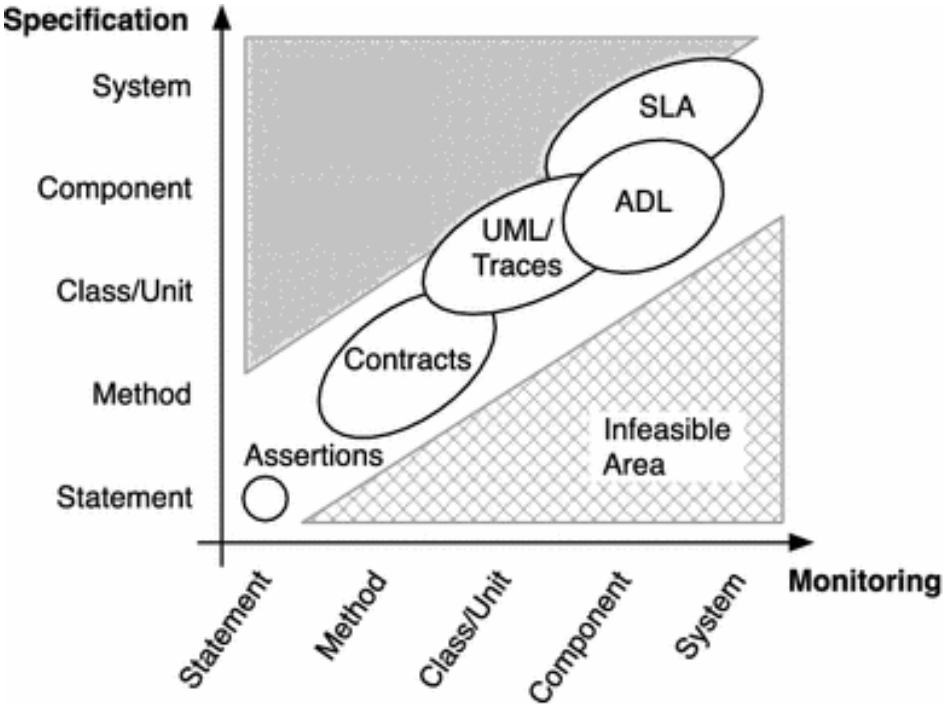
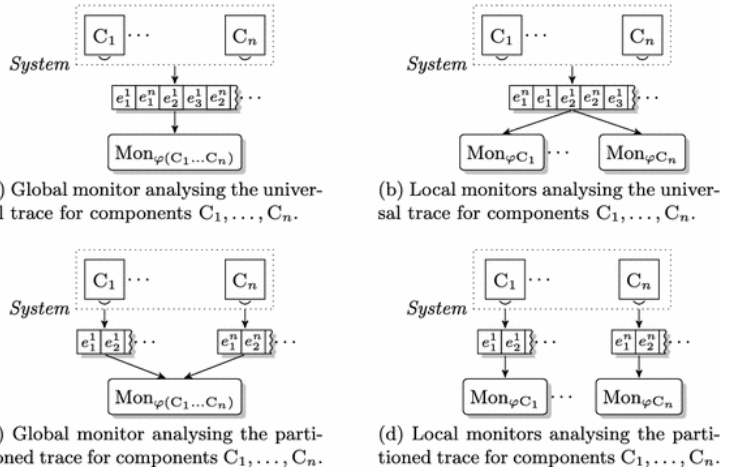


Figure 3.1: Abstraction level of specifications vs. runtime monitoring abstraction levels based on [PW16]

solution is for Java (AspectJ based solution), which is not suitable for embedded systems.

There is another work [AF17] that presents an instrumentation technique for monitoring asynchronous components. Their solution generates partitioned traces reflecting the interleaved execution of the constituent components under scrutiny. In addition, the solution considers four architectural set-ups for component-based runtime monitoring: (a) Global monitors analysing the universal trace for components, (b) Local monitors analysing the universal trace for components, (c) Global monitor analysing the partitioned or component specific traces, and (d) Local Monitors analysing the partitioned or component specific traces. In figure 3.2 the approach is shown.

Table 3.5 summarizes the section. Neither of the approaches relies on model based formalism to perform the runtime verification.



(a) Global monitor analysing the universal trace for components $C_1, \ldots, C_n$.

(b) Local monitors analysing the universal trace for components $C_1, \ldots, C_n$.

(c) Global monitor analysing the partitioned trace for components $C_1, \ldots, C_n$.

(d) Local monitors analysing the partitioned trace for components $C_1, \ldots, C_n$.

Figure 3.2: Four architectural set-ups characterizing component-based runtime monitoring [AF17]

Table 3.5: Software Monitors with different abstraction levels for specification and monitoring

SW Monitor	Externalized RV	Model based formalism for RV	Programming Language
LuMiNous [PW16]	Yes	No	Java
RV-Async [AF17]	Yes	No	Erlang

3.2.4 Automatic generation of Runtime Monitors

One of the first works exploring the idea of generating monitors from a specification and developing a tool chain for it was the MaC (Monitoring and Checking) framework [ea04]. They presented a tool that uses the specification information to generate the verification system. It is a debugging tool, designed to supplement testing. While the MaC framework showed the effectiveness of monitoring systems for compliance with specification, the framework's target was primarily for software systems with instrumented code, requiring modifications to the source code and making it unsuitable for usage in safety-critical embedded applications.

In [AT04] a runtime Monitoring of Reactive System Models is introduced. The core of the approach is automatic creation of monitoring state machines from formulas that specify the system's behavioral properties in a proposed assertion language. Such monitors are then translated into code together with the system model, and executed concurrently with the system code.

Another related work is [BLS] which presents a dynamic model-based analysis approach for distributed embedded systems, based on runtime reflection. In this case,

Table 3.6: Automatically generated Software Monitors

SW Monitor	Externalized RV	Formalism used for RV	Programming Language
MaC [ea04]	Yes	No	Java
RV-Statemate [AT04]	Yes	Assertion Language (including SM information)	C/Assertion Language
RV-Distributed [BLS]	Yes	Linear Temporal Logic (LTL)	C++

the solution is for distributed systems and they don't consider how the component is designed. This approach for dynamically analysing distributed embedded systems could be used in the testing phase or even in operation for detecting failures as well as identifying their causes. The approach is based upon monitoring safe properties, specified in a language that allows to express dynamic system properties. For such specifications, monitoring components are generated automatically to detect violations of software components.

Table 3.6 summarizes the section. We can conclude that in this case, all the automatically generated runtime verification systems are externalized solutions and one of them ([AT04]) has the ability to check the correctness of the checked systems in behavioral model terms.

3.2.5 Runtime Software Adaptation and Enforcement

As explained in [GAM17], there are two main dynamic software adaptation approaches using runtime models of the software: planned and unplanned adaptation. Planned adaptation is proactive in which manual or automated decisions are made to dynamically change the software system at runtime. Unplanned adaptation is triggered by unexpected events or when a fault is detected at runtime and reactive decisions are needed to dynamically adapt the system to avoid system failures.

BIP (Behaviour, Interaction and Priority) [FJ17] [EHFJ18] framework aims the runtime enforcement of specifications at component-based systems. They use runtime enforcement to complement model repair and this runtime enforcement targets correctness at operation. At runtime, the monitor consumes information from the execution and modifies it whenever necessary. In order to reach this target, the solution needs to instrument the code. It provides a language and a theory for incremental composition of heterogeneous components, ensuring correctness-by-construction for essential system properties such as mutual exclusion, deadlock freedom and progress.

Table 3.7: Runtime Software Adaptation solutions

SW Monitor	Code Instrumentation	Model based formalism for RV	Programming Language
BIP [FJ17] [EHFJ18]	Yes	No	C++
MoP based solution [CF16]	Yes	No	Erlang
MoCo [Der15]	Mediator	Yes	Java
ReMinds [VRG+16]	Probe	No	Java based technology

Cassar and Francalanza presented in [CF16] a Monitor-Oriented Programming solution and that aims runtime adaptation. They observe the behaviour of the controllers and adapt them when something wrong is detected. Their solution is not based on Model-Driven Development approach and it is not a models@runtime solution. They use an Aspect-Oriented Programming framework to instrument injections at specific points of interest in the code.

There is another solution called MoCo [Der15] that also aims runtime adaptation. In this case, it is a non-redundant, reusable and executable combination of logically related models and code in an integrated form where both parts are stored together in one component. They need a "mediator", an interpreter to link both parts and the first reference implementation did not address embedded systems. They designed a very generic approach, but neither automatic code generation nor how to transform the solution to different technological solutions was addressed at this point. In [DGEE16], they describe a methodology for performing their approach that will be valid for any technology.

In [VRG+16] Vierhauser et al. present a flexible runtime monitoring framework adaptable to different system architectures and technologies called ReMinds. They use Probe [MSS93] as a component to extract or intercept arbitrary information from the monitored systems. They instrument the monitored system by Probe.

Regarding Runtime Enforcement (RE), there are specific works that address generating enforcement monitors such as [FMFR11] and [CFAI18]. The first one, define notions and properties to be used in these monitors whereas the second presents a theoretical foundation of how to implement runtime enforcement.

Table 3.7 summarizes the section. After analyzing the summarized information, we can conclude that all solutions need to instrument the source code of software components to be monitored and only [Der15] works with model terms information to perform the runtime adaptation.

3.3 Critical analysis of the State of the Art

This section critically analyses the current state of the art in Runtime Verification and Adaptation based on software components' reflection ability in model element terms, specifically UML-SM based software components, which aims at providing potential research opportunities. In addition, the Models@Run.Time approach has been analyzed.

In the scope of Models@run.time, different solutions to trace the UML-SMs in order to obtain information about the monitored software components at runtime have been analyzed. Most of the approaches focus on instrumenting the code and not the model (e.g., [MWPB16, DGJ+16, ALG15]).

Reflective UML State Machine are one way to have internal status information in model terms at runtime due to their introspection ability. In this field, there are different frameworks (e.g., [BHB09, EMS+08]) that provide a runtime state-based component model but most of them are not oriented to systems with resource limitations. In addition, they have not defined a MDE approach that enables designers to decide which states of the model will be observed at runtime and neither are oriented to adapt their behaviour once an error or an unexpected circumstance is detected. Moreover, these solutions usually depend on a specific programming language and the transformation rules are specific for one kind of solutions.

To close the MDE approach, we have to add that there are different tools that generate the code automatically (shown in Table 3.3). After analyzing the information of this table, we concluded that there is a lack of solutions able to provide internal status information in model terms at runtime.

Regarding Runtime Verification (RV) solutions, we focused on software based runtime verification solutions. After analyzing different software monitor solutions, we concluded that most of them does not: (1) address resource limited systems and (2) use UML-SM model based formalism for RV. There are two solutions, CoMA and RV-Monitor that use similar formalisms. Nevertheless, CoMA needs an extra support such a simulator to work and RV-Monitor solution is not an externalized RV solution: it integrates the RV part in the controller. It is not a modular solution.

As regards to the way of performing the runtime verification, most of these tools perform the runtime checking at the same abstraction level, that is, only by checking system level properties could system misbehavior be detected (e.g., [ea11b, ea16, ea02, BGL93, DJC94, AS98]). Nevertheless, component or class level properties can give valuable information in detecting system level problems and undesired emergent behaviour.

We have identified two specific solutions, LuMiNous [PW16] and RV-Async [AF17], that are able to check at system level using other abstraction level information. Nevertheless, none of them are using internal information in model terms to check the correctness of the components or systems at runtime: they are not models@runtime approaches.

Some of the analyzed Runtime Verification (RV) tools are generated automatically. The automatic generation of these modules is very important but as long as we know, current solutions are not addressing UML-SM based formalism to be checked at runtime.

To close de section, different solution have been analyzed in the area of Runtime Adaptation (RA) and Runtime Enforcement (RE). Most of the analyzed solutions do not use model element terms or a model based approach to perform the adaptation. We have identified one work that considers the model@runtime approach (MoCo [Der15]) but it needs a mediator to link the code of the software controllers with the models at runtime. In addition, its implementations is not oriented for resource-limited solutions.

Chapter 4

Theoretical Framework

Contents

In this chapter we give a theoretical overview of the dissertation. Specifically, we define four research objectives (Section 4.1) together with the hypotheses (Section 4.2). Furthermore, we give an overall overview of the theoretical framework proposed for generating UML state machine-based software components with reflection and introspection abilities (Section 4.3). In addition, we explain the employed case studies that were used to validate the effectiveness of the solutions proposed in the theoretical framework (Section 4.4). Lastly (Section 4.5), we explain which case studies where employed for the validation of each of the contributions (shown in subsection 1.3).

4.1 Research Objectives

As we have mentioned and concluded in Section 3.3, low cost mechanisms able to increase the safe behaviour of resource-limited CPSs is not a topic that is solved by current solutions.

Having analyzed the existent solutions in the area of the aforementioned topic, we have identified the desirable characteristics of the software components that control these types of systems:

- provide internal status information in model element terms at runtime without instrumenting the code
- provide adaptation ability at runtime
- automatic generation following a MDE approach
- address resource limited systems

As far as we know, there are not solutions that address these four characteristics.

In order to enhance the safe response of the software components and systems at runtime, as mentioned before, we need a Runtime Verification (RV) system to check their correctness and start adaptation processes if necessary. After analyzing the features of the existent RV and RA systems, we decided that our approach will address *software* monitoring systems and we have defined which are the desirable characteristics for our solution:

- independent to the software controller: externalized solution. Verification and Adaptation process will be independent to the software components or system. The code of the software components do not need to be modified,
- use of model-based formalism to perform the RV,

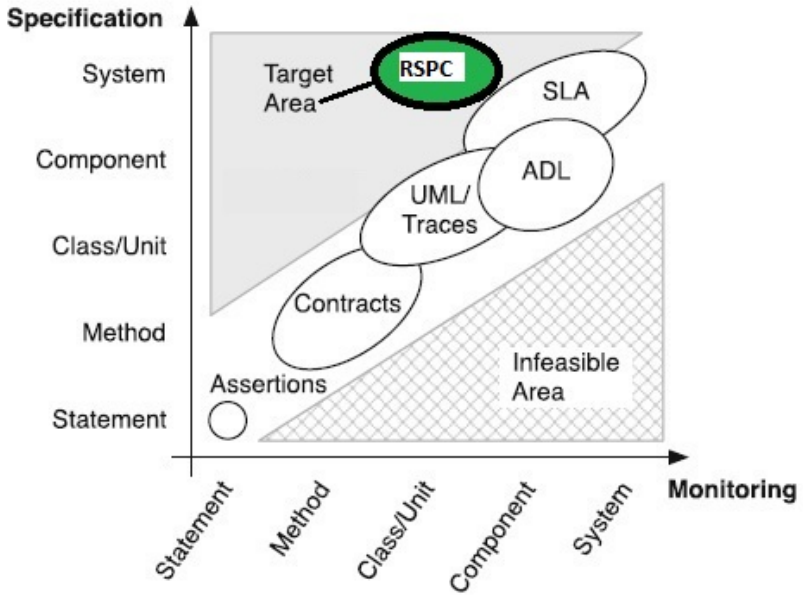


Figure 4.1: Abstraction level of specifications vs. runtime monitoring abstraction levels based on [PW16]

- use of software component level model information to check different level/abstraction requirements: software component level and system level requirements will be considered. Our target area is the one depicted in the fig. 4.1,

- automatic generation of the RV system.

As far as we know, there are not solutions that address these four characteristics, nor reflective software components automatic generation frameworks.

The presented solution, when performing the Runtime Verification (RV), will check the safe requirements and safe properties at system or software component level. We understand these safe requirements and safe properties as:

- Safe Requirements Specification (SRS) is a specification containing all the requirements to ensure the correctness and safe behaviour of the software component and software systems.

- Safe Properties (SP) assert that the system always stays within some allowed region. The properties asserting that observed behavior of the system always stays within some allowed set of finite behaviors, in which nothing "bad" happens, have a

special interest. For example, we may want to assert that every message received was previously sent.

- State-based Safe Properties (SBSP): a safe property that specifies properties related to the internal behavior of the software components that are part of the system in terms of their UML-SM model.

All things considered, this is the main goal defined in this research work:

> To provide a set of tools and methods for the enhancement of safe behaviour on software components and systems. In order to reach this goal:
> - define a model based methodology following the models@runtime approach. Thus, models used at design and development phases will be maintained at runtime.
> - Runtime Verification and Adaptation techniques based on model terms will be used to enhance safe behaviour of software components and systems.

This objective can be divided into the following sub-objectives:

- *Objective 1: Develop a methodology which permits the automatic generation of UML State Machines based software components with runtime introspection and reflection abilities in model elements' terms.* The aforementioned runtime introspection and adaptation abilities are added automatically to the software component and it does not require the developer to make any extra effort.Thus, one expert will be concentrated designing the behaviour and the logic of the control while the other has to consider only the system's and software components' safe requirements

- *Objective 2: Develop and evaluate a tool for automatic generation of UML State Machines based software components with runtime introspection and reflection abilities on a cost-effective manner.* The information they will provide at runtime will be in model elements' terms (models@runtime approach).

- *Objective 3: Develop and evaluate an externalized module able to monitor and verify the status of software components at runtime at different levels (i.e., software components level safe properties and system level safe properties) based on model elements' terms information.*

- *Objective 4: Develop and evaluate an externalized module able to adapt the behaviour of the monitored software components in the event that a component or system level error is detected at runtime.* The adaptation performed at runtime will be based on adapting the model of the software component involved in the process.

Each objective defined in this section is directly linked with the technical contributions we defined in Section 1.3 which are:

- A methodology supported by a framework, REflective State-Machines based observable software COmponents (RESCO), that is able to generate software components modeled by Unified Modeling Language - State Machine (UML-SM) that provide their internal status information in model elements terms at runtime.

- RESCO framework: Automatic generation of software components with internal status information observation ability in UML-SM model terms (current state, event, next state,...).

- Runtime Verification. An external monitor and verification system is used to check the internal status of the UML- SM based software components in model terms before a transition in their state, and therefore a change in the output signal, is performed.

- Runtime Adapter. This module has been based on a previous work [GS02]: the adaptation is triggered by unexpected events or when a fault is detected at runtime.

4.2 Research Hypotheses

Based on the objectives defined in this research work, the following research hypotheses have been defined:

- *Hypothesis 1:* The use of Unified Modeling Language (UML) State Machine based software components with introspection and reflection ability in model terms helps the early detection of errors in software systems. This hypothesis corresponds to research objective 1 and 2.

- *Hypothesis 2:* A runtime verification system that use software components' internal information in model element terms permits to enhance the safe behaviour of systems composed of these software components and of the software components themselves. This hypothesis corresponds to research objective 3.

- *Hypothesis 3:* Runtime adaptation permits to increase the availability and to enhance the safe behaviour of software systems. This hypothesis corresponds to research objective 4.

4.3 Overview of the Theoretical Framework

Figure 4.2 depicts the overall overview of the developed methods for generating UML State Machines based software components and the externalized Runtime Verification and Adaptation system.

We designed a model-driven approach and framework to generate software components (namely, RESCO framework), which are able both to provide their internal information in model terms at runtime and adapt their behaviour automatically when an error or an unexpected situation is detected. The aforementioned runtime introspection and adaptation abilities are added automatically to the software component and it does not require the developer to make any extra effort.

Our approach is based on Unified Modeling Language - State Machine (UML-SM) and the software components are generated automatically from the UML-SM model defined at the design phase. In addition, the models used at design time are kept at runtime. Thus, it is possible to perform Runtime Verification using the information of model elements (current state, event, next state,...) of the UML-SM model of the software component(s) under study. This enables us to use a common language to design and verify software components and systems at runtime.

We can use software component level information in model terms to check different level safe properties at runtime: we can define system level safe properties using the model elements of the different software components the system is composed of; we can define safe properties in model terms for each of the software components; or we can also combine both approaches. In any of the cases, the solution enables Runtime Verification (RV) systems to detect errors before the software component or system reaches a failure condition.

Chapter 5 presents the RESCO (1) methodology and (2) framework developed in the thesis. These two contributions respond to the first two objectives defined in the research work. Then, Chapter 6 is focused on the Runtime Verification & Adaptation aspects but in this case the solution presented is used at software component level. After that, in Chapter 7 same research topics (Runtime Verificaiton & Adaptation) are discussed and presented but in this case the solution is for software systems. In these two last Chapters, objectives three and four are addressed.

In table 4.1 the correspondence between Technical Contributions, Objectives and Chapters of the document is shown.

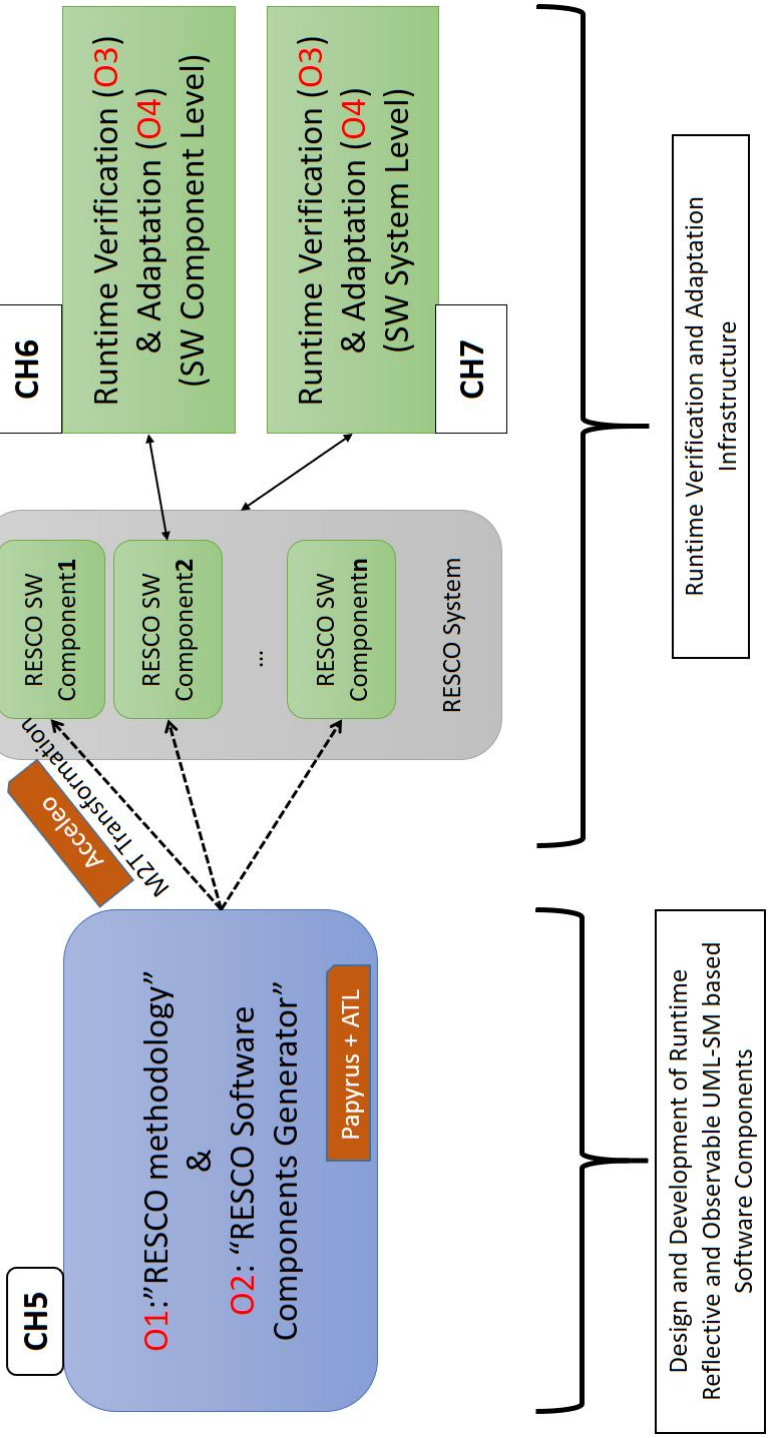


Figure 4.2: Overview of each contribution of the dissertation, their dependencies and structure of the thesis

Table 4.1: Correspondence between Technical contributions, Objectives and Chapters of the document

Technical Contributions	Objectives	Chapters
RESCO Methodology	1	5
RESCO framework	2	5
Runtime Verification module	3	6,7
Runtime Adaptation module	4	6,7

The scope of this dissertation is to advance the practice of enhancing the safe behaviour of software components and systems at runtime following a models@runtime approach and using runtime verification and adaptation techniques.

After analyzing and having identified the technological gaps, our research area has been limited to generate software components that provide internal status information in model terms at runtime and will be working in resource limited CPSs. In addition, as UML-SM is a commonly used formalism to describe CPSs' behaviour in industry, those software components will be modelled by UML-SMs. Thus, runtime model information will be based on this formalism.

In order to answer the identified needs and challenges, a methodology called REflective State-Machines based observable software COmponents (RESCO) **(O1)** has been defined in this work. The aim of this methodology is to define a process to generate UML-SM based software components that are able to provide their internal status information in model terms (UML-SM) at runtime. In order to maintain the model at runtime, the software components need to have introspection and reflection ability.

Using this methodology, in the first step the developer generates REflective State-Machines based observable software COmponents (RESCO) State Machine models that are platform independent and adds information about which states will be available to send internal information at runtime, enabling introspection at runtime. All these characteristics will be used at runtime to check the correctness of the final solution in order to enhance its safe behaviour.

Once the behaviour of the controllers is defined and RESCO State Machine models generated, the final software component is generated automatically **(O2)**: RESCO Software Component. Thus, the identified gap for automatic reflective state machines code generation for different types of system (including the resource limited ones) is filled. In addition, the approach aims to be independent of the final platform in which it is going to be deployed.

The solution is supported by existing tools such as Papyrus [ecl18] (behaviour modeled by UML-SM), ATL [ATL18] (Model To Model (M2M) transformation) and

Acceleo [A16] (Model To Text (M2T) transformation). In the last step of the methodology, a Model To Text (M2T) transformation is performed and this step will be specific for each programming language. In our solution, we implemented a transformation to C++ programming language and the C++ REflective State-Machines based observable software COmponents (CRESCO) were generated. Thus, our solution is developed for resource limited systems.

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.

Runtime Verification (RV) [AF17] is a lightweight post-deployment verification technique that uses monitors or externalized runtime checkers/adapters to incrementally analyse the behaviour of the running system (exhibited as a sequence of trace events) up to the current execution point, in order to determine whether a correctness specification is satisfied or violated. Since correct traces will be finite and predefined in the runtime verification system, when the received trace is not defined as a correct one, the runtime verification system indicates that a trace-violation has been detected.

Runtime Verification can be performed in different ways and one of them could be performed using the information of model elements (current state, event, next state,...) of the UML-SM model of the software component under study. There are not mature solutions that are able to perform runtime verification by using model element terms and this area has been identified as a research topic. When using models@runtime, the traces that are observed during runtime verification are information in model terms.

Moreover, working with models allows us to perform the instrumentation at model level. Software components' instrumentation is needed for runtime checking and to receive traces/information from the monitored software component. Traditionally this instrumentation is made at source code level: this technique is used to modify source code to insert appropriate code (usually conditionally compiled so you can turn it off). Programmers implement instrumentation in the form of code instructions that monitor different aspects of a system. On the other hand, model instrumentation specifies which elements of the model will be monitored at runtime and it automates the source code instrumentation. Our approach follows the latter option.

One of the objectives defined for the thesis was to provide a runtime verification system that was able to use software components' internal status information in model elements terms **(O3)**. In this topic we identified different solutions that were able to detect and monitor at different levels: statement, class, component, system, etc. On the one hand, most of the identified solutions were only able to monitor and detect the errors at the same abstraction level. On the other hand, most of them were not able to

make runtime verification using information in model elements terms.

In our approach, we have developed a solution that consist of monitoring at software component level in model elements terms and could be used to detect system or software component level errors. Our approach considers global monitors for checking system level safe properties and local monitors for checking software component's level safe properties (a) and (b) options of the figure 3.2.

In both options, the runtime verification system is an externalized system. Thus, a communication system to send the internal information at runtime and then receive the response is needed. In our concrete solution, it was not a research topic and we just used a middleware called Internet Communications Engine (ICE)[1] to implement the solution. Therefore, the communication between the software components and the runtime monitoring, verification and adaptation system is based on the aforementioned middleware.

The last research topic was runtime adaptation. Once the error is detected at runtime, a solution is needed to avoid system failures. Runtime adaptation is one way to solve this problem. Nonetheless, a gap in RA systems that address model based adaptation processes was identified **(O4)**. Our solution aims to fill this gap suggesting a runtime adaptation based on UML-SMs adaptation at runtime.

All things considered, the overall solution to enhance the safe behaviour of software components and systems, an externalized Runtime Verification and Runtime Adaptation system has been developed. As the UML-SM models used by the solution will be verified in the development process using techniques such as model checking, some types of faults will not be present at runtime. Therefore, errors to be detected by our solutions are mainly random software and hardware errors as well as remaining software errors and unanticipated environmental errors. Summarizing, we can say that 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 [GT05],
- residual faults not detected when testing,
- unanticipated faults that were not considered in the design and development phase.

The overall solution presented in this work provides the following elements to generate reflective software components and detect and recover from errors at runtime:

[1] Internet Communications Engine (ICE) [Zer19] belongs to the object-oriented middleware category. It is conceptually very similar to CORBA [DSE+11], but offers a much smaller and more consistent API, lighter implementations, advanced services, and good performance [MAJJ08]. The product supports C++ and Java, and runs on Linux and Windows.

- RESCO Software Components (O2): Software components with reflection/observation ability in UML-SM model terms (current state, event, next state,...). The software components are generated by the RESCO framework [IEES17] which is one part of the developed solution and presented in the Chapter 5. This framework is able to generate software components with introspection and reflection ability. Moreover, in the RESCO framework, logic of the software components and their reflection/observability ability are orthogonal. The developer focuses on the design of the functional behaviour of the software component, whereas the reflection/-observability ability in model terms is added orthogonally and automatically.

- Runtime Monitoring and Verification System (O3): An external checker is used to check the internal status of the software components and systems in model terms before the transition in their state, and therefore output signal, is performed. This allows us to detect faults before the failure happens, increasing the resilience against the faults enumerated above.

- Runtime Adaptation System (O4): An externalized runtime adaptation is used. This adaptation is triggered by unexpected events or when a fault is detected at runtime. This allows us to protect against unsafe conditions ensuring that the software component performs safe actions. In particular, it supports the "active monitor" or "safety bag" concept described in [IEC10].

Our approach has the following characteristics: (1) specification of what we want observe at runtime can be tuned in the models at design phase. Thus, we avoid instrumenting the source code and we add the required information and infrastructure that enables introspection and adaptation at runtime; (2) code generation takes this information and generates source code automatically; (3) observed information not only does it consider the outputs of the software components but also monitors their states and events status; (4) systems composed of C++ REflective State-Machines based observable software COmponents (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 adaptation system sends an event to the observed software component and this component changes its operation mode automatically to a previously defined mode (e.g. safe-mode, degraded-mode,...).

4.4 Case Studies

With the objective of validating the proposed methods, 5 synthetic (academic) uses cases and two industrial case studies were developed. The industrial case studies were from different domains (i.e., energy and railway) and complexities. In this document, we are going to consider the following industrial case studies.

4.4.1 Burner Controller

One of the selected case used for evaluation is an industrial software component that controls a micro-generation device: the Burner controller of the Whispergen commercial device [Pra16]. Microgeneration applies to a rather surprising mix of heat and power technologies with a thermal output below 45kWt or an electrical output of 50kWe. It covers electrical generation from wind, solar photovoltaic (PV) and hydro, heat generation from biomass, solar thermal and heat pumps as well as micro Combined Heat and Power (CHP) which produces heat and power from renewable or fossil fuels. Centro Stirling S.Coop develops these type of machines and they collaborate with Mondragon Goi Eskola Politeknikoa S.Coop in the development of such device's SW Control components.

For evaluation purposes, the Burner controller shown in figure 4.3 was implemented. The selected Burner controller's state machine has 10 simple states, 3 composite states, 13 transitions and 13 events. The behavior of the controllers is modeled by UML-SMs.

In case an error is detected, the behaviour of this controller will be adapted to a safe operation-mode. In this case, the safe-mode UML-SM has 7 simple states, 3 composite states, 9 transitions and 9 events. In the figure 4.4 the safe-mode UML-SM is shown.

By means of this use case, we are going to evaluate how RESCO infrastructure is able to automatically generate software components with reflection and introspection ability. The information they will send at runtime will come in terms of model so it will be possible to verify component level behavior at execution time in model terms as done in the development phase.

4.4.2 Train Control and Monitoring System (TCMS)

In the second case study, a door control management performed by a Train Control and Monitoring System (TCMS) was considered. The TCMS is a complex distributed system that controls many subsystems such as the door control, traction system, air conditioning, etc. of a train.

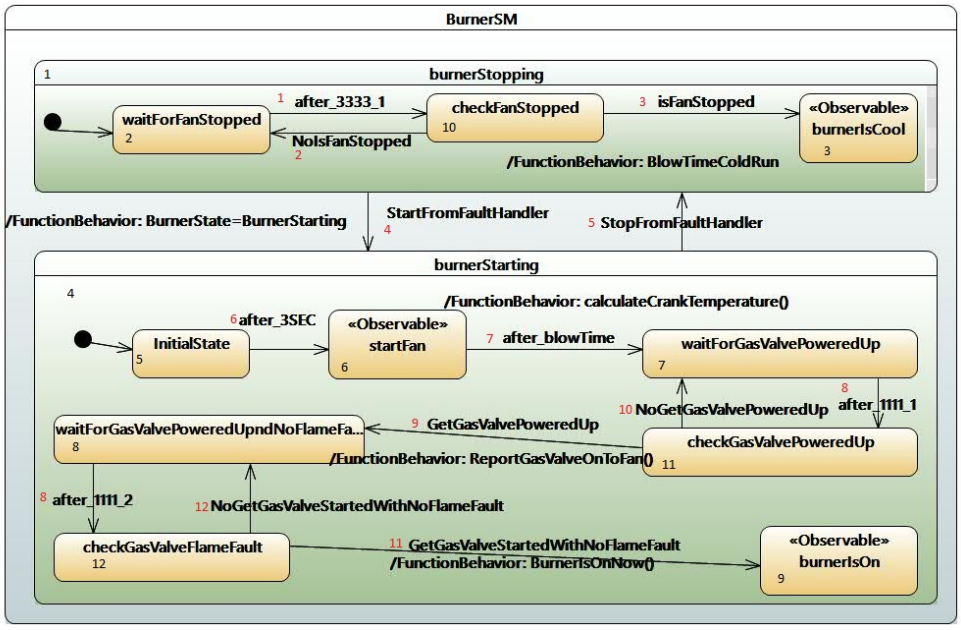


Figure 4.3: The Burner's SM

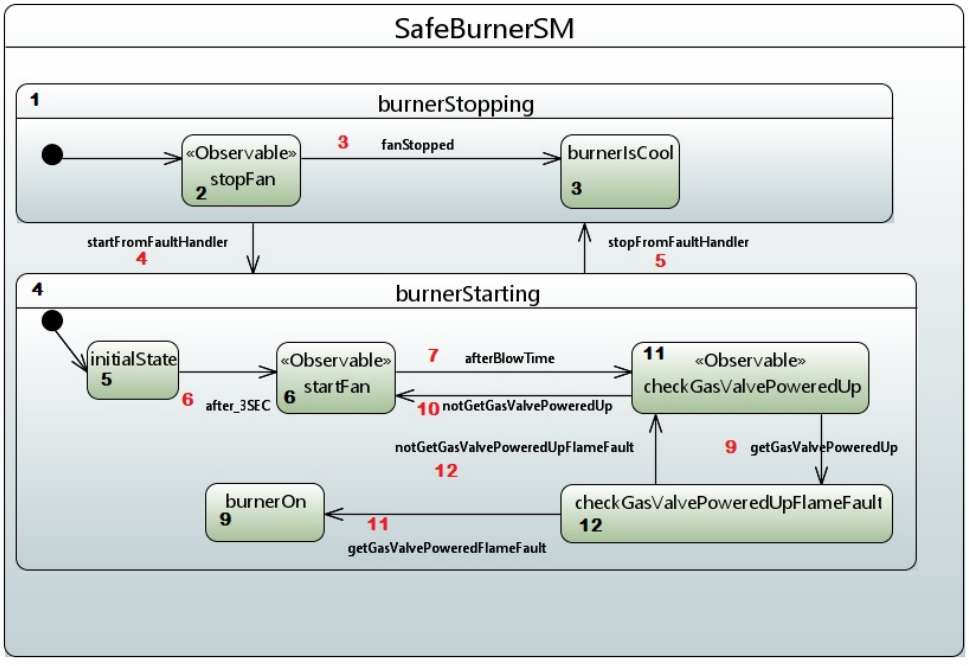


Figure 4.4: The Safe-Mode Burner's SM

The case study concerns a real industrial system where some simplifications were made. Figure 4.5 shows the different subcomponents and their relation with the TCMS.

The train subsystems such as the door control system are safety-critical systems and, therefore, railway standards must be applied during their development. The major standards are the European EN5012x family of railway standards.

In a compositional approach [np15] [Saf13], a safety case would contain the top-level claim about the safety of the overall system, the decomposition into more detailed claims about its constituent subsystems or components, the arguments that show that the components fulfill the safety-related claims that are made about its properties and parts of the component specification may be expressed as component safe properties (SP), i.e. as assumptions on the component's environment and guarantees of properties that the component will satisfy when those assumptions are fulfilled. Some of these SPs will address Safe Requirementss (SRs).

Fig. 4.6 shows the UML-SM of the DoorController, ObstacleDetector and Traction.

System Requirements

The system level requirements concerning the operation of opening and closing of doors are satisfied by the following components:

- *TCMS* component decides whether to enable or disable the doors considering the driver's requests and the train movement. Thus, doors must be enabled before they

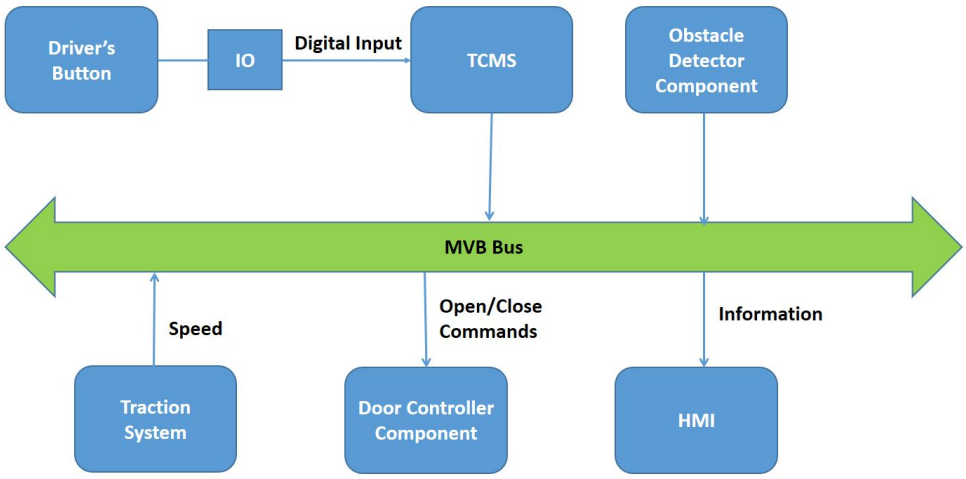


Figure 4.5: The TCMS System and others components.

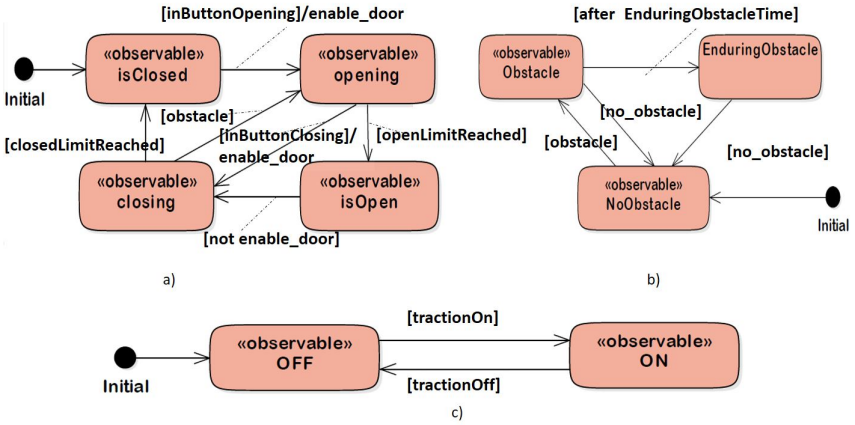


Figure 4.6: UML-SM Diagrams of a) DoorController, b) ObstacleDetector and c) Traction

can be opened;

- *Door* component controls and commands the opening and closing of a door;
- *Traction* component controls and commands the train movement;
- *Obstacle Detection* component manages the obstacle detection in the door.

By means of this use case, we are going to evaluate how RESCO infrastructure is able to automatically generate software components with reflection and introspection ability. The information they will send at runtime will come in terms of model so it will be possible to verify system level state based safe properties at execution time.

4.4.3 Overview of the key characteristics of the case studies

The two case studies correspond to different domains. Furthermore, to ensure the sufficient degree of heterogeneity, the complexity of each case study is different. Table 4.2 summarizes the main characteristics of each case study. *UML-SM* blocks is referred to the number of UML-State Machines in the use case. Column *Complexity* refers to the complexity of the state machines considered in each of the use cases. Considering the works in [Bel13] and [GMP03], to measure the size and complexity of state machines we used the Cyclomatic Number of McCabe (Structural Complexity metric) adapted to state machines.

Safe Properties refers to the number of safe-properties considered in the use case. *Adapt* refers to the total number of Safe Mode UML-State Machines considered in case that an error is detected at runtime and an adaptation process is activated. In addition,

in some experiments we used software component level Runtime Verification (RV) module and in others a software system level RV was used. Column *RV Level* defines this characteristic. It is important to highlight, that for some evaluations, different versions of the case studies could have been employed, and thus, these characteristics could vary.

Table 4.2: Main characteristics of each case study

Case Study	**UML-SMs**	**Complexity (McCabe)**	**Safe Properties**	**Adapt**	**RV Level**
Burner	1	3	21	1	Component
TCMS	3	Door:4 Traction:2 Obstacle:4	5	0	System

4.5 Case studies employed in each contribution

As mentioned before, each of the contributions defined in section 1.3 was independently assessed by means of an empirical evaluation. For each evaluation, one or more case studies were employed.

Table 4.3 shows which case study was employed in each of the contributions. Specifically, for the Reflective UML-State Machines based software components generation, both use cases were used. For measuring the enhancement of the safe behaviour at software component's level, the Burner controller use case was used. In addition, we evaluated the ability of adaptation at runtime using the Burner's use case. The Train Controller use case was used to evaluate the ability to check software systems' safe properties at runtime. Thus, the safe behaviour at system level is enhanced.

Table 4.3: Case studies used in each of the contributions

Contributions	**Burner**	**TCMS**
1.Methodology for generating reflective software component	X	X
2.RESCO framework	X	X
3.Runtime Verification:		
3.1. at software component level	X	
3.2. at system level		X
4.Runtime Adaptation	X	

Part II

Software Components Generation

Chapter 5

Reflective UML-SM Software Components Generation

Contents

5.1 Introduction

In this chapter, we present a models@run.time approach to automatically generate UML-SM based code with runtime introspection, verification and adaptation ability. We highlight two main contributions in this chapter: (1) a Model-Driven Development (MDD) based methodology to design and develop UML-SM based software components with introspection and reflection ability at runtime and (2) a framework for the automatic generation of those software components. As a result, we have defined and developed the REflective State-Machines based observable software COmponents (RESCO) methodology and framework. These contributions provide the following benefits:

1. Runtime monitoring, verification and adaptation ability which are based on information provided by the software components in terms of their model elements at runtime.

2. The software engineer is not involved in instrumenting the code and thus, can focus exclusively on modelling the behaviour of the software components by UML-SMs. Additional infrastructure for having introspection and adaptation ability at runtime is automatically generated.

The chapter is structured as follows: Section 5.2 gives a general overview of the proposed methodology for the automatic generation of UML-SM based software components. The Reflective UML-SM based software components generator approach is explained in detail in Section 5.3. Section 5.4 presents a complete evaluation of the proposed approach. Finally, conclusions are outlined in Section 5.5.

To evaluate the proposed framework, we selected five synthetic (academic) use cases and two different industrial use cases. Results show that the software components generated by the presented solution provide reflection and introspection at runtime. Thanks to these abilities at runtime, the software components are able to provide runtime information and adapt automatically from their normal-mode behaviour to a safe-mode behaviour which was defined to be used in erroneous or unexpected situations at runtime. Therefore, as it is demonstrated in Chapter 6 and in Chapter 7, it is possible to enhance the safe behaviour of the systems consisting of these software components.

5.2 Overview of the Approach

The overview of the approach considered in this study to automatically generate UML-SM based software components with introspection and reflection ability at runtime is highlighted in Figure 5.1.

It represents the Model-driven workflow that safety and software engineers must consider when using the methodology we propose. In the first step, the software engineer models the behaviour of the software components using the UML-SM formalism. Depending on the final system, they have to model at least two versions of the UML-SM: one that defines the normal behaviour of the software component (normal-mode) and another one or more that define the behaviour of the software component to be adapted when an error is detected (safe-mode).

Once the UML-SMs are defined, and based on the initial requirements, the states that may be related to safe requirements are annotated to be instrumented automatically (model instrumentation). Thus, the annotated states will be observed at runtime. In addition, other states that are not involved in the safe requirements could be also be annotated. It is a decision to be made by the safety engineer.

In the second step, those annotated UML-SMs are transformed automatically (Model To Model (M2M) transformation) and the RESCO-State Machine (RESCO-SM) models are generated.

In the third and last step, depending on the final system where the software components will be deployed, the Model To Text (M2T) transformations will be performed to different programming languages. For instance, if the system addresses resource-limited systems, we can transform the model to C++.

Figure 5.2 shows the SPEM process of the RESCO methodology. In this figure we have included the specific tools that we have used in our developments. We have used Papyrus modeling tool [Pap19] in the first step, ATL [ATL18] for performing Model To Model (M2M) transformations in the third step and Acceleo [A16] to specify Model To Text (M2T) transformation in the last step (C++ transformation).

5.3 RESCO: Automatic UML-SM based Reflective Software Components Generator

This section presents the model-driven approach for generating runtime observable and adaptable UML-SMs components following a models@runtime approach. In the next subsections we will present the details of the process of generating reflexive software components that provide information in terms of UML-SM model elements at runtime

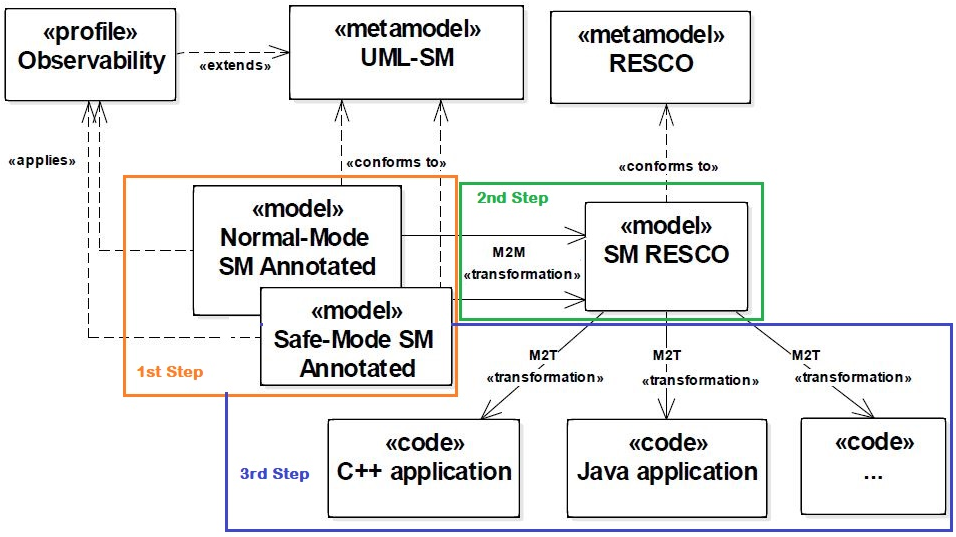


Figure 5.1: Model-driven Workflow

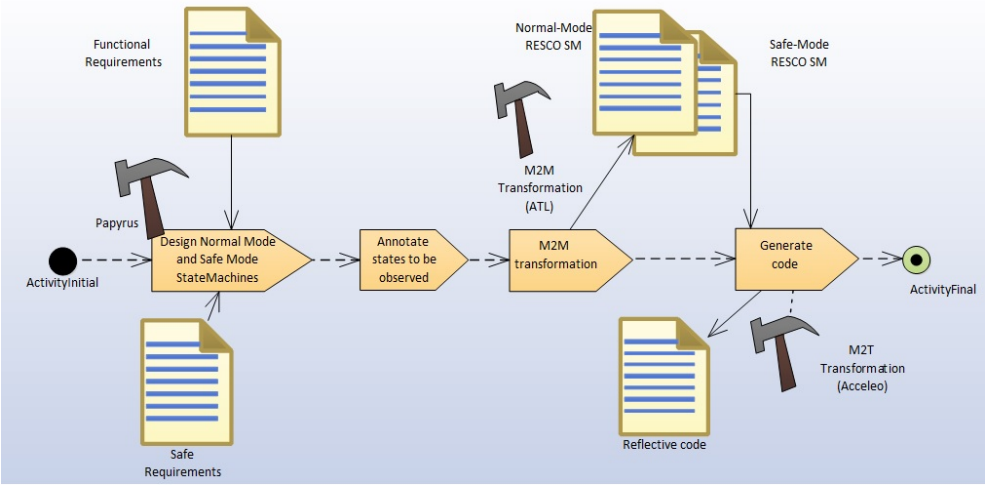


Figure 5.2: SPEM diagram of the RESCO methodology

and allow adapting their behaviour dynamically when an unexpected situation or error is detected.

5.3.1 RESCO Framework Architecture

The solution presented in this work, is based on the RESCO-SMs and first, in this section we are going to show its architecture in order to provide a global vision of the approach.

The architecture is based on two main parts: one in charge of the infrastructure needed during the software component's design and development phase and the other plays its role at runtime.

At design time, the State Machine (the solution's part that defines the behaviour of software components by states, transitions, conditions and events) and Executor (part of the solution that has the actions to be performed by the State Machine in specific situations) are designed and the behaviour of the software controller is implemented. In the next subsection they will be shown with more details.

As for runtime, the main module is the Dispatcher (part of the solution that manages the transitions' triggering when an event/signal arrives). This module follows the UML-SM formalism but in this specific solution, this module adds the introspection and reflection ability. Once an event reaches the dispatcher, it analyzes the current state of the software component and provides the ability to check and decide to execute or not the suggested transition. In figure 5.3 we can see the overall architecture and the interaction between these main modules of the solution.

Thanks to the architecture and design of the solution, we can clearly separate two parts: on the one hand, we have the behaviour and the logic of the software controllers (State Machine and Executor) and on the other hand the part that adds reflection, introspection and adaptation abilities (Dispatcher). These two parts are independent from each other and this is one of the highlighted characteristic of the solution.

5.3.2 RESCO Metamodel

REflective State-Machines based observable software COmponents (RESCO) metamodel is the central artifact of the approach. The goal of the RESCO metamodel is to model components with state machine models@run.time capabilities. When performing the literature review of reflective state machines, we saw that there are certain design patterns which provide the artifacts we need to reach our objective. After considering the different options, we defined two parts to describe the state machines of our solution considering that they will need to be reflective and provide introspection ability at runtime: on the one hand, the part that is responsible for the

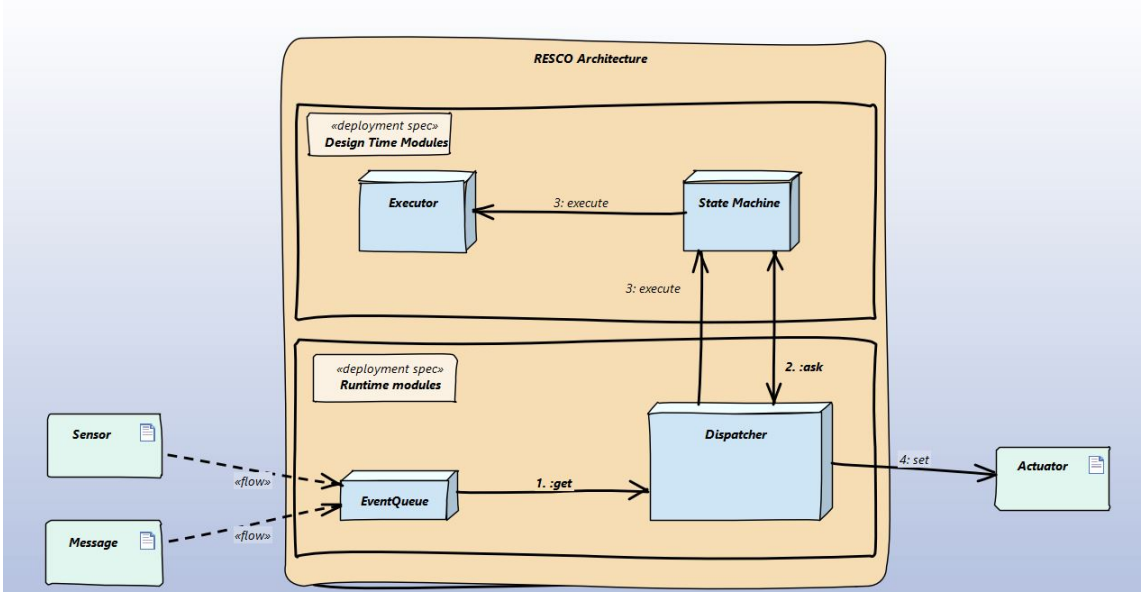


Figure 5.3: Overall Architecture of RESCO framework

actions to be performed (Executor) when the system is in execution and on the other hand, the part that defines the state machine (State Machine).

Let us consider the state machine of figure 5.4 as a guiding example. This state machine is divided in the two parts aforementioned. Design time modules are represented by means of a RESCO model in figure 5.5. Each of the states (s0,s1,s2,s11,s12) of the state machine are represented by hierarchical objects that compounds a tree-like structure (this is the State Machine object). The father state, s0 in this case, is on the top of the structure and the nested states hang directly or indirectly from this one creating a tree-like structure. This tree-like composite object-structure reflects the state structure of the model and each state has the specification of the behavior attached to its different elements. The actions to be performed in each state are defined by means of links to the executor part. Thus, each object has defined their reactions when a specific event is triggered. As an example, we can see that when an EvA is triggered, and the software component is in s2, if condition method02 (n%2==0) is fulfilled it performs the action method03 (n++) and performs a transition to s12 which is defined as target state.

As we can see, having the solution organized in this way makes that, a change in the object structure of the state machine means model modification and vice-versa.

All these things considered, the RESCO metamodel that have been designed in this thesis is composed of two packages: (1) a design package that is used for modeling the application specific part and brings together the two parts (state machine and executor)

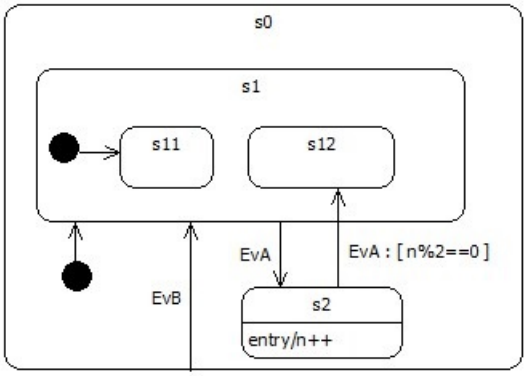


Figure 5.4: State Machine, guiding example

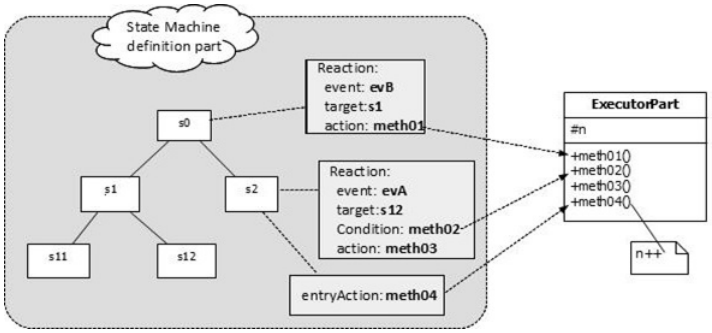


Figure 5.5: Guiding Example: SM transformed into a RESCO model (Design Package part)

defined above and (2) a runtime package that enables a UML-SM based software component to reflect the model it comes from (models@runtime approach).

The application specific part of the model is modeled using the design package and it includes the *StateMachine* and *Executor* concepts defined earlier. The *StateMachine* describes an hierarchical state machine along with a description of the *Reactions* defined in each of the *States*. The *Executor* has the implementation of the *Actions* that need to be triggered and the *Conditions* that need to be evaluated. The *Reactions* have the references to them. The *StateMachine* and the *Executor* are generated automatically from the UML-SM model defined by the designer.

The runtime part, that is generic for all the components and applications, is modelled using the runtime package. It includes generic elements used for providing models@run.time observation capabilities: the *Dispatcher*, the *Observer* and the *EventReceiver*. The runtime elements are used for implementing the execution semantics of state machines. This is explained in the execution model of RESCO in Section 5.3.3.

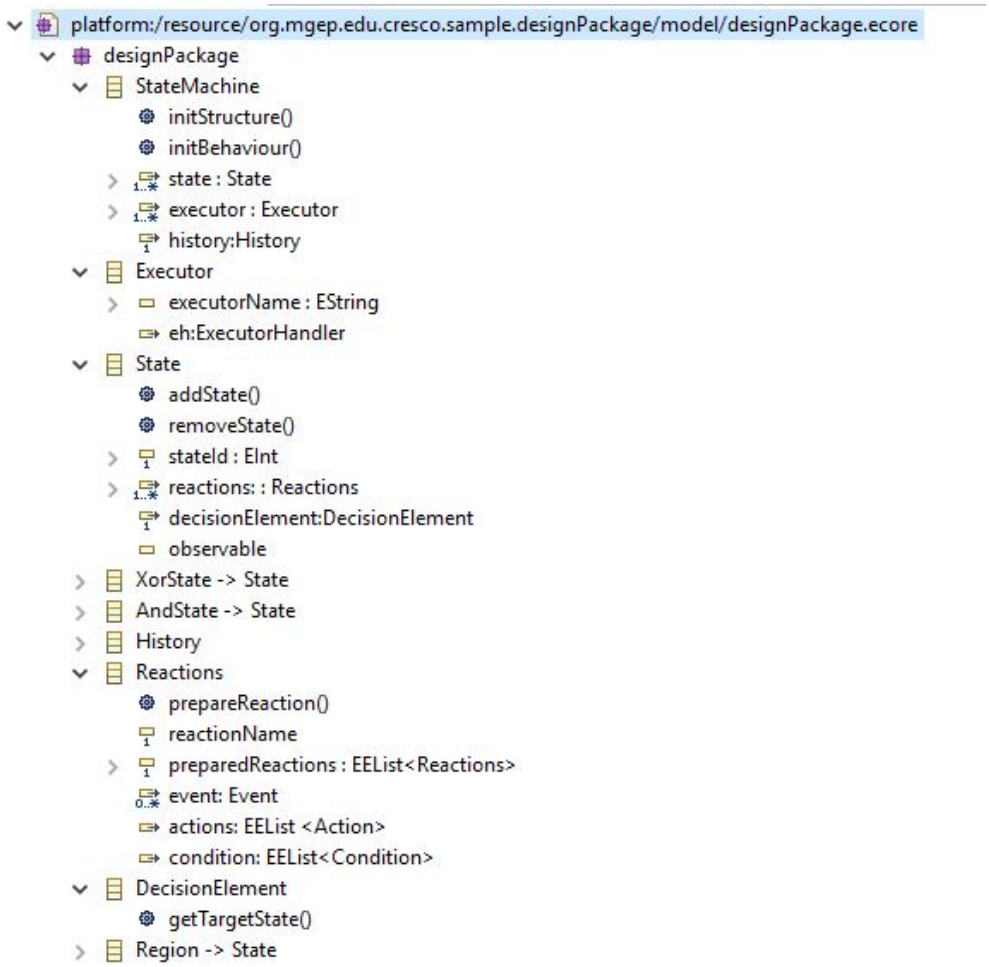


Figure 5.6: RESCO Design Package Metamodel

Using this two packages, an object structure that reflects the model is generated. Thus, it will be possible to adapt the model without recompiling the solution. As aforementioned, this is possible thanks to the architecture and design of the solution. The logical part and the one that provides the introspection, reflection and adaptation abilities are independent.

RESCO Design Package

Figure 5.6 shows the metamodel of the RESCO Design Package.

From a structural point of view, each *StateMachine* will have several *States*. Each *State* of the *StateMachine* could be implemented by *XorState* or *AndState* classes and the latter could have different *Regions*. Each *State* will also contain other *States* and, as particular characteristic of this solution, each *State* has the attribute of *Observable*.

This attribute has a special meaning in the solution. We use this attribute in order to define if a state will be observed at runtime or not. Thus, when it is *Observable*, the *Dispatcher* will provide its runtime status information to an external runtime verifier with the aim to check its correctness at runtime.

Each state may also have *Reactions* that can be triggered when a particular event is received. These *Reactions* can involve a transition to other *States*. *Reactions* have information about the next state to be reached, the actions to be performed and conditions that must be met to trigger them. When the control has to perform a transition, different steps are carried out. One of these is to identify the prepared reactions and calculate the path from the source state to the target state following the same semantic as the one defined in the Precise Semantic of UML State Machines (PSSM) [OMG17].

As mentioned before, the *StateMachine* has links to the *Executor*. The *Executor* has the implementation of the *Conditions* that must be evaluated at some point in time when responding to an *Event*. It also has the implementation of *Actions* attached to different *Reactions*. Thus, this structure reflects the software component's model at runtime.

Having implemented the solution following the tree-like structure that reflects the model of the software component and having linked the action to be preformed to an independent executor, makes it possible to (1) provide internal status information of the software components in model elements terms at runtime and (2) adapt the model of software components at runtime.

RESCO Runtime Package

RESCO Runtime Package represents the infrastructure that enables the solution to have the elements of the model at runtime. This part is generic and does not depend on the behaviour of the software component. Figure 5.7 shows the metamodel of the RESCO Runtime Package.

We can say that the heart of the RESCO's Runtime Package is the *Dispatcher*. The *Dispatcher* upon the reception of an *Event* at runtime, asks about the active *State*s the component is in, which transitions can be triggered, checks if the *Condition*s are met, and finally triggers the list of *Action*s related to a particular transition. This collaboration is illustrated in figure 5.8.

In this way, the *Dispatcher* is able to know which *State*s are active, which transitions need to be triggered, and the target state the component is transiting to. The *State*s that are defined as *Observable* could be monitored at runtime by an *Observer*. This *Observer* is in charge of delivering the information that is known by the *Dis-*

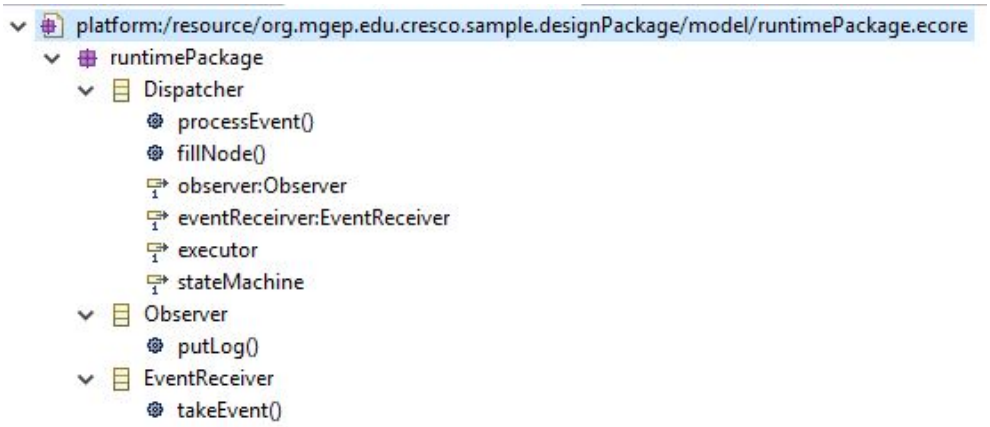


Figure 5.7: RESCO Runtime Package Metamodel

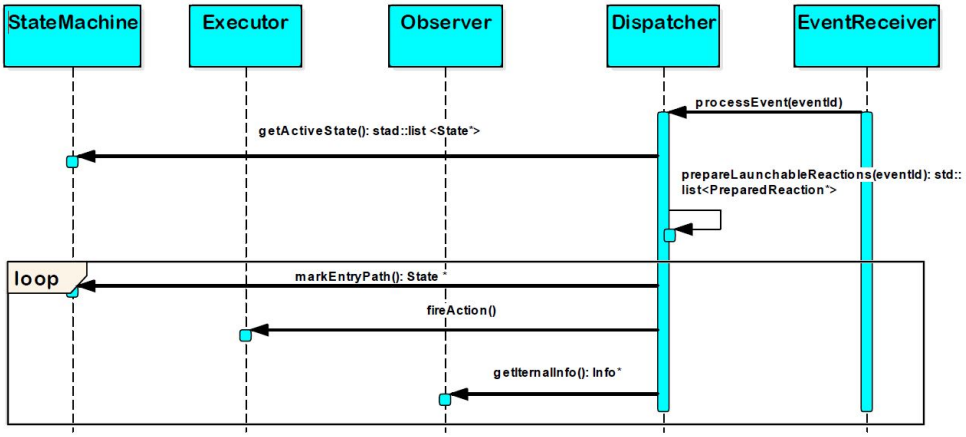


Figure 5.8: RESCO Sequence Diagram

patcher to external monitors and runtime checkers in order to check the correctness of the behaviour of the software component.

The *Dispatcher*, will start its work when a new *Event* is received. The component in charge of the reception of the *Events* is the *EventReceiver*. It has an *Event* queue and notifies the *Dispatcher* each time an *Event* is received.

5.3.3 RESCO Execution model: Algorithms to process events & basis for software component runtime observation

The execution model of RESCO defines the algorithms to process the events. These algorithms are part of the *Dispatcher*, which offers us the ability to interrogate the sofware components at runtime. These execution algorithms are application-independent.

As explained before, the first element receiving the notification of an *Event* is the *Dispatcher*. Next, this determines the *Actions* to be executed once it has evaluated the *Conditions*. Finally, the relevant *Reactions* are triggered. In the event that the *Dispatcher* is in charge of passing arguments to the methods in charge of executing *Actions* or evaluating *Conditions*, we will be mixing the general semantic of the state machine execution with the particular details of specific applications and our aim is to provide a generic framework to be used by any application. As a conclusion, we defined as a design rule not to pass arguments to the methods.

Considering the last statement, these are the main characteristics of the execution algorithms:

- The *Events* are processed following the run-to-completion paradigm and they are stored temporarily in a global repository. The methods that need this additional information provided by *Events* can access this repository.
- The methods in charge of executing *Actions* or evaluating *Conditions* do not have any arguments. This way, there is no need for argument passing and this characteristic makes it possible to implement a generic *Dispatcher*.

If we consider the example shown in figure 5.9, using the RESCO framework we do not need to instrument the code in order to have the internal status of the software component at runtime. We only have to annotate which of the states have to be observed at runtime. Then, the *Dispatcher* will be in charge of interrogating the states that are annotated as observable at runtime.

As we can conclude, the developed *Dispatcher* solution has not dependencies with the logic final use case, it is a generic solution that will be used in all the cases. Thus, we can add that the logic and the reflection, introspection and adaptation ability of the solution are independent.

How to determine the "current" transitions

State machine formalism defines a transition as a family of transitions that is represented as one graphic element. Once the execution is ongoing, the specific transition is determined at runtime. The transition that is triggered at runtime is called the "current" transition.

Consider the state machine of figure 5.9 and the transition triggered by event B (from s21 to s1). When this *Reaction* is triggered, the system has to exit from one of the substates of s21 and from one of the states of region r2. This could only be solved at runtime.

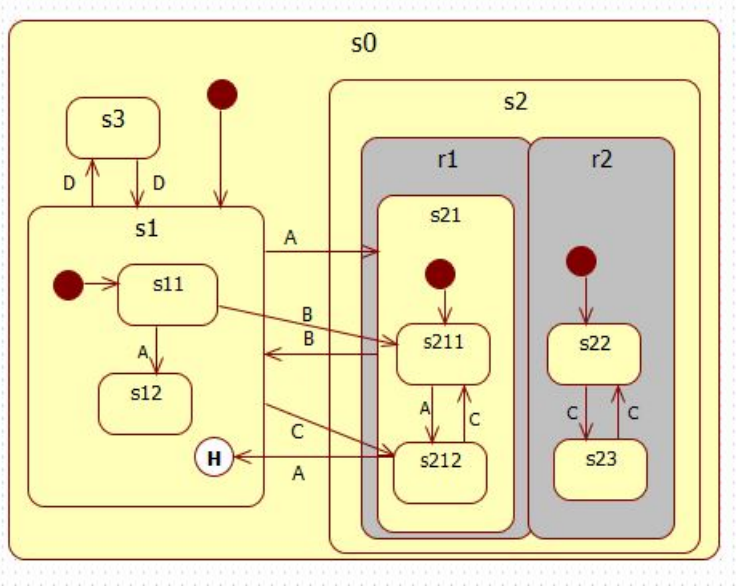


Figure 5.9: State Machine example.

Considering the same transition, we do not know the exact final destination. We only know that the target state will be a substate of s1. If we are using the *History* or *Deep History* pseudostates, the situation becomes worse because we do not know the target of a transition before runtime.

Using the same example we showed in figure 5.9, consider event A that starts in the state s212 and goes to the pseudostate H of s1. The destination of this transition depends on the last active substate of s1. We can not know the destination until runtime.

Once the source and target of a transition are determined, and before the transition is triggered, we need to calculate the output path of states in order to execute the corresponding exit actions and an entry path to go to the target substates.

Finally, before triggering any transition we also have to consider the circumstance where an event has different reaction possibilities. The best option is when the state machine execution-semantic helps to perform this discrimination by selecting only one transition. For example, s2 state has different defined reactions for the A event. The ones that have to prevail are the ones that are defined in the nested states in relation to the superstates that contain them. Nonetheless, there will be more nested cases with contradictory definitions and we can only detect them at runtime. The latter is a bad design characteristic.

In the code fragment of listing 5.1 , we present the defined algorithm to avoid and solve these difficulties.

```
1 Wait for a message (event)
2 Ask the active states if they have reactions to be triggered
3 { 1. Ask about transitions to the event to be processed.
4  This involves questioning the superstates
5  2. If a reaction is found
6  {2.1 Check if its conditions permit triggering of the
7   reaction
8   2.2 In affirmative case, add this reaction to the set of
9   reactions to be triggered}
10 Trigger all the reactions (the order should not influence)}
```

Listing 5.1: Event process algorithm

Transition Triggering Algorithm

The transition triggering algorithm, following the PSSM [Gro17] execution semantic for transition activation, involves executing the exit actions starting from the source state and finishing in a common ancestor of the target state. Additionally, the associated actions have to be executed and, finally, starting from the common ancestor, we have to execute the entry actions up to the target states. In order to trigger a transition, we have to solve the following issues:

- Determine the common ancestor of the source and target states.
- Determine the path of the states that we have to enter up to the target state.

Considering the same example of figure 5.9, we will focus on event C defined in s1 whose target state is s212. Figure 5.10 shows, within the structure that reflects the state machine, the sequence of exit and entry actions to be performed. Following the curved line, it denotes exit actions when going upwards and entry actions when going downwards.

5.3.4 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 was shown by an SPEM process in figure 5.2.

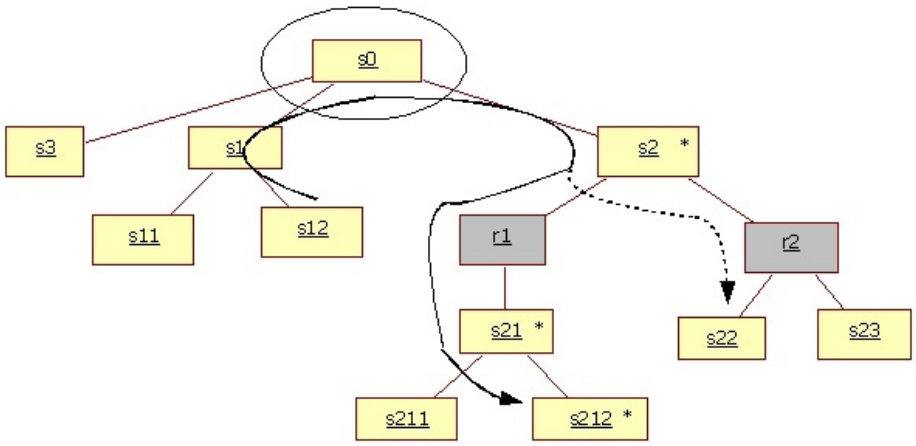


Figure 5.10: Objects that reflect the example state machine's structure.

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 [Pap19] modeling tool. In the presented solution, runtime adaptation is one of the 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.

Different types of systems and operation-modes require different monitoring needs. Although it is possible to obtain information about all the states, events and transitions, due to reduced processing resources, it may be desirable to monitor only a subset of available monitoring information that will be verified. We have defined an observability profile that provides an «Observable» stereotype to select at design level those parts of the controllers which need to be observed. The designer, as a substep in this initial step, annotates which of the states of the UML-SM models will be «Observable».

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 [BHD17], our approach uses M2M transformation techniques for creating an instrumented version of the user-defined model supporting introspection, checking and adapting activities at runtime. This support is added by adding reflection and adaptation capability to the model by the execution algorithms in the (2) runtime package and the Dispatcher of the same package. 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 5.11 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 5.11. 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 an 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 component, the dispatcher analyzes the current

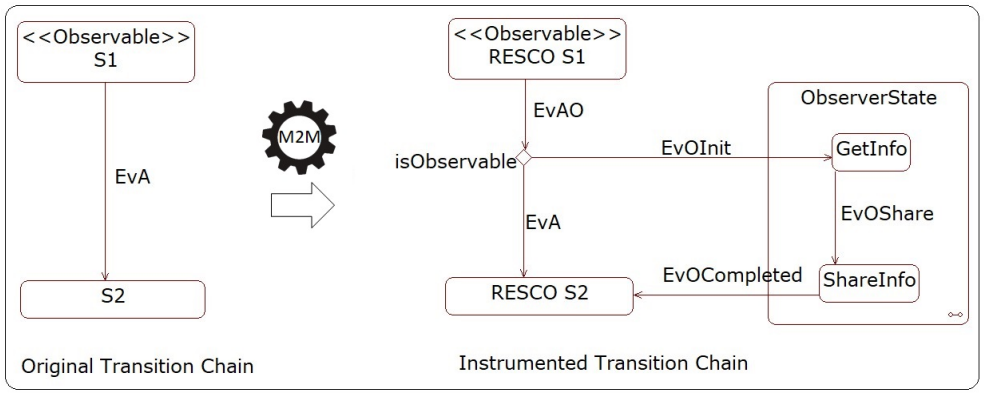


Figure 5.11: Model Instrumentation: Transformation Rule of the runtime package of RESCO metamodel. Thanks to the (2) runtime package, the same ObserverState State object is used in all the transitions.

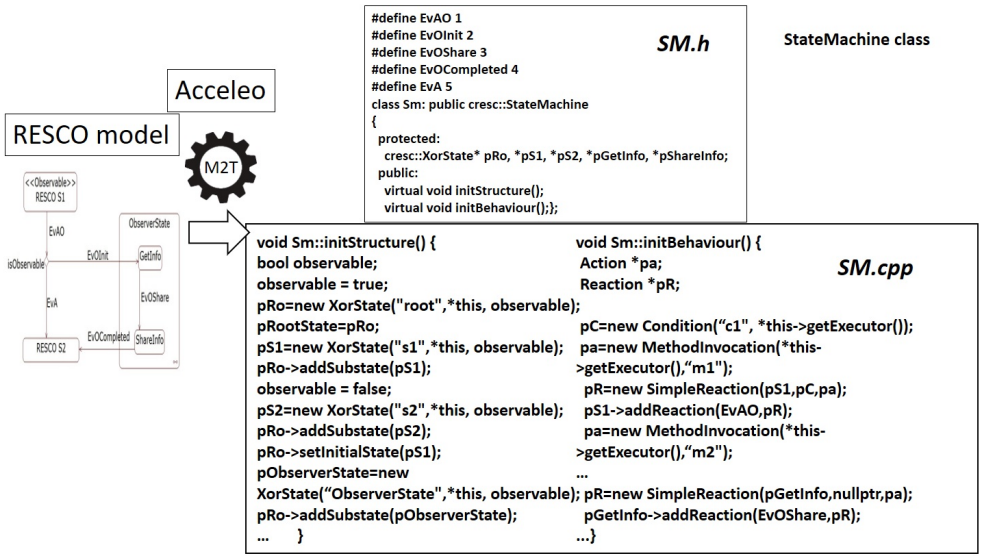


Figure 5.12: CRESCO: RESCO State Machine M2T Transformation to C++ code

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.

3rd step: C++ reflective UML-SM based software components generation (CRESCO)

In this section we will present the concrete implementation of 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 [A16] tool, and (2) an implementation in C++ of the runtime infrastructure. In figure 5.12 an excerpt of the result of the State Machine M2T transformation is shown.

Regarding the executor part, in figure 5.13 an example of the result of the M2T transformation is shown. The *Executor* has the implementation of the conditions that must be evaluated at some point in time when responding to an event and also the implementation of actions attached to different reactions.

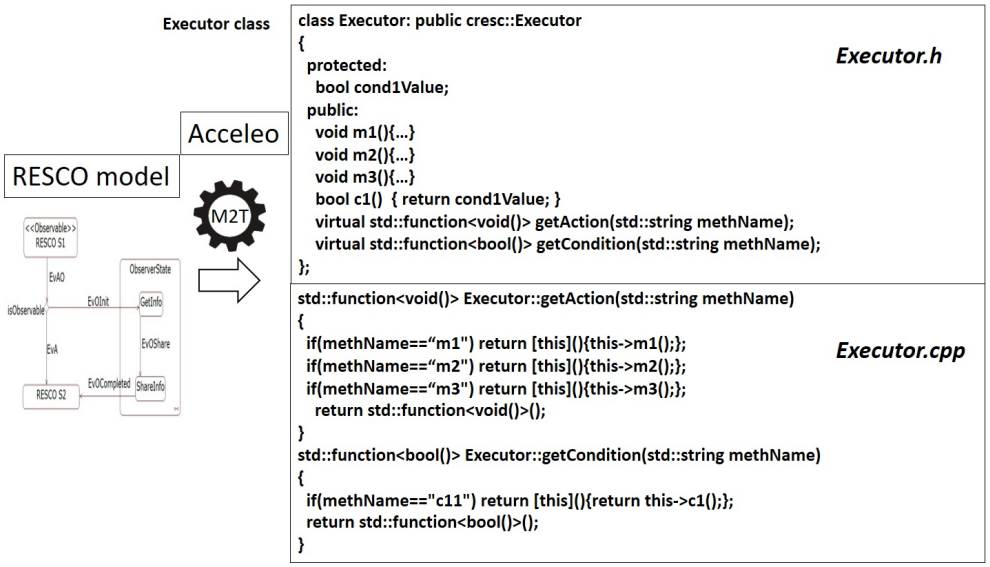


Figure 5.13: CRESCO Executor M2T Transformation

This specific solution addresses embedded and resource limited systems. In this vein, for CRESCO, we performed a M2T transformation to C++ code and we did not use dynamic memory allocation.

Regarding the generic part (runtime infrastructure), listing 5.2 shows a fragment of code of both the Dispatcher and Observer modules which manage the CRESCO software components' states' observability.

```
1  void State::enter()
2  {       ...
3   if(this.context->getWorkingState()->observable)
4   this->context->dispatcher->fillInformation();
5           ...
6  }
7  void Dispatcher::fillInformation()
8  {       ...
9   observer->log.sname=sm->workingState->sname;
10  observer->log.nsname=sm->workingState->nsname;
11  observer->log.fsname=sm->workingState->fsname;
12  observer->log.exeventId=sm->workingEvent.getId();
13  observer->putLog();
14 }
```

Listing 5.2: Fragment of code managing the CRESCO software component's state's observability

Following this solution, table 5.1 provides the information available from the RESCO software components at runtime. The Runtime Monitoring Verification and Adaptation (RMVA) and Runtime Safe Properties Checker (RSPC) that will be presented in Chapters 6 and 7 receive this information at runtime from the states annotated as *observable* in order to check the correctness of the Safe Properties or Correct Transitions defined for each use case.

The messages that are sent from the RESCO software components to both checkers (RMVA and RSPC) are generated by this information and the format/notation of these messages is defined as:

EVId `number`; CurrentState `number`; NextState `number`; FatherState `number`;

5.4 Evaluation

This section evaluates the proposed approach for generating runtime observable and adaptable UML-SM based software components. We provide two case studies and check that the generated software components are able to provide internal information in model terms at runtime. Later, we compare the difference of generating software components employing our tool against other commercial tools. Finally, the obtained results are discussed and some threats to validity of the performed evaluation are highlighted.

5.4.1 Case Studies

The case studies we employed for the evaluation of our approach are the ones presented in Section 4.4.

One of the selected case used for evaluation is an industrial software component that controls a micro-generation device: the Burner controller of the Whispergen commercial device [Pra16]. It covers electrical generation, heat generation as well as micro CHP (Combined Heat and Power). Additionally, we generated 6 synthetic controllers to evaluate the effects on performance and time when we have UML-SMs

Table 5.1: 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

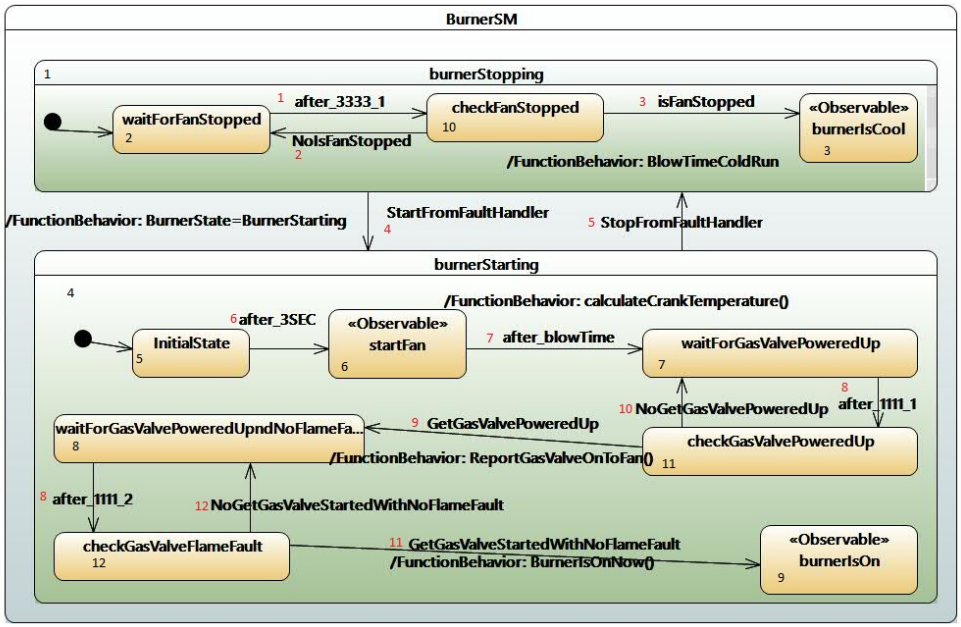


Figure 5.14: Burner's normal behaviour UML State Machine model (SMB1)

with different size and complexity levels. We defined the 6 synthetic cases based on the original Burner's controller UML-SM. We also designed and developed one safe-mode UML-SM to be used in the event of detecting an error at runtime. Thus, the solution performs an adaptation process to the safe mode UML-SM.

For evaluation purposes, the RESCO framework was compared with two commercial tools: SinelaboreRT version 3.7.2.2 [Mue18] tool (specific tool for real time systems) and the Sparx Systems Enterprise Architecture (EA) version 11 tool [Sys15] (generic tool). We selected SinelaboreRT because it is used for developing real-time resource limited systems and Sparx Systems Enterprise Architecture for the reason that it is a generic tool used in a huge number and different domains. The Burner controller shown in figure 5.14 was implemented using both tools and thus, RESCO's performance was compared with tools used in different environments in order to obtain more meaningful results.

The selected Burner controller's normal-mode UML-SM had 13 simple states, 2 composite states, 13 transitions and 13 events. The safe-mode UML-SM had 7 simple states, 2 composite states, 9 transitions and 9 events.

The other case we studied was the Train Control and Monitoring System (TCMS) software system. The TCMS is a complex distributed system that controls many subsystems such as the door control, traction system, air conditioning, etc. The case study concerns a real industrial system where some simplifications were made.

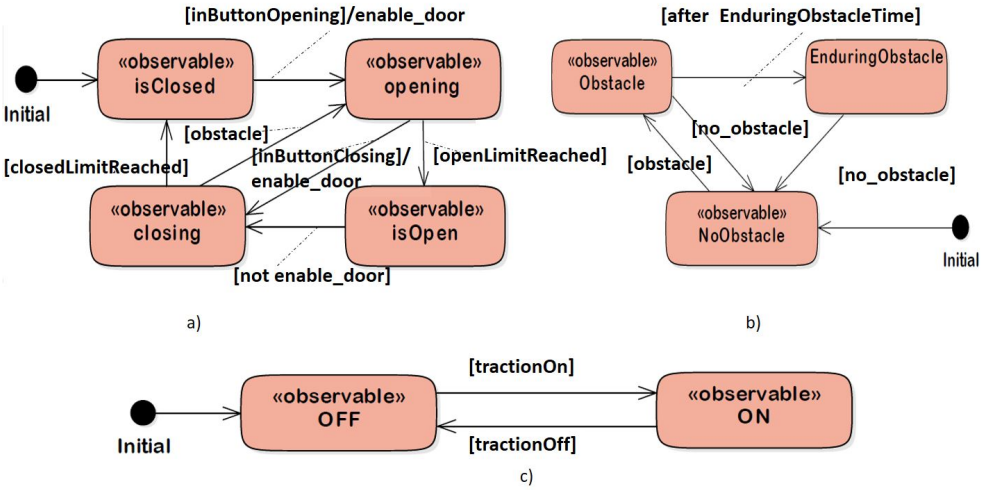


Figure 5.15: UML-SM Diagrams of a) DoorController (ST1) b) ObstacleDetector (ST3) and c) Traction (ST2)

Specifically, the interaction with other components of the TCMS, the dependencies with other subcomponents and their communication were omitted. We only considered 3 subsystems that are the ones shown in Fig. 5.15: DoorController, ObstacleDetector and Traction. In this case, we decided not to compare the performance of RESCO components with software components generated by the other two tools because these controllers are very simple and the expected results were not be meaningful.

Research Questions

One of the objectives of the experiments was to show how it is possible to observe the information of the running system in model terms at runtime (RQ1). Another objective was to demonstrate that the logic of the controllers and the reflection/observation ability of the controllers generated by the C++ REflective State-Machines based observable software COmponents (CRESCO) framework are independent (RQ2). Finally, the overhead of the solution was measured (RQ3 and RQ4). We have defined the following research questions:

RQ1. Is it possible to obtain the model of the software component by analyzing observed information at runtime?
RQ2. Are the logic and the reflection/observation ability of the controllers generated by CRESCO independent?

RQ3. Is the performance of SW components generated by CRESCO framework as good as the SW components generated by other existing tools (EA v11 and SinelaboreRT v3.7.2.2)?
RQ4. How it affects the observability level in the performance of the RESCO framework?

Experimental Setup

All the experiments were executed as a standalone application over a Linux virtual machine configured with a 1 Core processor, 2196MB of RAM, 20GB SSD, and running 64-Bit Ubuntu 16.4 LTS. We have used Eclipse IDE for C/C++ Developers version Oxygen.1a Release (4.7.1a) for generating the executable state machines using the code generated by both CRESCO framework, SinelaboreRT (v3.7.2.2) and EA tool (v11).

To analyse RQ1, RQ2, RQ3 and RQ4, we defined 18 experiments. In order to have more reliable results, each experiment was repeated 1000 times. Table 5.2 shows the characteristics of each of the experiments.

The SMTx state machines are the three controllers that form the TCMS case study:

- SMT1: Door Controller's UML-SM
- SMT2: Traction Controller's UML-SM
- SMT3: Obstacle Detector Controller's UML-SM

The UML-SM SMB1 is the original Burner Controller. SMB2 to SMB7 are the synthetic state machines created for testing purposes to perform experiments with different size and complexity level state machines. To that purpose, we added different number of states to the original one (SMB1): some of them in a flat way (adding new states at the same level) and others hierarchically (adding nested composite states). For example, SMB2 was generated linking two SMB1 original state machines in a flat way; in SMB3 we added a third SMB1, but in this case we added it as a nested composite state in one of the states of the original SMB1; the rest of the state machines were generated following a similar process: addition of SMB1 state machine in the same level (flat way) or adding nested composite states of other SMBx. Thus, we performed the experiments with state machines that have different size and complexity levels.

Considering the works in [Bel13] and [GMP03], to measure the size and complexity of state machines, we used the next metrics: Number of Simple States (NSS- Size

Table 5.2: Experiments Setup

State Machine	Applied to RQ	Observ. %	NSS	NCS	McCabe
SMT1	RQ1,RQ2	100	4	0	4
SMT1	RQ2	50	4	0	4
SMT1	RQ2	0	4	0	4
SMT2	RQ1,RQ2	100	2	0	2
SMT2	RQ2	50	2	0	2
SMT2	RQ2	0	2	0	2
SMT3	RQ1,RQ2	100	3	0	4
SMT3	RQ2	50	3	0	4
SMT3	RQ2	0	3	0	4
SMB1	RQ1,RQ2,RQ4	100	10	2	3
SMB1	RQ2,RQ4	50	10	2	3
SMB1	RQ2,RQ3,RQ4	0	10	2	3
SMB2	RQ3	0	25	4	3
SMB3	RQ3	0	49	4	5
SMB4	RQ3	0	113	9	6
SMB5	RQ3	0	25	5	3
SMB6	RQ3	0	49	11	5
SMB7	RQ3	0	113	26	6

metric), Number of Composite States (NCS- Size metric) and Cyclomatic Number of McCabe (Structural Complexity metric) adapted to state machines.

5.4.2 Results

RQ1 and RQ2 Results

To answer RQ1, first we initialized the original UML-SMs (SMT1, SMT2, SMT3 and SMB1), sent to them 10.000 random events and the externalized monitoring system received and stored the internal status information in model terms of these software components at runtime. Analyzing the runtime observed information, the externalized system was able to represent the structure and transitions of the Software Components Under Study (SCUS). In this experiment, all the states were annotated as *observable*.

Listing 5.3 shows a fragment of the logged information at runtime of the SMB1 software component. In the figure 5.14 (original state machine) we can see the interpretation of the received information (numbers) that represent the name of states and events.

```
1 EvId 4; CurrentState 2;NextState 4; FatherState 1;
2 EvId 6; CurrentState 5;NextState 6; FatherState 4;
3 EvId 7; CurrentState 6;NextState 7; FatherState 4;
4 EvId 8; CurrentState 7;NextState 11;FatherState 4;
```

Listing 5.3: Fragment of the logged information at runtime

Regarding the RQ2, first of all, we have to say that this question could be answered by showing how the RESCO framework is designed. We explained in section 5.3 that the framework was made up of two packages: design package and runtime package. The design package is responsible for the logic and behaviour of the software controllers. As for the runtime package, it is in charge of adding the required infrastructure to have introspection ability at runtime. The latter, which is reused in all the different software components, is always the same. Thus, the RQ2 is answered.

Anyway, in order to answer RQ2 by an experiment, we considered the same experiment as in RQ1. We initialized all the UML-SM based software components, and configured all the states as non-observable. We sent 3333 random events and, as there were no observed states, the externalized runtime monitoring system did not receive internal status information of the software component at runtime.

After that, as the solution enables us to change the observed states at runtime, we changed the configuration of the states and configured 50% of the states as observable. Then, we sent the same initial 3333 random events. In this case the externalized runtime monitoring system received internal status information from the observed states. In the listing 5.3 we can see the logged information of the SMB1. States 2 (waitForFanStopped), 5 (InitialState), 6 (startFan) and 7 (waitForGasValvePoweredUp) were configured as observable.

Lastly, we performed the last change at runtime. In this case, we decided to configure all the states as observable. In this last step, we sent the same 3333 random events to all the software components and the information from all the states was received by the externalized runtime monitoring system.

The same experiments were performed to the SMT1, SMT2 and SMT3. After studying the logs obtained at runtime we were able to extract the model of the three state machines.

RQ3 and RQ4 Results

In RQ3, performance was evaluated in terms of execution time (milliseconds) and percentage of CPU usage. To measure the execution time, we used the *gettimeofday*

instruction. This instruction obtains the current time, expressed as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970. This instruction was launched at the beginning and the end of the execution.

For this research question, we considered the Burner's controller UML-SM (SMB1) and the six synthetic state machines (SMB2-SMB7). As the TCMS's state machines were very simple, we did not consider them in this study.

In these experiments we used 1.000 input events and the observability level of the states in RESCO was 0%. We used this observability level in order to make the comparison in the same conditions for the different tools. SinelaboreRT and Sparx Systems Enterprise Architecture tools do not offer the possibility to generate software components with introspection ability at runtime. RESCO, offers the runtime introspection/observation capability. However, when this capability is activated, additional infrastructure is used and more computational and communication resources are needed. In order to avoid the interference of this infrastructure, we configured the observability level at 0% in the RESCO solution.

Figures 5.16 and 5.17 illustrate the results for RQ3. In terms of time response, SinelaboreRT is the tool (specific for real time systems) that achieves the best results. However, CRESCO's results are similar or even better when the state machines' complexity is low. Concerning the CPU usage percentage, all the results are similar. One reason for that is that the different experiments were performed in very similar situations and in addition, the complexity level of the experimented software components was the same in all the cases. So, the time was the parameter that was affected by the different experiments scenarios. If we consider the synthetic UML-SM (SMB2 to SMB7) for CRESCO and EA tools, when the size and complexity of the state machine increase, the performance of the tools is affected negatively and it decreases: the execution time increases although the percentage of the CPU resource used is only slightly affected. As for SinelaboreRT, when the complexity and size of the state machines is increased, the execution time also increases but to a lesser extent. We have to add that the SinelaboreRT tool is specific for designing and developing real-time systems and Enterprise Architecture tool is a generic tool for different types of systems.

Regarding RQ4, we also launched the original state machine (SMB1) generated by CRESCO when 50% and 100% of the states were observed. When the observability level was 50%, the time response in milliseconds was 272, and when all the states were observed the result was 411. In all the experiments we used 1.000 input events.

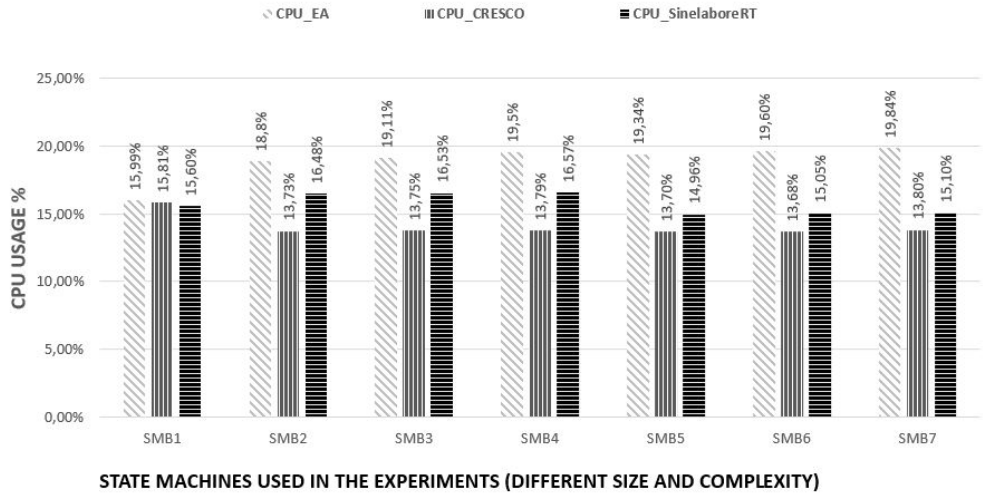


Figure 5.16: RQ3 results: CPU usage % results for SinelaboreRT, EA and CRESCO tools (when observability level 0%).

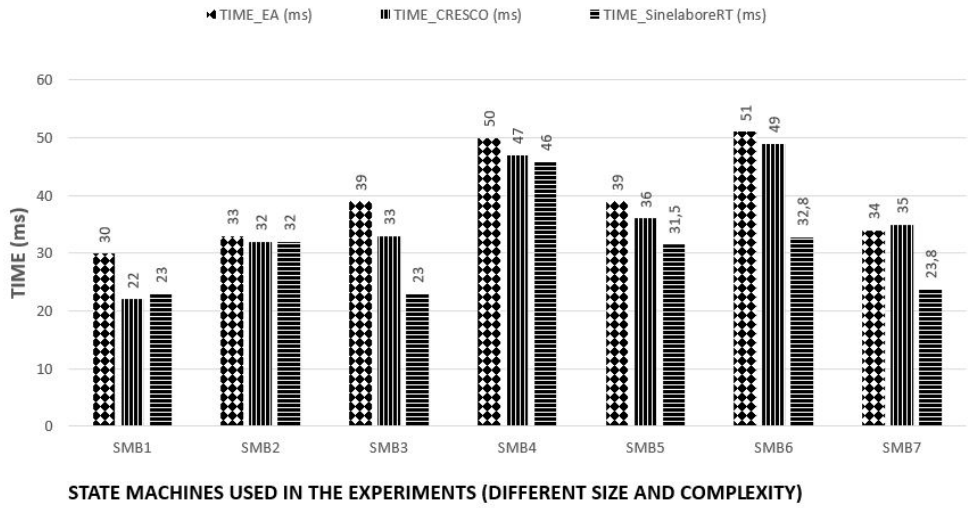


Figure 5.17: RQ3 results: Timing results for SinelaboreRT, EA and CRESCO tools (when observability level 0%).

5.4.3 Discussion

This section discusses the obtained results (Section 5.4.2). The results of the RQ1 show that the solution that we are presenting is able to provide the internal status information in model terms. This runtime information can be employed for various purposes, e.g., fault detection and localization, runtime adaptation/reconfiguration, runtime enforcement and observation of software runtime behavior.

Regarding RQ2, the general objective was to demonstrate that the logic and the reflection ability of the software components generated by CRESCO are independent. Moreover, we wanted to demonstrate that we can change the observability level of the states of the component at runtime. To that end, we initialized the component with all states as non-observable and verified that the externalized runtime monitoring system was not receiving any input from the software controller. After 3333 events, we changed the configuration file at runtime and configured 50% of the states as observable. The externalized runtime monitoring system received information related to the states configured as observable. Finally, we configured all the states as observable at runtime (with no need to change the code) and we received the internal information from all the states at runtime. In this way, we validated that there is no need to change the source code nor to recompile it to change the observability level of the software components; and also that the (1) logic of the software component and the (2) runtime reflection/observation abilities are independent and orthogonal.

One of the strong points of the presented approach is that, in this case, the developer does not need to be bothered about these issues. The behaviour, adaptation and the introspection ability are added orthogonally. The developer only has to focus the efforts on the behaviour of the SCUS, and there is no need to instrument code manually, thereby avoiding new points of introduction of faults.

The general objective of RQ3 was to evaluate the CRESCO framework's performance compared with commercial tools. We compared the performance of the CRESCO framework (0% observability level) with the SinelaboreRT v3.7.2.2 and EA v11 tool results. We considered seven different state machines of different size and complexity. The results show that SinelaboreRT was the tool with best time response results but CRESCO was the next best. We have to consider that we are adding some more infrastructure and logic to enable the solution to have introspection and adaptation ability at runtime. This is one of the main reasons for achieving worse results than SinelaboreRT.

With regard to RQ4, the results with different observability level show that the performance of software components generated by the RESCO framework decreases when increasing the percentage of states observed at runtime. Thus, taking into account

the safe requirements, safety engineer will have to consider a trade-off between the system's performance and its safe behaviour level.

5.4.4 Threats to Validity

This section summarizes the main threats that can invalidate our evaluation:

External validity: An *external validity* threat that usually affects most studies is the number of case studies employed. We used two industrial use cases and in one of them, Burner controller's use case, we configured different state machines for the different experiments. However, some of them (SMB2 to SMB7) were based on the initial Burner's controller UML-SM. Different state machine controllers might lead to different results. Nevertheless, this first experiment was valid to check the correctness of the design and development of the solution since the main objectives were: (1) to show how the software components generated by CRESCO framework were able to provide internal status information in model terms at runtime; (2) to show that the logic and the reflection ability of the generated software components by CRESCO are independent; (3) to measure the CRESCO software components' performance; (4) and to measure the overhead added by the observability level. The experiment employs 10 different state machines and 3 different observability levels, which takes us to 18 different experiments.

Conclusion validity: A *conclusion validity* threat involves the way the execution time and percentage of the CPU usage was measured. To mitigate this threat, each execution is repeated 1000 times.

5.5 Conclusion

The section presents a model-driven approach to automatically generate software components based on UML-SMs with the ability to provide their internal status in terms of model elements at runtime. We defined a platform independent metamodel called RESCO to represent the model of these software components based on UML-SM models annotated by the *observability* profile. Finally, the CRESCO framework, a concrete implementation of the RESCO approach for embedded and resource limited systems in C++ code, was presented.

In order to demonstrate the characteristics of the presented work, we performed an empirical evaluation of how a software component generated by CRESCO was able to represent the software components' UML-SM model elements at runtime. In this empirical evaluation we used two different use cases: Burner Controller of the

Whispergen device and Train Control and Monitoring System. We checked the effects of having different observation levels, too.

We empirically evaluated the performance of the framework (in terms of execution time and percentage of CPU usage) using state machines of different size and complexity. Some experiments were also implemented using different commercial tools (EAv11 and SinelaboreRT) in order to compare their results with those of the CRESCO framework.

The experiments showed that the software components generated by CRESCO framework (RESCO framework instantiation for resource-limited systems) were able to provide their internal status information in model elements terms at runtime. Thus, this characteristic enables us to perform runtime verification in model terms.

In addition, the independence between the logic/behaviour and the reflection, introspection and adaptation capabilities was verified.

Finally, we also concluded that the performance of the software components generated by CRESCO framework is similar to others generated by commercial tools used for resource-limited systems. Nonetheless, this performance is decreased when the observability level of the CRESCO software components is increased. In contrast, the safe behaviour level is increased. Thus, a trade-off between the safe behaviour level and performance has to be made for each specific use case.

Part III

Runtime Monitoring, Verification and Adaptation

Chapter 6

Software Components Level Runtime Verification and Adaptation

Contents

6.1 Introduction

One of the challenges for complex software systems is to increase their safe behaviour. In this way, a Model-Driven Engineering (MDE) approach helps in the design and development phase of these systems and Runtime Verification (RV) techniques help to enhance safe behaviour. Both techniques are complementary.

In order to observe and check the status of the software components, most of the software runtime checking solutions instrument the final source code. In this process, most of the analyzed runtime checking solutions do not use the models used at design time and these are no longer kept at runtime.

Our runtime verification and adaptation solution is based on REflective State-Machines based observable software COmponents (RESCO) framework presented in Chapter 5. Thus, the solution takes advantage of the runtime introspection and reflection ability that offers the software components generated by this framework. The runtime information of the software components observed at runtime could be used for checking their status in model terms at runtime. As a result, failures are avoided because we are detecting errors, based on this runtime in model terms information, before they become failures. The solution presented in this chapter, addresses *software component level* runtime verification and adaptation.

The Runtime Monitoring Verification and Adaptation (RMVA) has the benefit of checking the internal status of the software components and not only their output signals. Consequently, the faults are detected before any transition is performed and output signals are changed, thereby avoiding system failures. This is demonstrated in the evaluation presented in the Section 6.3 and discussed in subsection 6.3.3.

In this chapter we are presenting a runtime verification system able to (1) verify automatically the consistency and correctness of the behaviour of software components at runtime and (2) manage safe-adaptation operations when errors or unanticipated situations are detected.

The software components that will be checked will be modelled by UML-SMs and they are going to provide their own internal information whenever they have to react to an event that may provoke a state transition. At this moment, the runtime verifier will check if the transition is defined as a correct one. The correct behaviour of these software components is inherent to the UML-SM formalism and the checker verifies the runtime information in model terms that provides each software component when is going to perform a transition (current state, current event, next state,...).

The solution addresses software components (e.g. for embedded and resource-limited systems or other type of systems) modelled by Unified Modeling Language -

State Machine (UML-SM) and 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 [GT05],
- residual faults not detected when testing,
- unanticipated faults that were not considered in the design and development phase.

An evaluation of the approach has been carried out using an industrial Burner controller of the electric micro generator Whispergen commercial device [Pra16]. In addition, 3 academic use cases have been implemented. Results indicate that the usage of this framework can accelerate error detection, thus enhancing safe behaviour of the software components at runtime. In this document, we are only showing the results obtained in the industrial use case.

The rest of the chapter is structured as follows: Section 6.2 presents the Runtime Monitoring Verification and Adaptation (RMVA) module. The solution is evaluated in Section 6.3, where an empirical evaluation is performed. Finally, conclusions and future work are summarized in Section 6.4.

6.2 Runtime Monitoring, Verification and Adaptation

This section presents the generation process of the Runtime Monitoring Verification and Adaptation (RMVA) system and its overall architecture and solution which is based on RESCO SW components' introspection and reflection abilities.

First, we present the generation process and then we will continue with the description of the overall architecture of the solution.

6.2.1 Process for defining Safe Adaptation processes and generating the RMVA

The process to generate the RMVA is embedded in a typical design process for developing dependable systems. After designing the software component by means of state machines (UML-SM diagrams), the process for defining safe adaptation processes starts.

Before showing the process and in order to next explanations to be more clear, we are going to define the "List of Correct Transitions" term:

- List of Correct Transitions: All the transitions modelled in the UML-SM. This list is automatically generated from the UML-SM. The format of each correct transition is as follows:

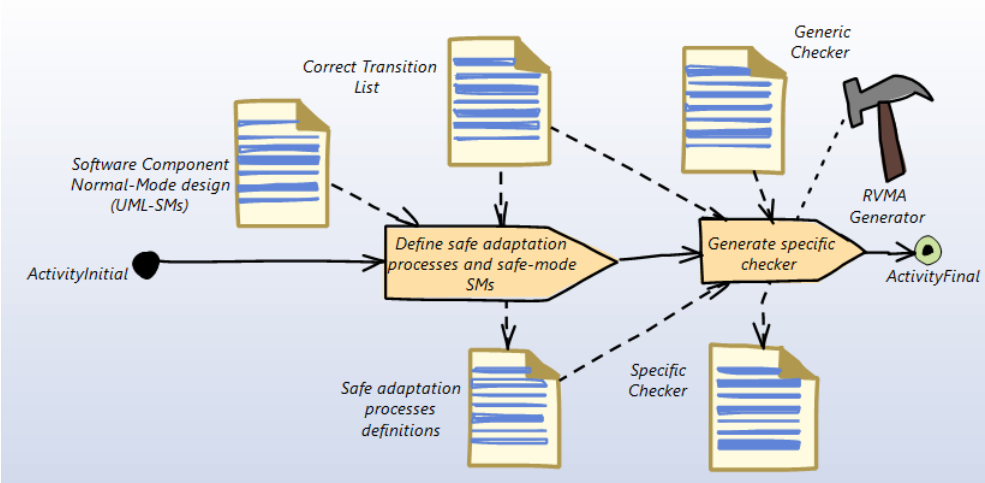


Figure 6.1: Process for generating the RMVA

EVId `number`; CurrentState `number`; NextState `number`; FatherState `number`;

This process has two steps (see figure 6.1):

1. Step1: Define $safeAdapt$ processes to be launched in the event that a transition not defined in the List of Correct Transitions is sent to the RMVA at runtime .

2. Step2: Generate the checker: Runtime Monitoring Verification and Adaptation Generator (RMVAGen) tool transforms the Correct Transition List to RMVA Code (checker, in C++) automatically. RMVAGen uses a generic checker as a basis and adds the specific transition list to the RMVA Checker module and safe adaptation processes to the Runtime Safe Adaptation Manager module.

6.2.2 Runtime Monitoring Verification and Adaptation Architecture based on RESCO software components

Figure 6.2 shows the overall architecture of the Runtime Verification solution.

In this architecture, on the one hand, we have a RESCO software component which is compound by a software controller designed by UML-SM models (defined in Chapter 5). On the other hand, we have another execution environment which acts as a safety bag called Runtime Monitoring Verification and Adaptation (RMVA) module. The main aim of the solution is to take advantage of the runtime introspection and reflectivity ability of the RESCO software components to perform Runtime Verification and Adaptation when needed.

The aim of the Runtime Monitoring Verification and Adaptation (RMVA) module is to receive the internal status of software components to be checked in terms of

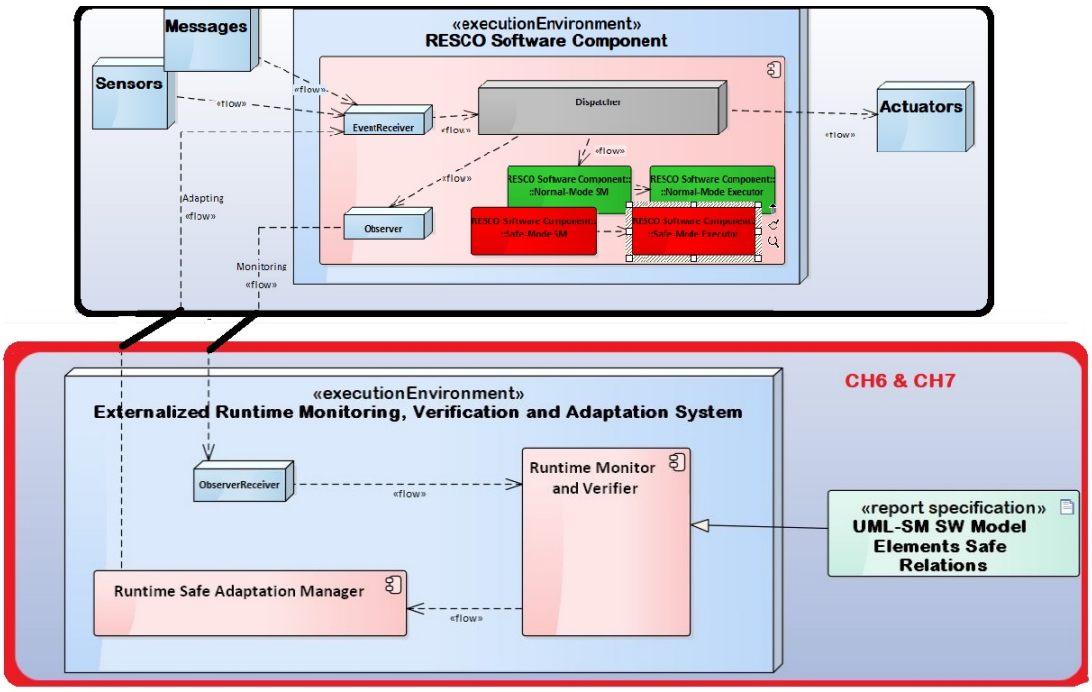


Figure 6.2: Overall architecture of the Runtime Monitoring, Verification and Adaptation (RMVA) scenario

UML-SM model elements and validate this information. For doing this work, the RMVA has two main modules: the Runtime Monitor and Verifier module and the Runtime Safe Adaptation Manager module. First, the information is analyzed by the Runtime Monitor and Verifier module and it decides if the software component is in an unsafe scenario.

The development of the RMVA module was inspired by the solution defined in [AGR14] (the Conformance Monitoring by Abstract State Machines (CoMA) runtime monitor). This module compares the real current system's logged information at runtime with the correct information of the software component under study based on their Abstract State Machines model elements but this solution is implemented in Java. In addition, their solution needs to modify the original Java code of the software components that are going to be checked. Our solution is not using Abstract State Machines model elements but in our case, correct transitions are inherent to the SM's design and a list of correct transitions is generated. Additionally, the software components generated by RESCO do not have to be modified (instrumented) to send their internal information in model terms at runtime.

The RMVA is composed of three blocks:

- ObserverReceiver: this block is in charge of receiving the internal status of RESCO software components. As we are checking only one software component, we do not have to provide a synchronization mechanism to ensure that the information received is a consistent snapshot of the software component. However, the information received by this module provides a timestamp to ensure that it arrives in the same order in which it was produced. It is imperative that the messages verified at runtime are causally ordered so that a correct interpretation of the behavior of the software component is carried out.
- Runtime Monitor and Verifier: this block receives the inputs from the Observer-Receiver and compares them with the expected/safe transitions (event, current state, next state) of the software component to decide if the component is working safely and therefore whether the control is as expected. When generating this checker, a list of correct transitions is provided to this module.
- Runtime Safe Adaptation Manager: this block acts when the software component is not working as expected. Its main functionality is to send a `safeModeProcess` event to the unsafe/uncertain software component's Dispatcher. Thus, the software component performs an adaptation to the predefined 'safe-mode' and therefore, we avoid hazardous situations. This mode of operation ('safe-mode') will be application-specific and defined by the safety engineer.

RMVA Behaviour: The events to be processed (messages, changes in sensors,...) go to the EventReceiver which sends them to the Dispatcher to start the execution algorithm and process the reception of the event at runtime. The Dispatcher analyzes the current status of the software component and calculates if a transition to a new state has to be performed. If a transition has to be performed and the involved states in the transition (at least one of them) is annotated as observable, the current internal information (event, source state, target state, ...) of the software component is sent to the Runtime Monitoring Verification and Adaptation (RMVA) module. The ObserverReceiver element of the RMVA receives this information and sends it to the Runtime Monitor and Verifier in order to check the correctness of the transition.

The RMVA, after checking this information, in the event that the transition is defined as a correct one it sends an acknowledgment event to the Dispatcher as en event to be processed. After the reception of this acknowledgment event in the software component, the corresponding transition and, when needed, updates in the output signals are performed. The system continues working in a 'normal-mode'.

As regards to the RMVA, when the status of the software components is safe, its status information is updated and the RMVA waits for new transition events.

In the event that the RMVA detects that the information of the transition is wrong and not defined in the list of the correct transitions, the Runtime Safe Adaptation Manager starts the adaptation process in order to change the software component to a 'safe-mode' operation model. For doing that, this Adaptation Manager sends the corresponding adaptation event to the Dispatcher of the software component. As a consequence, the software component updates its behaviour to a 'safe-mode' and follows the indications defined for the corresponding adaptation. Thus, we can say that we are performing runtime adaptation for runtime enforcement. This last process is application-specific and has to be defined by the safety engineer.

Once the adaptation in the software controller is performed, the RVMA has also to be updated. As the software component is working now with the safe-mode UML-SM, the Runtime Monitor and Verifier has to consider safe-mode state based correct transition list. After that, the software component will be waiting for new events to process and the RVMA module will wait for new transition traces.

Additionally, the RMVA has the ability to know the internal status of the observed software component when the unsafe/uncertain situation is detected. This information may help in locating the fault that has generated this unsafe/uncertain scenario. We define as uncertain situation or scenario as a circumstance where the current state of knowledge is such that (1) the order or nature of things is unknown, (2) the consequences, extent, or magnitude of circumstances, conditions, or events is unpredictable, and (3) credible probabilities to possible outcomes cannot be assigned.

The next code fragment (listing 6.1) shows the runtime adaptation infrastructure added to the automatically generated code.

```
1 void Dispatcher::processEventinError(Event ev)
2 { ...
3  cresc::State *activeState;
4  ...
5  this->context->getOwnerSm()->setActive(0);
6  this->context->getOwnerSm()->reinit();
7  this->context->getOwnerSm()->id=1;
8  activeState=this->context->getWorkingState();
9  ...
10 this->context->getOwnerSm()->setActive(1);
11 ...}
```

Listing 6.1: Fragment of code managing the RESCO software component's adaptation ability

6.2.3 Internal status information of the monitored software components

For runtime checking, some decisions must be taken about what to observe. Table 6.1 provides the information available from the RESCO software components at runtime. The RMVA receives this information at runtime from the states annotated as *observable* of the software components that compounds the system.

6.2.4 Safe Adaptation Process definition

When the RMVA detects that the transition to be performed is not in the correct transition list, it sends this information to the Runtime Safe Adaptation Manager module. This manager has a table that was created in Step 1 of the RMVA generation process. The information of the table is organized as presented in Table 6.2.

6.2.5 RMVA's Automatic Generation and Dependencies

The final RMVA, is composed of three modules and each of these modules will have specific dependencies. Nevertheless, the code that depends on specific use cases characteristics is generated automatically. In the following list, the dependencies of each of the modules and the way that they are automatically generated is explained:

1. ObserverReceiver: this module is a generic one so it does not need any specific information to generate automatically. This part of the solution is always the same:

Table 6.1: 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

Table 6.2: Safe Adaptation Process Information at Software Component Level

Current State	Id of the Current State
Wrong Next State	Id of the Next State
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

to receive internal information in model terms from the software components. The messages that this module receives follows the previously defined notation/format:

EVId `number`; CurrentState `number`; NextState `number`; FatherState `number`;

2. Runtime Monitor and Verifier: part of this module is specific for each software component. The state-based Correct Transitions list is added in the final code as a "if-else" checking structure. The conversion process is automatic and this specific part is generated from the information generated during the process.

3. Runtime Safe Adaptation Manager: This module is in charge of generating messages (*safeModeProcess* event messages) to be sent to the RESCO software component when an adaptation is needed. The generation of these messages is specific for each software component and it also depends on the specific incorrect transition that has been detected. This part of the code, the creation of these messages, is automatically generated based on the information collected in the tables that define the safe adaptation process.

6.3 Evaluation

This section evaluates the proposed approach for checking at runtime the software components' behaviour in model terms and the adaptation ability when an error or an unanticipated situation is detected. In order to demonstrate our solution's benefits, we have carried out some examples with academic examples (such as simple elevator controls or artificial/synthetic use cases) but in this work we are going to present one industrial case study. Finally, the obtained results will be discussed and some identified threats to validity of the performed evaluation will be highlighted.

6.3.1 Case Study

In this case, the selected case used for evaluation is an industrial software component that controls a micro-generation device: the Burner controller of the Whispergen commercial device [Pra16] already presented in the section 4.4.1. It covers electrical generation, heat generation as well as micro CHP (Combined Heat and Power). For evaluation purposes, two state machines of the Burner controller shown in figures 6.3 and 6.4 were implemented in the RESCO framework: normal-mode Burner controller and the safe-mode Burner Controller.

To check the correctness of the behaviour of each software component, traces of correct transitions are defined using the relation between the events and the states of the controllers (format/notation mentioned before). Each event may have associated a

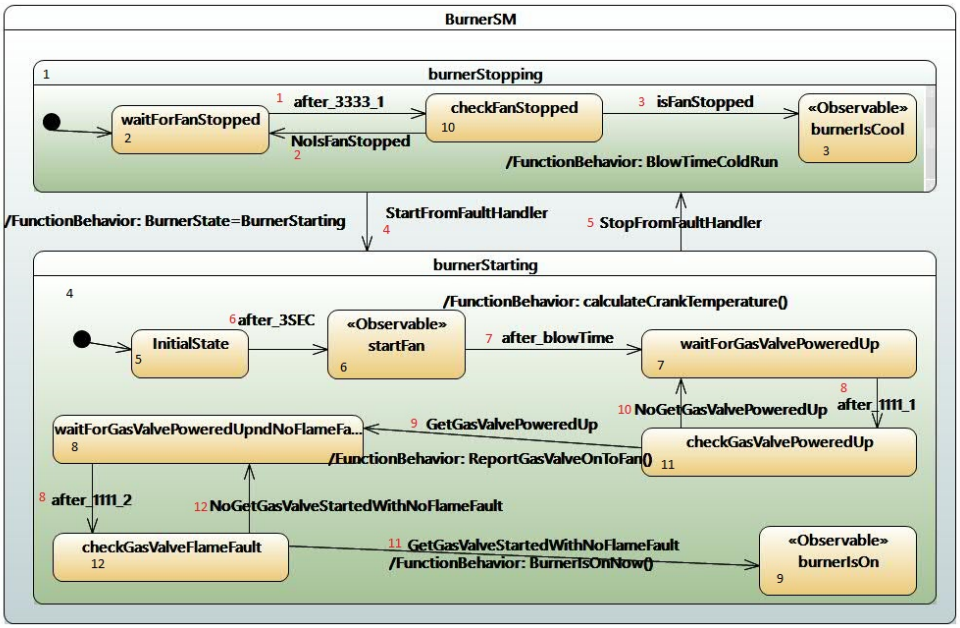


Figure 6.3: The Burner's SM

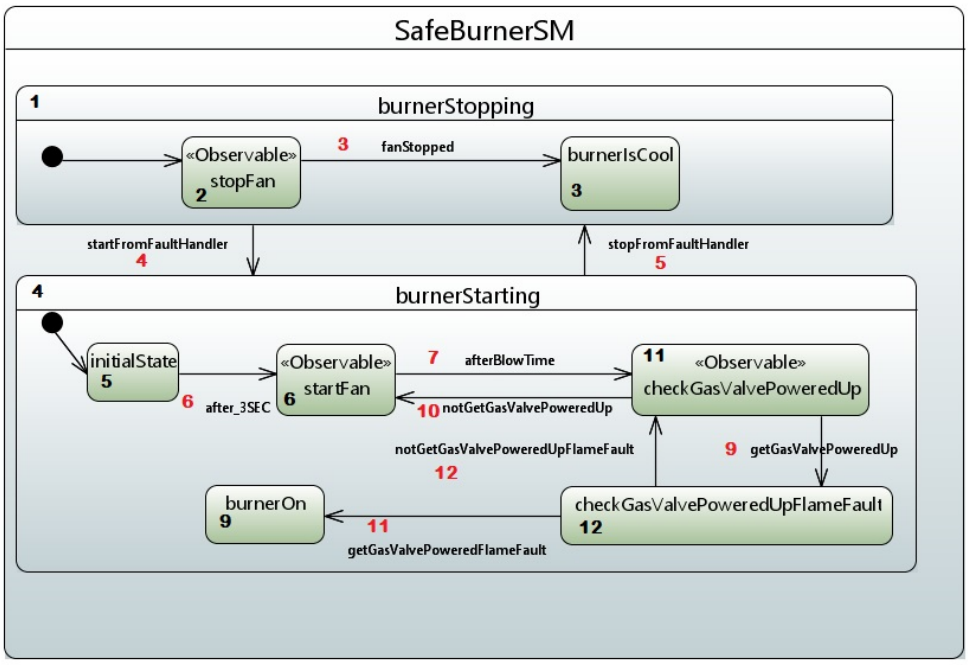


Figure 6.4: The Safe-Mode Burner's SM

reaction that may be a transition and this information is used to define the correct and safe behaviour traces. Then, at runtime, this will be the information to be checked. As an example, some correct traces of the Burner's controller are listed in 6.2. These traces could be automatically obtained from the software components of the system at runtime.

```
1 EvId 4; SourceState 2;TargesState 4; FatherState 1;
2 EvId 6; SourceState 5;TargesState 6; FatherState 4;
3 EvId 7; SourceState 6;TargesState 7; FatherState 4;
4 ...
```

Listing 6.2: Correct and safe traces to be checked at runtime

Before starting with the experiments, the correctness of the implementation was checked at model level by verifying the model (model checking by Papyrus [Pap19]). In addition, as the code of the software components was generated automatically by RESCO, the code structure makes several implementation faults unfeasible. Therefore, we did not inject certain implementation faults (C++ class related faults, such as the ones defined in MuCPP [DPMBPL$^+$17])

Research Questions

The main objective of the experiments is to evaluate how safe behaviour of software components is enhanced by the Runtime Monitoring, Verification and Adaptation module presented in this chapter. As a second objective, we defined to check the relation between the observability level and the performance of the solution. Additionally, as a third objective, the runtime adaptation ability of the solution is demonstrated.

RQ1. Is the presented Runtime Monitoring, Verification and Adaptation solution based on RESCO software components an effective tool for failsafe operation at software components level?
RQ2. How is the relation between the observability level, failsafe detection accuracy and time response?
RQ3. Can this runtime information be employed for runtime adaptation?

Failsafe detection was evaluated by performing fault injection campaigns and measuring the detected number of faults and the time needed. We also measured the response time in the RMVA varying the observability level of the software components in order to know its effect.

Metrics

Failsafe detection was evaluated by performing fault injection campaigns and measuring how many errors were detected by the RMVA checker. Performance was evaluated in terms of CPU resource usage percentage and execution time (milliseconds). To measure the CPU resource usage percentage, we used the output of the command */proc/stat* in Linux. Regarding the execution time, we used the *gettimeofday* instruction at the beginning and the end of the execution.

Experimental Setup

All the experiments were executed as a standalone application over a Linux virtual machine configured with a 1 Core processor, 2196MB of RAM, 20GB SSD, and running 64-Bit Ubuntu 16.4 LTS. We used Eclipse IDE for C/C++ Developers version Oxygen.1 Release (4.7.1a) for generating the executable state-machines using the code generated by RESCO framework.

To analyse RQ1, RQ2 and RQ3 we defined 5 experiments. Table 6.3 shows the characteristics of each of the experiments. All the experiments are based on the original correct software component. In order to address the first two research questions, we used the last four experiments. The main aim for RQ1 was to demonstrate that the solution is able to detect faults at runtime. Regarding RQ2, we also needed to measure the impact of observability level and the solutions' failsafe detection accuracy. In both cases, we needed faulty scenarios therefore we inserted different number and types of faults to the original controller (Burner Controller Normal Mode State Machine - SM1). The third research question aimed to demonstrate the runtime adaptability ability of the solution. In this case, we made experiments with all the different configurations in order to show how the adaptation process was started when an hazardous situation was detected at runtime.

In our case, after analyzing the different fault injection techniques summarized in [HTI97], we decided to use a software fault injection approach and we performed fault injection campaigns to test the RMVA module.

6.3.2 Results

In order to answer all research questions, we used the externalized Runtime Monitoring Verification and Adaptation (RMVA) system and to evaluate the failsafe operation of this system (RQ1), we injected artificial faults by modifying the source code using libfiu tool [Lib]. Libfiu is a library that can be used to inject faults to your code. It

Table 6.3: Experiments Setup

Experiment	**Applied to RQ**	**Number of Unconditional Faults**	**Number of Random Faults & Probability**
Burner Controller Normal Mode State Machine (SM1)	RQ3	0	0
SM1-Fault Injection 1	RQ1,RQ2,RQ3	0	2 (%50)
SM1-Fault Injection 2	RQ1,RQ2,RQ3	2	2 (%50)
SM1-Fault Injection 3	RQ1,RQ2,RQ3	2	2 (%50) + 2 (%75)
SM1-Fault Injection 4	RQ1,RQ2,RQ3	4	2 (%50) + 2 (%75)

aims to be user-friendly by means of a simple API, with minimal code impact and little runtime overhead when enabled.

In order to emulate the random hardware, software (heisenbugs) and unanticipated environmental faults, we injected them by using the *fiu_enable_random(probability)* (Random) option of the libfiu library. This option enables the point of failure in a non-deterministic way, which will fail with the given probability.

Regarding the remaining software faults, we emulated them by using the *fiu_enable()* (Unconditional) option of the libfiu library. This option enables the point of failure in an unconditional way, so it always fails.

We used the last four experiments defined in table 6.3 to demonstrate RQ1 and RQ2. Table 6.4 illustrates the results for both RQ1 and RQ2. Runtime Monitoring, Verification and Adaptation module caught 100% of error activation when all the states were observed and this percentage decreased when fewer states were observed: 86.96% errors detected when 75% observability level, 56.06% when 50%, and 49.5% errors detected when 25% of the states were observed. As to the time response (RQ2), as can be expected, the fewer the observable states, the shorter the time response.

Regarding the RQ3 experiment, we used the same scenario as for RQ1 and RQ2 but in this case we activated the adaptation process. Once the system was correctly configured, we started injecting the same faults as in the first experiments but in this case, when the first error was detected, the original state machine was deactivated and the alternative safe-mode state machine was activated. Therefore, the behaviour of the controller was changed automatically at runtime. RMVA continued logging and analyzing the runtime information. Once the experiment was finished we analyzed the logged information and concluded that after the runtime error detection the safe-mode state machine started working.

Listing 6.3 shows the logs of the safe-mode state machine. Comparing with the

Table 6.4: RQ1 and RQ2 results: Injected runtime faults and the Runtime Verification Module failsafe detection results.

Experiment	Unconditional faults	Random Faults (Probability)	Observed States (%)	Detected Errors (%)	Time (ms)
SM1-FaultInjection 1	0	2 (50%)	100%	100%	225
SM1-FaultInjection 1	0	2 (50%)	75%	82.35%	207
SM1-FaultInjection 1	0	2 (50%)	50%	22.54%	116
SM1-FaultInjection 1	0	2 (50%)	25%	14.7%	70
SM1-FaultInjection 1	0	2 (50%)	0%	0%	9,499
SM1-FaultInjection 2	2	2 (50%)	100%	100%	462
SM1-FaultInjection 2	2	2 (50%)	75%	98.7%	470
SM1-FaultInjection 2	2	2 (50%)	50%	82.3%	332
SM1-FaultInjection 2	2	2 (50%)	25%	75.9%	320
SM1-FaultInjection 2	2	2 (50%)	0%	0%	8,495
SM1-FaultInjection 3	2	2 (50%) + 2 (75%)	100%	100%	409
SM1-FaultInjection 3	2	2 (50%) + 2 (75%)	75%	85.6%	483
SM1-FaultInjection 3	2	2 (50%) + 2 (75%)	50%	84.8%	352
SM1-FaultInjection 3	2	2 (50%) + 2 (75%)	25%	77.3%	254
SM1-FaultInjection 3	2	2 (50%) + 2 (75%)	0%	0%	8,148
SM1-FaultInjection 4	4	2 (50%) + 2 (75%)	100%	100%	414
SM1-FaultInjection 4	4	2 (50%) + 2 (75%)	75%	81.2%	413
SM1-FaultInjection 4	4	2 (50%) + 2 (75%)	50%	33.8%	247
SM1-FaultInjection 4	4	2 (50%) + 2 (75%)	25%	30%	256
SM1-FaultInjection 4	4	2 (50%) + 2 (75%)	0%	0%	7,583

state machine of the figure 6.4 we can conclude that its behaviour was correct and system failure was avoided.

```
1 EvId 3; CurrentState 2;NextState 3; FatherState 1;
2 EvId 7; CurrentState 6;NextState 11; FatherState 4;
```

Listing 6.3: Fragment of the logged information at runtime

6.3.3 Discussion

The main objective of RQ1 was to evaluate if the Runtime Monitoring Verification and Adaptation (RMVA) module was an effective tool for failsafe operation or not for detecting software component level errors or unanticipated situations simulated

by fault injection techniques. We injected unconditional and random faults into the controllers to check whether they were detected by the RVMA externalized module.

The results presented in table 6.4 show that all the faults that affect the observed states were detected by the RMVA. Furthermore, the results show that, as the internal status of the components are observed by the RMVA, those faults that can not be detected immediately by looking at the output signals are detected earlier by the RMVA.

As an example, we are going to consider the one of the wrong transition mutation that was performed by injecting a fault: being the burner controller in the `waitForGasValvePoweredUp` state, the timer *after_1111_1* event was generating a wrong transition to the `waitForGasValvePoweredUpAndNoFlameFault` state instead of a transition to the `checkGasValvePoweredUp` state. There are not changes in the output signals until entering in the `checkGasValveFlameFault` state and other SW monitors are not going to detect the fault until reaching this state. Our solution was able to detect this error before perform the transition to the wrong state (`checkGasValveFlameFault`).

Thanks to having the internal status information of the software components, the RMVA module can prevent faulty scenarios earlier. Furthermore, we can not detect the root cause of the fault by checking only the output signals. Using the RVMA we can have a more accurate insight to detect possible root causes of the fault. Taking these first results into account, we can conclude that the safe behaviour of the system is enhanced by using the RMVA externalized module which starts a `safeModeProcess` operation mode when an error is detected.

As for RQ2, we measured the overhead and time response of the solution having considered different levels of observability in the software components. When the observability level is higher, more states are observed at runtime and, as a consequence, the system is safer. If we check the time response, we can see that, when we have more observed states, the time response increases and the overhead of the software component is also increased. This is because time response is directly linked to the number of observed states. The observed states have to send information to the RVMA and this communication is what increases the time response. It is not a matter of performance but response time. Analyzing the results, we observe that in some of the experiments, the observed percentage of the states and the detected errors do not have a linear relation. This is because some of the injected faults were random and in some cases the states that were configured as observed were the more affected ones by the injected faults. In other cases, we were in the opposite situation: most of the errors were in the non-observed states. The results presented in table 6.4 are the mean values

of the 10 repetitions for each of the experiments.

We may conclude that the safety engineer and the software engineer have to decide on the best initial system configuration by considering these different aspects. A good trade-off between the overhead and the safe behaviour enhancing of the system has to be found. Anyway, in the evaluation scenario, events were sent one after the other in a matter of milliseconds. In real scenarios, this is not a usual way to work and we normally have more time between different events. Thus, the effect of the overload measured in this experiment does not have such an importance.

The results of RQ3 were also positive and the solution was able to indicate in which state and transition of the Software Components Under Study (SCUS) the fault happened. In addition, in case that the adaptation process was activated the system adapted automatically to the predefined safe-mode state machine at runtime. When the observability level of the states of UML-SMs of the SCUS was 100%, the RMVA detected the faults and started the adaptation process before the next state transition was performed. Moreover, the solution allows the runtime externalized RMVA module to ask the SCUS about its internal status at any time. Thus, the robustness of the system is increased.

6.3.4 Threats to Validity

This section identifies threats that could invalidate the performed evaluation. An external validity threat could arise due to considering only one industrial use case. However, other academic examples were performed to avoid this threat. The experimental evaluation presented in this work is a simple industrial use case which was composed of a simple state-machine. Nevertheless, the main objective was to measure the externalized RMVA module's fault detection ability at software component level, its performance and the runtime adaptation ability. The experiment employs 1 correct software component and 4 different faulty scenarios. A conclusion validity threat could possibly arise due to the way the fault detection ability, the execution time and percentage of the CPU usage were measured. To mitigate this threat each experiment is repeated 10 times and results are statistically tested.

6.4 Conclusion

This chapter presents an externalized Runtime Monitoring Verification and Adaptation (RMVA) module that monitors and checks software component level correct transitions defined in model element terms to be verified at runtime.

We evaluated if the RMVA is an effective tool for failsafe operation at runtime. Some experiments were implemented by injecting transient random faults and unconditional faults into the controllers. Furthermore, we empirically evaluated the response time and the detection ability when the observability level of the software components was changed. In addition, the ability to start the adaptation process and how the software components generated by RESCO framework performed the adaptation was evaluated.

The main conclusion is that the RMVA detects all the faults that affect the observed states at runtime, thereby enhancing the safe behaviour of the software components. As it uses components' internal information, it has the ability to prevent faulty scenarios before the system output signals changing significantly. This represents an advantage over software monitors that can only check the output signals.

Another conclusion is that the time response is affected by the number of the observed states and its relation with the error detection rate is inversely proportional. Thus, a trade-off between the time response and safe behaviour have to be considered when designing the software components. Nevertheless, in real scenarios, as the time between events is more relaxed than the ones used in these experiments, this time response will not have a very big effect.

Finally, we have demonstrated that the automatically generated software components by RESCO have the ability to adapt their behaviour when the externalized RMVA module detects an error or an unanticipated scenario.

Chapter 7

System Level Runtime Software Verification

Contents

7.1 Introduction

This chapter presents a Runtime Safe Properties Checker (RSPC). The solution addresses *software system level* runtime verfication and adaptation. RSPC checks a component-based software system's compositional safe properties defined at design phase. As an example, let us consider a system compounded of the following software components:

- a software component that monitors the temperature (O1),
- a software component that controls an engine (O2),
- a software component that activates or deactivates the air system (O3).

The compositional safe properties of this system are defined as (using natural language):

1. If temperature (O1 status) is higher than 30 ^{o}C the engine has to be stopped (O2 status).
2. If temperature (O1 status) is between 20-30 ^{o}C the air system has to be on (O3 status).
3. If temperature (O1 status) is lower than 5 ^{o}C the engine has to be stopped (O2 status).

The main aim of the solution presented in this Chapter 7 is to detect that these compositional safe properties are fulfilled. We address CPSs that are composed of software components that are designed by Unified Modeling Language - State Machine (UML-SM). Those software components are able to provide their internal information and this information is observable in terms of model elements at runtime.

As presented in Section 5.3, REflective State-Machines based observable software COmponents (RESCO) framework generates software components that provide this observability ability at runtime. The RSPC uses software components' internal status information to check the correct composition by checking the states where each of the software component is at runtime. System level safe properties are defined based on the possible states in which each of the software components can coexist at runtime. The checker detects when a system safe property is violated and starts a safe adaptation process to prevent the hazardous scenario.

The approach has been validated in the case study of a Train Control and Monitoring System (TCMS) studied in [GMRE^{+}16] and presented in Section 4.4.2 showing promising results. Main contributions of RSPC are:

- check system level safe properties by monitoring internal status of the system's software components,
- avoid incorrect change of output signals preventing system failures by not performing wrong transitions in the software components,
- development of system's software components and its own (RSPC) is independent: the development of both parts (system and RSPC) does not interfere with each other.

The RSPC solution can be used independently of software components generated by RESCO framework. In any case, the software components we are addressing have to fulfill the following conditions: (1) they have to be designed by UML-SMs and (2) they have to provide the internal status of their observed states at runtime.

RSPC also requires consistent snapshots of the system and, to this end, the observed system's messages must be causally ordered.

The Chapter is structured as follows. Section 7.2 presents the Runtime Safe Properties Checker (RSPC). The experimental evaluation performed to evaluate the approach is presented in Section 7.3. Finally, Section 7.4 summarizes the conclusions of our study and future work.

7.2 Runtime Safe Properties Checker (RSPC)

Component-based software systems are composed of various software components that interact to provide a given system functionality. The aim of the present work is to generate a checker to avoid failures and hazardous scenarios of component-based software systems at runtime due to unsafe interactions of components. To this end, system level runtime safe properties, based on internal status information of the software components, are defined.

These safe properties define the interactions between software components defining the states in which each of the software components can coexist at runtime. In order to check these safe properties, a specific checker, Runtime Safe Properties Checker (RSPC), is generated automatically by a tool that we have developed for that aim: the RSPCGen tool. This tool, takes as an input documents with the definition of the safe properties and the specific safe adaptation processes for the particular use case. Adding the generic part of the checker to this specific information the RSPCGen generates the use case's particular application specific final checker. The generated specific checker (RSPC), will verify the global state of the system based on the states of each software component at runtime.

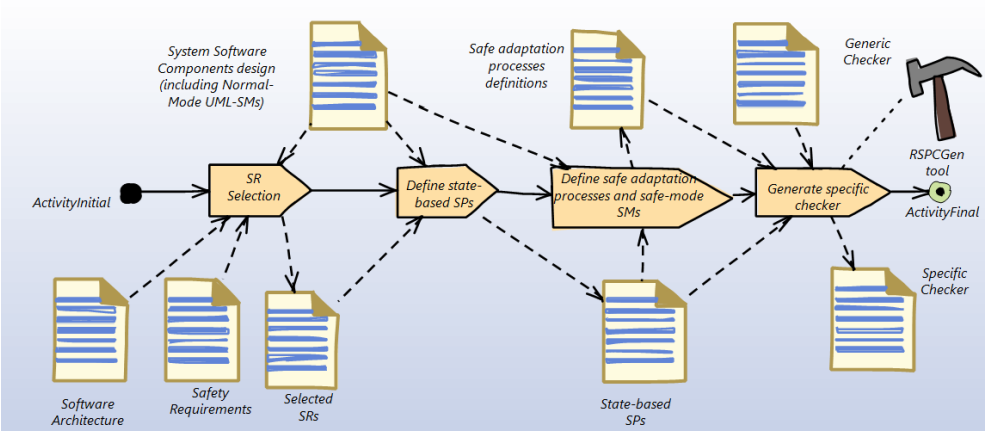


Figure 7.1: Process for generating state-based safe properties chekcer

This Section presents the process of generating the Runtime Safe Properties Checker (RSPC) (subsection 7.2.1), the architecture of the safe properties checker system (subsection 7.2.2), the internal status information to be checked by the RSPC (subsection 7.2.3), how the runtime state-based safe properties are specified (subsection 7.2.4), the safe adaptation process (subsection 7.2.5) and finally how the automatic generation of the checker is performed and its dependencies (subsection 7.2.6).

7.2.1 Process for defining Safe Properties and generating the RSPC

The process to generate the RSPC is embedded in a typical design process for developing dependable systems. After performing software system design phase 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 safe properties starts.

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

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

This process has four steps (see figure 7.1):

1. Step1: Select Safe Requirements (SR) 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 Properties (SP) 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.

7.2.2 RSPC Architecture

In fig. 7.2, the overview of component-based software system's safe property checking architecture is shown.

The software components of the system are modelled by UML-SM and the RSPC has three main components: the Observer Receiver, the RSCP checker and the Runtime Safe Adaptation Manager.

These three components are very similar to the ones described in subsection 6.2.2. The main difference is that in this case, the RSCP checker is checking system level safe properties and not the correctness of the transitions to be performed by specific software components.

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.

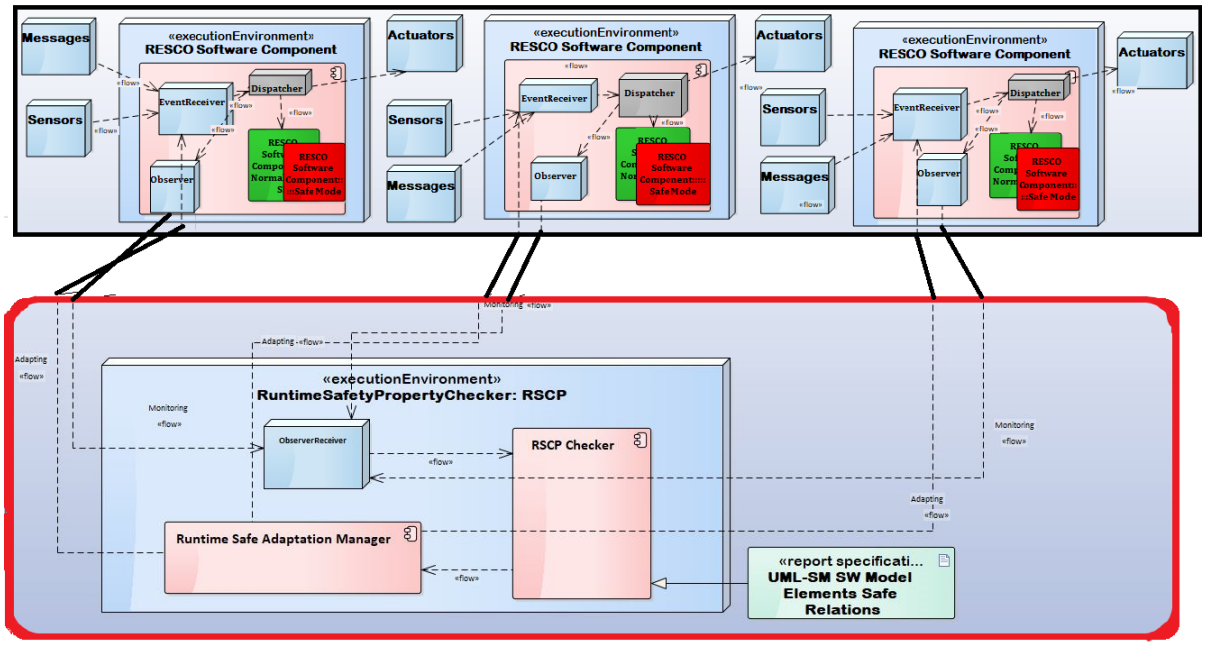


Figure 7.2: General Architecture of the Safe Properties Checking System

7.2.3 Internal status information of the monitored software components

In this case, the information sent by the monitored software components is the same as the information defined in subsection 6.2.3. In both solutions, the monitored software components are generated by the RESCO framework.

7.2.4 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 system properties related to the internal behavior of the software components that are part of the system in terms of their UML-SM model. 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 can not be further decomposed. Furthermore, the primitive components have a behavior 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.

A safe property will be related to the states of the components involved in the property. A grammar with regular expressions is used to specify what to check. The relevant syntax of this grammar has been specified in the following listing:

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

With this grammar, it is possible to specify properties regarding the active states of components of the system. For instance, we can specify that the active state of the C_{Door} must be S_{Closed} when the active state of the $C_{Traction}$ is S_{On}.

$$C_{Traction}.S_{On}\ \ \textbf{implies}\ \ C_{Door}.S_{Closed}$$

We can also use logical operators as well as parentheses to build more complex conditions. For example, when C_{Door} is in the S_{Closed} and $C_{Traction}$ is in the S_{On}, then the $C_{Obstacle}$ must be in S_{NoObst}.

$$C_{Door}.S_{Closed}\ \ \textbf{and}\ \ C_{Traction}.S_{On}\ \ \textbf{implies}\ \ C_{Obstacle}.S_{NoObst}$$

In the previous examples, we referred to the active states of the components in the current time point. We can also specify constraints regarding future states of components using timedConstraints. A timedConstraint specifies a constraint that must become true in the interval between the current time and the current time plus timeUnits. For example, when the active state of the $C_{Obstacle}$ is S_{Obst} then the active state of the C_{Door} must be $S_{Opening}$ and not any other state of the C_{Door} component in less than 2000 milliseconds time.

$$C_{Obstacle}.S_{Obst}\ \ \textbf{implies}\ \ C_{Door}.S_{Opening}\ \ \textbf{in}\ \ 2000$$

Table 7.1: 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

7.2.5 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 Runtime Safe Adaptation 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 7.1.

7.2.6 RSPC's Automatic Generation and Dependencies

The RSPC's automatic generation process follows the same steps that we defined for the RMVA. The main difference between both solutions is that the RSPC is checking Safe Properties and the RMVA Correct Transitions.

The final RSPC, is also composed of three modules and each of these modules have specific dependencies. In the following list, the dependencies of each of the modules and the way that they are automatically generated is explained:

1. ObserverReceiver: this module is a generic one so it does not need any specific information to generate automatically. This part of the solution is always the same: code in charge of receiving internal information in model terms from the software components. The messages that this module receives follows the previously defined notation/format:

   EVId `number`; CurrentState `number`; NextState `number`; FatherState `number`;

2. RSCP Checker: part of this module is specific for each use case. The defined state-based System Level Safe Properties are added in the final code as a "if-else" checking structure. The conversion process is automatic and this specific part is generated from the information generated during the process.

3. Runtime Safe Adaptation Manager: This module is in charge of generating messages (*safeModeProcess* event messages) to be sent to the RESCO software components when an adaptation is needed. The generation of these messages is specific for each use case and it also depends on the specific safe property that is not fulfilling. This part of the code, the creation of these messages, is automatically

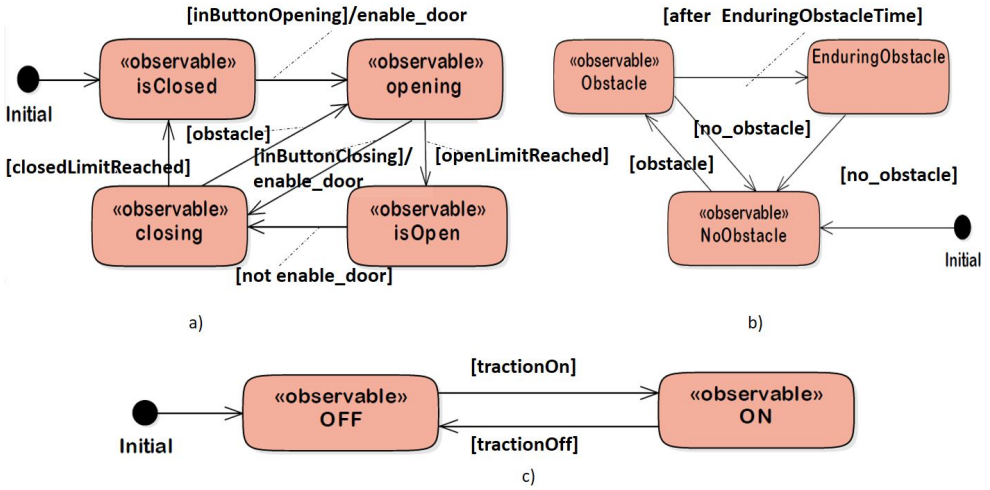


Figure 7.3: UML-SM Diagrams of a) DoorController b) ObstacleDetector and c) Traction

generated based on the information collected in the tables that define the safe adaptation process.

7.3 Evaluation

7.3.1 Case Study

In this Chapter, a Train Control and Monitoring System (TCMS) was considered. As presented in Section 4.4.2, the TCMS is a complex distributed system that controls many subsystems such as the door control, traction system, air conditioning, etc.

Fig. 7.3 shows the UML-SM of the DoorController, ObstacleDetector and Traction.

The system level requirements concerning the operation of opening and closing of doors are satisfied by the following components:

- `TCMS` component decides whether to enable or disable the doors considering the driver's requests and the train movement. Thus, doors must be enabled before they can be opened;
- `Door` component controls and commands the opening and closing of a door;
- `Traction` component controls and commands the train movement;
- `Obstacle Detection` component manages the obstacle detection in the door.

The case study concerns a real industrial system where some simplifications were made. Specifically, the interaction with other components of the TCMS, the dependencies with other subcomponents and their communication were omitted.

At software system level, to check the correct behaviour of the system, the safety engineer has to define system level Safe Requirements (SR). Then, some of these SRs are transformed to Safe Properties (SP). Each SP will be defined using the state information of the different software components that conform the software system. Then, at runtime, this will be the information to be checked.

The following safe requirements are defined in the context of this case study: (Section 7.2.1: Process step 1)

1. **SR1.** When door is open or opening or closing, the traction is off.
2. **SR2.** When an obstacle is detected and the door is closing, it starts to open within 2 seconds.
3. **SR3.** When traction is on, the door must remain closed.
4. **SR4.** When traction is on, there must not be obstacles detected.
5. **SR5.** The door shall not be in opening state more than 15 seconds.

The next step is to express these safe requirements as state-based safe properties (Section 7.2.1: Process step 2), and the result is shown in the listing 7.1. Each system level rule in this listing corresponds to each one of the aforementioned requirements.

```
1  SP1:CDoor.Sopen or CDoor.Sopening or CDoor.Sclosing
2      implies CTraction.Soff
3  SP2:SP2:CObst.Sobst and CDoor.Sclosing implies
4          CDoor.Sopening in 2000 and not CDoor.isOpen
5          and not CDoor.isClosed and not CDoor.Sopening
6  SP3:CTraction.Son implies CDoor.Sclosed
7  SP4:CTraction.Son implies CObst.Sno_obst
8  SP5:CDoor.Sopening implies CDoor.Sclosed
9      or CDoor.Sclosing or CDoor.Sopened in 15000
10     and not CDoor.Sopening
```

Listing 7.1: Safe Requirements expresed as Safe Properties

At this point, safe adaptation processes are defined for each of the safe properties (Section 7.2.1: Process step 3). In this case, when designing the adaptation process we analyzed different possibilities. One of them was to design safe mode UML-SMs for

each subsystem and adapt all of them but as a more simple solution we finally decided to stop the train as a safe solution. For doing that, the safe adaptation process is in charge of sending a $tranctionOff$ event to the Traction software component. Thus, the EventReceiver of this software component receives this event and after that, the Dispatcher performs a transition to the OFF state and once the outputs signals are activated the train stops.

Research Questions

The objective of the experiments is to evaluate the error detection ability and the overhead of the RSPC tool. We have defined the following research questions (RQs):

RQ1. Is the RSPC an effective tool for failsafe operation for component based systems? This RQ is defined to measure the ability of RSPC to detect different types of system level errors. To address this RQ, we performed fault injection campaigns.
RQ2. Can the RSPC detect errors before the system output signals are changed? RQ2 aims to demonstrate that RSPC is able to detect errors before propagating them to system's outputs. To tackle this RQ, we measured the time the RSPC needs to detect violations of safe properties and compared with the time it would take to detect the same error by checking the output signals.
RQ3. How is the RSPC's overhead in terms of resource usage and timing affected by the number of system safe properties? RQ3 measures the performance and overhead of RSPC with different number of safe properties. Thus, the scalability of the solution will be tested. To address this RQ, new safe properties were defined and the experiments were performed with different numbers of safe properties. The percentage use of CPU and time response were measured in all the experiments.

Metrics

Failsafe detection was evaluated by performing fault injection campaigns and measuring how many errors were detected by the RSPC checker. Performance was evaluated in terms of CPU resource usage percentage and execution time (milliseconds). To measure the CPU resource usage percentage, we used the output of the command */proc/stat* in Linux. Regarding the execution time, we used the *gettimeofday* instruction at the beginning and the end of the execution.

Experimental Setup

To analyse RQ1, RQ2 and RQ3 we defined 6 experiments. In order to have more reliable results, each experiment was repeated 100 times. In addition, as we had 3 different UML-SMs, we wanted to experiment different combinations between them randomly. RSPC relies on a previous successful verification of the design by model checking, thus design faults are outside the scope of this analysis. As the code of the software components was generated automatically by RESCO, the code structure makes several implementation faults unfeasible. Therefore, we did not inject certain implementation faults (some of them defined in MuCPP [DPMBPL$^+$17]).

As for the RMVA solution presented in Chapter 6, the faults that the presented solution is designed to provide a failsafe against are: (1) random faults (hardware and software (heisenbugs)), (2) remaining implementation faults (inserted by the diverse implementation of the actions or the non-detected ones in complex systems) and (3) uncertain and unanticipated environmental faults. Taking as a basis the different components of the Train Controller, we performed fault injection campaigns to test the RSCP.

As there are different types of fault injection techniques, as for the RMVA presented in Chapter 6, after analyzing the different techniques summarized in [HTI97], we decided to use the software fault injection techniques.

To evaluate RSPC's failsafe ability, we injected artificial faults by modifying the source code using libfiu tool [Lib]. We decided to use this method because it was one way to simulate faulty scenarios, with faults that our solution addresses, that enabled us to measure the failsafe ability of the RSPC solution. The characteristics of this tool were presented already in subection 6.3.1

Table 7.2 shows the characteristics of each of the experiments. The first experiment was performed using a correct system without implementation faults. In this case, we inserted 5 random faults with a probability of 50% by the *fiu_enable_random(0.5)* (Random) function of the libfiu library in 5 different points of the solution. We checked the 5 system level safe properties we defined (see listing 7.1).

In the next three experiments, we injected 1, 2 and 3 faults respectively using the libfiu's Unconditional function (*fiu_enable()*). Specifically wrong transitions were inserted. One of the wrong transitions were inserted in a non observed state ($EnduringObstacle$: the $noObstacle$ event's target state was changed to the $Obstacle$ state). The other two wrong transitions were performed in observed states (in the $closing$ state, the $obstacle$ event was generating a self transition and in the $opening$ state, the $openLimitReached$ event was generating a self transition).

Thus, we emulated the effect of random hardware and software faults (such as

Table 7.2: Experiments Setup.

Experiment	Applied to RQ	Number of Random Faults & Probability	Number of Unconditional Faults	Number of Safe Properties
FaultInjection1	RQ1,RQ3	5 (50%)	0	5
FaultInjection2	RQ1,RQ2	5 (50%)	1	5
FaultInjection3	RQ1	5 (50%)	2	5
FaultInjection4	RQ1,RQ2	5 (50%)	3	5
SafeProperties1	RQ3	5 (50%)	0	10
SafeProperties2	RQ3	5 (50%)	0	20

heisenbugs) in all the FaultInjection experiments and unanticipated environmental and implementation faults (such as systematic faults) in the FaultInjection 2, 3 and 4.

In order to answer the third research question we added some system level artificial safe properties. We performed two more experiments in the same conditions as in the first experiment but in this case, the number of safe properties to be checked by the RSPC was bigger (+5 and +15) in both cases. The artificial safe properties we added did not have real physical sense but we defined new artificial relations between the different states in which each of the software components can coexist at runtime. In all the experiments we used 1000 input events.

All the experiments were executed as a standalone application over a Linux virtual machine configured with a 1 Core processor, 2196MB of RAM, 20GB SSD, and running 64-Bit Ubuntu 16.4 LTS.

7.3.2 Results

In this section we analyze the results of the empirical evaluation. Table 7.3 illustrates the results for RQ1.

As we performed each experiment 100 times, the results presented are the average values of the measured outcomes. RSPC catches 93% of fault activations and 100% of fault activations that result in safe properties' violations. RSPC does not catch all fault activation because some of the faults occurred in non-observed states. Therefore, these states are not providing information at runtime and consequently the RSPC can not detect those errors.

Regarding RQ2, RSPC not only detects the faulty scenarios but it detects errors before output signals are changed. We measured the time response of the *FaultInjection2* and *FaultInjection4* scenarios. In the event that we look at the output signals for error detection, we would detect the error once the output signal changed (after 2 and 15 seconds in the experimented scenarios). The experimental results are shown in table

7.4. Our results say that our solution was able to detect these erroneous situations in less than 70 milliseconds.

RSPC checks system level safe properties based on internal status of the various individual software components of the system. To our knowledge, no other software checkers have this ability. RSPC observes the components' internal status information and therefore detects critical errors earlier than other software checkers.

As an example, we are going to consider the first experiment of table 7.4. In this case, a wrong transition was injected as a fault: the door controller being in the $closing$ state, the $obstacle$ event was generating a self transition instead of a transition to the $opening$ state. There are no changes in the output signals and, as the safe property says that the system has to be in the $opening$ state before 2 seconds, other software monitors would not detect it until the timer expires. What is more, they might not detect the root cause of the fault. RSPC detected the faulty scenario in less than 70 milliseconds because the SP2 was not fulfilled and the transition suggested by the software component is not a correct one.

Concerning RQ3, a correct system with different number of safe properties was used and the overhead of the RSPC was measured. Fig. 7.4 shows the results when the number of system level safe properties is changed between 5, 10 and 20.

Both execution time and CPU usage percentage are increased when the number of safe properties are increased. The impact on the execution time is not significant (less than 1% of the time when the safe properties' increment is 400%). As for the CPU usage, in this case the impact is greater (from 49.4% to 58%).

Table 7.3: RQ1 results: Injected runtime faults and RSPC failsafe detection results. `IT:Incorrect Transition;OS:Observed State`

Experiment	Injected Implementation Faults	Number of Random Faults & Probability	Number of Safety Violations	Detected Faults (%)	Detected Safety Violations (%)
FaultInjection1	`0`	5 (50%)	76	100%	100%
FaultInjection2	`1 IT(in OS)`	5 (50%)	87	100%	100%
FaultInjection3	`2 IT(1 in NOT OS, 1 in OS)`	5 (50%)	85	87%	100%
FaultInjection4	`3 IT(1 in NOT OS, 2 in OS)`	5 (50%)	105	84%	100%
Total	**`6 IT, 2 NOT OS`**	**5 (50%)**	**353**	**93%**	**100%**

7.3.3 Discussion

We summarize the results of the empirical evaluation and what they tell us about the research questions.

The general objective of RQ1 was to evaluate if the RSPC was an effective tool for failsafe operation or not. We injected random conditional and unconditional faults in addition to implementation (wrong transitions) faults in the software components to be checked. We used the libfiu library to inject these faults and we checked if the RSPC checker was able to detect them. The results presented in table 7.3 show that all the safety critical faults were detected by the RSPC. The faults that violate the safe properties were detected because the states that are part of the safe properties are configured automatically as observable. It does not make sense not to configure as observable because the checker needs the information of these states at runtime to check fulfillment of the safe properties.

Table 7.4: RQ2 results: Error detection time for different approaches.

Experiment	**Error Description**	**Detection Time RSPC**	**Detection Time by Output Signals**
FaultInjection2	Door Component:**closing** state **when** obstacle detection event: **self transition to closing state**	70 msec.	2 sec.
FaultInjection4	Door Component:**opening** state **when** openLimitReached event: **self transition to opening state**	70 msec.	15 sec.

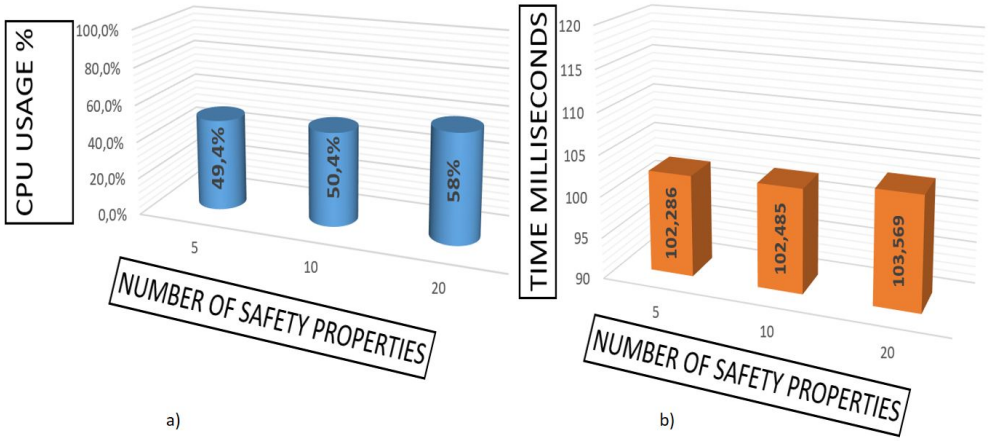


Figure 7.4: RQ3 results: When the number of safe properties is increased, a) the CPU usage is increased b) the response time hardly increases.

As for RQ2, the results show that, as the internal status of the components are observed by the RSPC, those errors that can not be detected immediately by looking at the output signals are detected sooner by the RSPC. Thanks to this, the RSPC checker can prevent faulty scenarios earlier. Additionally, using the RSPC, we can detect the root cause of the observed error. Taking these first results into account, we can conclude that the safe behaviour of the system is enhanced by using the RSPC checker.

Finally, the results of RQ3 show that when the number of safe properties is increased, the performance of the RSPC is decreased. The time response is hardly affected and, although the CPU usage is affected, it does not have a very big impact. We conclude that the RSPC can manage easily a considerable number of system level safe properties (more than 20). There will be other additional requirements to be checked in the system but in this research we were focusing only on the system level safe requirements.

The internal states of the software components generated by RESCO can be configured as observable or not at runtime. In previous experiments (see subsection 5.4.2, RQ4 results), the overhead of having different levels of observability of those states was measured. The conclusion was that having more observable states, the overhead of the software components was increased. Observing only the states that are part of the system level safe properties is a good trade-off between the overhead and the safe behaviour assurance of the system.

The RSPC is generated automatically without extra effort for the developers by the RSPCGen tool presented in section 7.2. Additionally, software components of the system have no interferences from the RSPC, except when detecting an error, in which, it starts a safeStop process.

7.3.4 Threats to Validity

An external validity threat could arise due to considering only one industrial use case. The system was composed of 3 simple state-machines. However, as the main objective was to measure the RSPC checker's error detection ability and performance, this first experiment was valid to check the correctness of the design and development of the checker.

The experiment employs 3 different safe property levels (5, 10 and 20), 1000 input events, 3 incorrect transitions in observed states, 1 incorrect transition in non-observed state and 5 random faults with a 50% probability of occurrence. A conclusion validity threat could possibly arise due to the way the error detection ability, the execution

time and percentage of the CPU usage was measured. To mitigate this threat, each experiment is repeated 100 times.

7.4 Conclusion

This chapter proposes a RSPC that is automatically generated and which considers system level specific safe properties to be verified at runtime.

We empirically evaluated the performance of the checker (in terms of execution time and percentage of CPU usage) using a different number of system level safe properties to be 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.

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.

Another conclusion is that the number of safe properties used at system level affects the performance of the RSPC but its impact is not very significant. The RSPC can manage a considerable number of safe properties (more than 20).

Our last conclusion is that the process to implement and generate the RSPC is cost-effective as the system level RSPC is 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.

Part IV

Final Remarks

Chapter 8

Conclusion

Contents

This chapter concludes the thesis. Specifically, Section 8.1 summarizes the contributions, discusses the validation of the hypotheses and highlights the main limitations of the proposed solutions. Section 8.2 discusses a set of lessons learned we extracted from the thesis. Finally, short and mid-term future work are exposed in Section 8.3.

8.1 Summary of the Contributions

Safe requirements of Cyber-Physical Systems (CPSs), as system that interact with humans or/and affect the environment, have increased in the last decade. These systems have to be robust and dependable under any circumstances. Taking into account these issues, runtime enforcement of CPSs' safe behaviour is a big deal and challenge we have identified.

In order to reach runtime enforcement safe behaviour we need to know the status of those CPSs at runtime. One way to have information of the internal status of these systems at runtime is to instrument their code. Instrumenting the generated code of the modelled software component is a expensive and non-optimal solution: faults could be inserted during the instrumentation process, and as a result, having automatically the software components' information in model terms at runtime has been envisioned as an efficient means for runtime verification [CK13]. In this dissertation, a methodology and a framework were proposed to automatically generate software components that provide internal status in UML-SM model terms at runtime. In addition, runtime verification and adaptation modules were generated in order to enhance the safe behaviour of software components and systems at runtime.

As we stated at the beginning of the document, the main four contributions of this work are:

1. A methodology supported by a framework, REflective State-Machines based observable software COmponents (RESCO), that is able to generate software components modeled by Unified Modeling Language - State Machine (UML-SM) that provide their internal status information in model elements terms at runtime.

2. RESCO framework: Automatic generation of software components with internal status information observation ability in UML-SM model terms (current state, event, next state,...). The software engineer focuses on the design of the functional behaviour of the software component, whereas the internal observability ability in model terms is added automatically. The software engineer is not involved in changing the model or source code to provide this information at runtime and thus, can focus exclusively on modelling the behaviour of the software components by

UML-SMs. Additional infrastructure for having internal status information and adaptation ability at runtime is automatically added by the framework. As a result, this information could be used to increase the safe behaviour of the CPSs without increasing the complexity of the development process. Software components generated by this framework especially address resource-limited systems.

3. Runtime Verification. An external monitor and verification system is used to check the internal status of the UML-SM based software components in model terms before a transition in their state, and therefore a change in the output signal, is performed. This allows us to detect faults before the failure happens, increasing the resilience against faults. Different types of monitor and verification systems have been considered: (1) systems that check the correct behaviour of each software component and (2) systems able to check the rules governing the relations between different software components in component based systems based on component level model based information. In the latter, system-wide rules or safe properties, based on software components' internal state status information in model terms, are defined by the safety engineer.

4. Runtime Adapter. An externalized runtime adaptation module has been developed. This module, has been based on a previous work [GS02]. In the solution, the adaptation is triggered by unexpected events or when a fault is detected at runtime. This allows to protect the system against unsafe situations and scenarios ensuring that the software component performs safe actions.

The first two contributions of the dissertation corresponds to a tool supported methodology that automatically generates reflective UML-SM based software components and the REflective State-Machines based observable software COmponents (RESCO) framework.

The RESCO methodology follows a process that uses existing tools and languages as Papyrus [ecl18], ATL [ATL18] and Acceleo [A16] for Model To Model (M2M) and Model To Text (M2T) transformations. The framework allows for a systematic generation of each software component in an automated manner starting with the design of the behaviour of the controller by UML-SM models. Furthermore, as the process is automatic, it reduces the error proneness. Once the UML-SMs are generated, an investment of time must be employed by safety engineers to decide which of the states of the controllers are critical and which of them have to be observed at runtime. These decisions will be based on the safe properties defined at software system level and the critical states of the controller. Having all this information, RESCO-SMs are generated automatically.

The methodology and tooling developed address software components that are designed by UML-SMs but in the last step they perform a Model To Text (M2T) transformation to the specific programming language. In the current solution we have developed this transformation only for resource-limited systems. It transforms the models to C++ code (by CRESCO) and the generated software components address resource-limited systems.

The third contribution corresponds to a runtime monitor and verifier which checks the correctness of the behaviour at component and system level. We have developed a process and tooling for generating automatically these checkers. For system level runtime checker we have defined a language for defining safe properties and once they are defined the checker is generated automatically.

To further enhance the safe behaviour of software components and the systems compound by them, the fourth contribution consists of a runtime adapter based on the decisions made by the runtime verifier. Once an error or an unanticipated scenario is detected, the runtime adapter starts its process. As the solution is based on reflective UML-SMs, they enables to adapt their behaviour at runtime. Thus, once the runtime adapter is notified, it detects which is/are the involved software component and decides which is the safe UML-SM model for each one to start the adaptation in a safe mode.

Overall, this thesis proposes methods to systematically and automatically generate reflective UML-SM based software components and runtime checkers for runtime monitoring, verification and adaptation. We believe that the proposed methods advance the current practice to generate robust software components and systems and enhance their safe behaviour at component and at system level.

8.1.1 Hypotheses Validation

We stated three research hypotheses in Section 4.2. This section analyses each of the contributions and argues whether the stated hypotheses have been validated.

First hypothesis

The first hypothesis is stated as follows:

> "The use of Unified Modeling Language (UML) State Machine based software components with introspection and reflection ability in model terms helps the early detection of errors in software systems".

We proposed a methodology supported by a tool, named RESCO. Specifically, as controllers' behaviour modeling notation we employed Unified Modeling Language

- State Machine (UML-SM) modelled using the Papyrus [ecl18] tool, which were later processed by the Model To Model (M2M) transformation in ATL [ATL18] to generate RESCO-State Machines. An observability profile is used in order to define the states of the UML-SM that are going to be observable at runtime. Finally, using Model To Text (M2T) transformations written in Acceleo [A16], C++ REflective State-Machines based observable software COmponents (CRESCO) software components were generated. These software components have the introspection and reflection ability in model terms at runtime.

We evaluated the method and the tool employing the case studies presented in Section 4.4, the controller of a Burner and a Train Control and Monitoring System (TCMS). We demonstrated (1) that the software components generated by RESCO had the introspection and reflection ability at runtime and in addition, (2) that the logic of the controllers (use case dependent) and the runtime reflection and introspection abilities (generic for all the cases) are independent. We also measured and compared the performance of the software components generated by the CRESCO framework with software components generated using other commercial tools such as SinelaboreRT and Enterprise Architecture.

The main drawback of employing our proposed method involves that when the percentage of the observed states increases, the performance decreases considerably. The main reason could be due to the communication infrastructure we are using to send the internal information between the different modules of the solution. We are using Internet Communications Engine (ICE) technology, but we have to consider other solutions in order to improve the results.

When the observability level of the states is 0, the performance results are good enough and better than a commercial tool (Enterprise Architecture of Sparx Systems [Sys15]) we used as generic automatic code generator. The commercial tool that is focused on generating software components for Real Time (RT) systems (SinelaboreRT [Mue18]) has better results. However, the difference is not big and the software components generated by that tool has not introspection and reflection ability at runtime.

Once the software components were generated, they were used in the use cases with the aim of clarify if we can detect errors and unexpected behaviours at runtime.

All the empirical evaluations demonstrated that when the observability level of the software components was the highest, all the errors and unexpected situations were detected before the output signals of the software components and systems were affected by the misbehaviour. Considering this, it can be assumed that the stated first hypothesis has been validated.

Second hypothesis

The second hypothesis is stated as follows:

> "The use of runtime monitoring and verification modules based on internal information in model terms of software components permits to enhance the safe behaviour of each of the software components in addition to the overall system composed by software components".

To test this hypothesis, we proposed two runtime monitoring and verification checkers that use internal information of the software components in model terms (UML-SM models' information). The first one is able to check the correctness of each software component at runtime and the second verifies system level safe properties at runtime.

The evaluation was performed with both use cases presented in Chapter 4.4: the Burner's use case was used to demonstrate the software level runtime monitoring and verification ability and the Train Control and Monitoring System (TCMS)'s use case was employed as a demonstrator at system level.

At software component level, the correct behaviour of the software component is inherent to the UML-SM formalism and the checker verifies the runtime information in model terms that provides each software component when it is going to perform a transition (current state, current event, next state,...). When developing the system level runtime monitoring and verification checker, we defined a specific language to define safe properties based on model term information provided by the software components that compound the system at runtime. The runtime monitoring and verification module was able to check the correctness of the system based on these safe properties defined by the safety engineer.

Both runtime modules (system level and software component level runtime monitoring and verification modules) were able to detect all the errors and unspecified situations when they were having information from all the states (100 % observability level).

In software component level, when decreasing the percentage of the observed level, the error detection ratio was also decreased. Specifically, on average, the percentage of the detected errors at software components level was 86.96% when 75% of the states were observed, 55.86% when 50% states were observed and 49.48% when 25% states were observed.

In case of the system level runtime monitoring and verification module, it was able to detect all the safety violations. In this case, we only observed the states that were involved in the safe properties. Thus, the observability level of the safety involved

states was 100%. Considering the performed empirical evaluation, we can conclude that the stated second hypothesis has been validated.

Third hypothesis

The third hypothesis is stated as follows:

> "The use of a runtime adaptation module permits to increase the availability and to enhance the safe behaviour level of software systems".

To test this hypothesis, we performed an evaluation with the use case presented in Chapter 4.4.1: the Burner's use case was used to demonstrate the runtime adaptation. Once the runtime monitoring and verification module detects an error or unexpected situation, it sends this information to the runtime safe adaptation manager. This manager, has the information about (1) the software component that need to perform the adaptation to its predefined safe-mode UML-SM and, depending on the last state of the software component, (2) which will be its initial state.

Once the experiment was performed, we confirmed that the software component was adapted to the predefined safe-mode UML-SM as expected. For doing that, we analyzed the information that the software component was providing at runtime to the runtime monitoring and verification system once the adaptation was performed. Thus, in case that an error or an unexpected situation was detected, the system continued working in a degraded safe mode and we did not have to stop the software component to restart it.

We can conclude that the availability and the safe behaviour level of the software components were enhanced. Considering the performed evaluation, we believe that the third hypothesis has been validated.

8.1.2 Limitations of the Proposed Solutions and the specific Implementation

This section discusses some of the limitations that the proposed solutions might have when applying them in practice.

One such limitation can be the selected software components' behaviour modeling notation. In our case, we selected Unified Modeling Language - State Machine (UML-SM) modeling due to its wide use in industry and as a tool, specifically we employed Papyrus [ecl18], since it is an open source, highly intuitive and quite powerful UML-SM modeling tool. In this design process, we add the information about the states to be observed at runtime using the Observability Profile we defined

for that. Nevertheless, some researchers have found limitations in Papyrus because it is not a tool that is broadly used in the industry for modeling software components behaviour by UML-SMs but as stated in [BB15] they are doing a considerable effort to be used as an industrial tool. Anyway, other industrial standard tools such as IBM Raphsody [Cen19] for modeling controllers by UML-SM notation will be considered as future line works.

It is important to highlight that all the RESCO methodology has been automatized using open source tools. However, we have to take into account the limitations of these tools and be ready to evolve our tools (Papyrus, the others are transparent for the users) to be able to handle new industrial-oriented tooling proposals.

In addition, we selected the C++ language as a basic language of our solution. The runtime monitoring, verification and adaptation modules have been implemented in C++ programming language and the communication between the software components and the runtime modules have been performed by the Internet Communications Engine (ICE) technology.

This decision was made due to, although the RESCO methodology is a platform independent one, when implementing the RESCO framework, we have developed the last step (Model To Text (M2T)) for resource limited and embedded systems. However, the software components that are checked by the runtime module could be developed with any other programming language. The only condition is that their internal information has to have a defined structure and in the current solution it has to be sent using the Internet Communications Engine (ICE) interface.

Other intercommunication interfaces could be used. IoT oriented solutions are also being employed to monitor remotely the controllers. In fact, in Mondragon University, in the context of the Productive 4.0 [eErp19] research project, we are conceptualizing the Safety Manager for the Arrowhead IoT platform. In this case, the idea is to use Arrowhead-compliant intercommunication between the monitored controlled and the Safety Manager.

We use a Model To Model (M2M) transformation language, ATL [ATL18] to transform the initial UML-SM model to a RESCO metamodel compliant model.

Once the RESCO compliant state machine is generated automatically, the last step that transforms the RESCO state machine model to code is performed by the Acceleo language by defining M2T rules. In this case, our specific implementation is focused on resource-limited systems and this last transformation is performed to C++ code.

Another limitation of the solution is related to the adaptation process. In the presented solution, the adaptations to be performed were previously designed as safe-mode UML-SMs. Notice that the designer have to design various adaptation

behaviour and the proposed solution is able to decide which of them select in the event of an error detection. Anyway, the safe adaptation process is not able to evolve or adapt to a design/behaviour decided at runtime. To overcome this problem, there are solutions, such as Process Mining based tools [MIn19], that evolve controllers behaviour extracting knowledge from event logs [VDAADM+11]. This working area has been detected as a future research line.

Lastly, we can add that the presented solution addresses the safe behaviour of CPSs but the generated framework and tools used during the defined methodology are not certified to be used in safety critical projects. As conclusion, we have to add that at this moment the development of safety critical systems (high SIL level systems) is beyond the scope of our solution.

8.2 Lessons Learned & Conclusions

This section summarizes lessons learned from the research carried out during this Ph.D. thesis. These lessons can be employed as a guidelines either, by researchers or industrial practitioners.

- *Reflective Unified Modeling Language - State Machine (UML-SM) based software components (RESCO) are appropriate to Runtime monitoring, verification and adaptation:* Advantages of software components generated by RESCO include (1) the logical/behaviour part of the controller and safe related part are orthogonal, (2) the provided runtime information in model terms helps on the Runtime Verification (RV) process and (3) the introspection and reflection ability they provide enables runtime adaptation. This is because the code generated for runtime introspection and observation work is independent to the behaviour and the states to be observed at runtime could be reconfigured at runtime. In addition, the runtime verification in models terms and runtime adaptation are possible because of the reflective characteristic of the solution. Moreover, the solution enables updating the observed states at runtime without having to recompile the software components code.

- *Having the behaviour of the software components in Unified Modeling Language - State Machine (UML-SM) model elements terms at runtime eases the runtime monitoring, verification and adaptation process:* In the last years, several research publications have shown the increasing trend of models@run.time approach for runtime adaptation in industry, especially in the autonomous system domain [CEG+14]. Highly reputed researchers have envisioned the use of models@run.times approach for runtime verification and adaptation [SZ16]. In this thesis we have focused on

the automatic generation of software components that follow the models@run.time approach. In this case, the use of internal status information of the software components in model element terms is highly important, especially because this characteristic enables the solution to detect errors before generating failures in the output signals.

- *The Safe Properties (SP) defined by software components' level model elements are effective notation at checking system level safe properties:* One of the conclusions of this thesis is that RV methods based on model elements terms are effective for enhancing the safe behaviour level of them. In fact, we used one case study and measured the use of these safe properties to detect errors and unexpected situations at runtime. The evaluation demonstrated that the generation of the runtime verification modules using these safe properties could be automatized and once generated, their effectiveness in detecting errors were very satisfactory.

- *The runtime adaptation is effective to increase the availability of them once the error or an unexpected situation is deteted:* We adapted the behaviour Unified Modeling Language - State Machine (UML-SM) model at runtime in case an error or undefined situation was detected at runtime. We have shown that runtime adaptation is not only good at increasing the operation availability software components but also at the safe behaviour level of software systems.

8.3 Perspectives and Future Work

In this section we summarize the short and medium term objectives to complement this work from three perspectives (i.e., industry transfer, application of the proposed methods in specific domains and further research).

8.3.1 Industry Transfer

The research performed by the Engineering School of Mondragon is industry oriented. We are currently contacting several industrial companies to present the results of the proposed methods in this thesis. Some presentation has been done to different industrial companies such as CAF Sginalling, and manufacturing related research centers (Ideko S.Coop) and industries (Danobat S.Coop). We plan to transfer them our methods during the Productive 4.0 project[1] to perform the automatic generation of software controllers based on reflective UML-SMs. Furthermore, in the scope of the Productive 4.0 project, we will continue developing our solution to provide it as

[1] https://productive40.eu/

Safety Manager service within the Arrowhead Framework. In addition, for doing the first examples and demonstrators of this last work, the Signify company, located in the Netherlands is interested and they have provided an initial use case.

8.3.2 Application of the Proposed Methods in Specific Domains

The proposed methods in this dissertation are generic for any system composed by software components such as CPSs or resource-limited embedded systems. However, it is worth mentioning that each system has its own particularities. For instance, if we take into account the runtime reflection and adaptation ability they provide, we may say that Autonomous System fits extremely well for the proposed methods in this dissertation. Nevertheless, we did not perform any experiment with these type of systems but we have pointed this topic as a future line.

8.3.3 Further Research

Further research as well as new developments can be performed to complement this work and we would like to expand the empirical evaluations using other real industrial cases and projects.

Performance of the RESCO framework and Communication interfaces

As for short-term future work, more measurements on the performance of the RESCO framework will be done. We will compare our solution with more commercial tools.

Regarding the second contribution of this thesis, we concluded that when the percentage of the observed states increases, the performance of the automatically generated software components decreases. The main reason of this is the use of the middleware Internet Communications Engine (ICE) to communicate the software components and the runtime monitoring, verification and adaptation module. In the future, we would like to adapt the communication middleware for other middleware (e.g., zeroMQ [Zer19]) as well as evolve the concept to be used in IoT oriented environments. In fact, in the context of the Productive 4.0 [eErp19] research project, we are planning to evolve the current communication interface to an Arrowhead compliant intercommunication.

Use of industrial standard tools

As highlighted in the limitations sections, despite Unified Modeling Language - State Machine (UML-SM) and the tooling used in the defined method do not pose important problems in our case for modeling software components that compounds the CPSs

and embedded systems, it is true that the modeling tools we used in the solution are not the ones that are standardized in the industry. As a mid-term future work, we plan to develop the solution considering other industrial standard tools such as IBM Raphsody [Cen19].

Adaptation: Dynamic adaptation

The current solution is only able to perform predefined runtime adaptations to known safe-mode UML-SMs. There is a further research opportunity in the area of runtime "dynamic" adaptation. This "dynamic" approach addresses the generation of new models at runtime in an autonomic way by techniques such as Machine Learning. Having this topic solved, the solution might be used in autonomous systems and uncertain environments.

To overcome this problem, inspired by the work in [MWPB16] we detected solutions, such as Process Mining (PM) based tools [MIn19], that evolve controllers behaviour extracting knowledge from event logs [VDAADM+11]. This working area has been detected as a future research line. Furthermore, with the derived techniques we would like to expand on the empirical evaluation by incorporating dynamic adaptations at runtime.

Bibliography

[A16] A. Acceleo. Technical report, https://www.eclipse.org/acceleo/, 2016.

[ABL+17] P. Asare, D. Broman, E.A. Lee, G. Prinsloo, M. Torngren, and Sunder S. A cyber physical systems-a concept map., 2017.

[Ada03] Paul Adamczyk. The anthology of the finite state machine design patterns. In *The 10th Conference on Pattern Languages of Programs*, 2003.

[AF17] Duncan Paul Attard and Adrian Francalanza. Trace partitioning and local monitoring for asynchronous components. In *International Conference on Software Engineering and Formal Methods*, pages 219–235. Springer, 2017.

[AGJ+14] Uwe Abmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin, and Mario Trapp. A reference architecture and roadmap for models@run.time systems. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Models@run.time*, pages 1–18, Cham, 2014. Springer International Publishing.

[AGR14] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Offline model-based testing and runtime monitoring of the sensor voting module. *Communications in Computer and Information Science*, 2014.

[AHJ+16] Reza Ahmadi, Nicolas Hili, Leo Jweda, Nondini Das, Suchita Ganesan, and Juergen Dingel. Run-time monitoring of a rover: Mde research with open source software and low-cost hardware. *Joint Proceedings of EduSymp and OSS4MDE*, 2016.

[ALG15] Hamoud Aljamaan, Timothy C Lethbridge, and Miguel A Garzón. Motl: a textual language for trace specification of state machines and associations. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pages 101–110. IBM Corp., 2015.

[Alh17] Pekka Alho. Service-based fault tolerance for cyber-physical systems. *Tampere University of Technology*, 2017.

[AS98] Ehab Al-Shaer. *Hierarchical Filtering-Based Monitoring Architecture For Large-Scale Distributed Systems*. PhD thesis, Old Dominion University, 1998.

[AT99] Jauhar Ali and Jiro Tanaka. Converting statecharts into java code. In *5th International Conference on Integrated Design and Process Technology (IDPT'99)*, page 42. Citeseer, 1999.

[AT04] Mikhail Auguston and Mark Trakhtenbrot. Run time monitoring of reactive system models. In *Proceedings of Second International Workshop on Dynamic Analysis WODA 2004, the 26th International Conference on Software Engineering ICSE*, pages 68–75, 2004.

[ATL18] ATL. Atl transformation language, 2018.

[Avi76] Algirdas Avizienis. Fault-tolerant systems. *IEEE Trans. Computers*, 25(12):1304–1312, 1976.

[AVW07] Deursen A.V., E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. In *International Workshop on Model-Driven Software Evolution*, 2007.

[Bar06] Franck Barbier. Mde-based design and implementation of autonomic software components. In *Cognitive Informatics, 2006. ICCI 2006. 5th IEEE International Conference on*, volume 1, pages 163–169. IEEE, 2006.

[Bar08] Franck Barbier. Supporting the uml state machine diagrams at runtime. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, pages 338–348, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BB15] R Barrett and F Bordeleau. 5 years of 'papyrusing'–migrating industrial development from a proprietary commercial tool to papyrus (invited presentation). In *Workshop on Open Source Software for Model Driven Engineering (OSS4MDE 2015)*, pages 3–12, 2015.

[BCLS17] Manel Brini, Paul Crubillé, Benjamin Lussier, and Walter Schön. 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.

[Bel13] Mokhtar Beldjehem. A granular hierarchical multiview metrics suite for statecharts quality. *Advances in Software Engineering*, Volume 2013:13, 2013.

[BGL93] Bernd Bruegge, Tim Gottschalk, and Bin Luo. A framework for dynamic program analyzers. *ACM SIGPLAN Notices*, 28(10):65–82, 1993.

[BHB09] C. Ballagny, N. Hameurlain, and F. Barbier. Mocas: A state-based component model for self-adaptation. In *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 206–215, Sept 2009.

[BHD17] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen 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*, pages 419–430. ACM, 2017.

[BJRT06] Michael Butler, Cliff Jones, Alexander Romanovsky, and Elena Troubitsyna. *Rigorous development of complex fault-Tolerant systems*, volume 4157. Springer, 2006.

[BLS] A Bauer, M Leucker, and C Schallhart. Model-based runtime analysis of reactive distributed systems. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC)*, pages 243–252.

[BSAN17] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. A survey on search-based model-driven engineering. *Automated Software Engineering*, 24(2):233–294, 2017.

[CEG+14] Betty H. C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, and Bernhard et. al Rumpe. Using models at runtime to address assurance for self-adaptive systems. In *Models@ run. time*, pages 101–136. Springer, 2014.

[Cen19] IBM Knowledge Center. Rhapsody uml statecharts, 2019.

[CF15] Ian Cassar and Adrian Francalanza. Runtime adaptation for actor systems. In *Runtime Verification*, pages 38–54. Springer, 2015.

[CF16] Ian Cassar and Adrian Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *International Conference on Integrated Formal Methods*, pages 176–192. Springer, 2016.

[CFAI18] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingolfsdottir. Developing theoretical foundations for runtime enforcement. *arXiv preprint arXiv:1804.08917*, 2018.

[CK10] Radu Calinescu and Shinji Kikuchi. Formal methods@ runtime. In *Monterey Workshop*, pages 122–135. Springer, 2010.

[CK13] Mikhail M. Chupilko and Alexander S. Kamkin. Runtime verification based on executable models: On-the-fly matching of timed traces. In Alexander K. Petrenko and Holger Schlingloff, editors, *MBT*, volume 111 of *EPTCS*, pages 67–81, 2013.

[Clu] The Safety-Critical Systems Club. Safety-critical systems and the accidents that don't happen.

[Del17] Jerker Delsing. *Iot automation: Arrowhead framework*. CRC Press, 2017.

[Der15] Mahdi Derakhshanmanesh. *Model-Integrating Software Components - Engineering Flexible Software Systems*. 01 2015.

[DGEE16] Mahdi Derakhshanmanesh, Marvin Grieger, Jürgen Ebert, and Gregor Engels. Thoughts on the evolution towards model-integrating software. *Softwaretechnik-Trends*, 36(3), 2016.

[DGH+16] Philip Daian, Dwight Guth, Chris Hathhorn, Yilong Li, Edgar Pek, Manasvi Saxena, Traian Florin Şerbănuţă, and Grigore Roşu. Runtime verification at work: A tutorial. In *International Conference on Runtime Verification*, pages 46–67. Springer, 2016.

[DGJ+16] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 36–43. ACM, 2016.

[DGR04] Nelly Delgado, Ann Q Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, December 2004.

[Dha17] Rohit Dhall. Designing graceful degradation in software systems. In *Proceedings of the Second International Conference on Research in Intelligent and Computing in Engineering*, volume 10, pages 171–179, 2017.

[DJC94] Michel Diaz, Guy Juanole, and Jean-Pierre Courtiat. Observer-a concept for formal on-line validation of distributed systems. *IEEE Trans. Software Eng.*, 1994.

[DLV12] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber-physical systems. volume Special issue on CPS, pages 13–28. IEEE, January 2012.

[Dön07] Andreas Huber Dönni. The boost statechart library. *Internet: http://www. boost. org/doc/libs/1*, 46(0):197, 2007.

[DPMBPL+17] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. Assessment of class mutation operators for c ++ with the mucpp mutation system. *Information and Software Technology*, 2017.

[DSE+11] Andrzej Dworak, M Sobczak, Felix Ehm, Wojciech Sliwinski, and P Charrue. Middleware trends and market leaders 2011. In *Conf. Proc.*, volume 111010, page FRBHMULT05, 2011.

[Dub13] Elena Dubrova. *Fault-tolerant design*. Springer, 2013.

[E+14] Sebastian Engell et al. Cyber-physical systems of systems—definition and core research and innovation areas. *Working Paper of the Support Action CPSoS*, 2014.

[ea02] O. Spinczyk et al. Aspectc++: An aspect-oriented extension to c++. In *Conferences in Research and Practice in Information Technology*, 2002.

[ea04] Moonzoo Kim et al. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 2004.

[ea11a] Chang-Guo Guo et al. A generic software monitoring model and feature analysis. *Journal of Software*, 2011.

[ea11b] Paolo Arcaini et al. Coma: Conformance monitoring of java programs by abstract state machines. In *International Conference on Runtime Verification*, 2011.

[ea16] Philip Daian et al. Runtime verification at work: A tutorial. In *International Conference on Runtime Verification*, 2016.

[ecl18] eclipse. Papyrus-rt, 2018.

[eErp19] Productive 4.0 : european ECSEL research project. Productive 4.0: Opening the gates to the digital future, 2019.

[EFL17] William Excoffon, Jean-Charles Fabre, and Michaël Lauer. Analysis of adaptive fault tolerance for resilient computing. In *Dependable Computing Conference (EDCC), 2017 13th European*, pages 50–57. IEEE, 2017.

[EHFJ18] Antoine El-Hokayem, Yliès Falcone, and Mohamad Jaber. Modularizing behavioral and architectural crosscutting concerns in formal component-based systems application to the behavior interaction priority framework. *Journal of logical and algebraic methods in programming*, 2018.

[EMS+08] X. Elkorobarrutia, M. Muxika, G. Sagardui, F. Barbier, and X. Aretxandieta. A framework for statechart based component

reconfiguration. In *Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems (ease 2008)*, pages 37–45, March 2008.

[FCSS17] Nicolas Ferry, Franck Chauvel, Hui Song, and Arnor Solberg. Towards meta-adaptation of dynamic adaptive systems with models@ runtime. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, Port, Portugal, February 19-21, 2017*, 2017.

[FJ17] Yliès Falcone and Mohamad Jaber. Fully automated runtime enforcement of component-based systems with formal and sound recovery. *STTT*, 19(3):341–365, 2017.

[FLPV13] John S Fitzgerald, Peter Gorm Larsen, Ken G Pierce, and Marcel Henri Gerard Verhoef. A formal approach to collaborative modelling and co-simulation for embedded systems. *Mathematical Structures in Computer Science*, 23(4):726–750, 2013.

[FMFR11] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[FR98] Luciane Lamour Ferreira and Cecília Mary Fischer Rubira. Reflective design patterns to implement fault tolerance. In *OOPSLA Workshop on Reflective Programming*, volume 199, 1998.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[GAM17] Hassan Gomaa, Emad Albassam, and Daniel A Menascé. Runtime software architectural models for adaptation, recovery and evolution. In *MODELS (Satellite Events)*, pages 193–200, 2017.

[Gar10] David Garlan. Software engineering in an uncertain world. In *FoSER*, 2010.

[GCH+04] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[GMP03] Marcela Genero, David Miranda, and Mario Piattini. Defining metrics for uml statechart diagrams in a methodological way. In *International Conference on Conceptual Modeling*, pages 118–128. Springer, 2003.

[GMRE+16] Elena Gomez-Martinez, Ricardo J Rodríguez, Clara B Earle, Leire Elorza, and Miren Illarramendi. A methodology for model-based verification of safety contracts and performance requirements. *Journal of Risk And Reliability*, 2016.

[Gro17] OMG (Object Management Group). Precise semantics of uml state machines (pssm). Technical report, OMG, 2017.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32. ACM, 2002.

[GT05] Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.

[HTI97] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[IBM] IBM. Ibm-rhapsody.

[IEC10] IEC. *61508:Functional safety of electrical/electronic/programmable electronic safety related systems*, 2010.

[IEE08] IEEE. Recommended practice on software reliability. 2008.

[IEES17] Miren Illarramendi, Leire Etxeberria, Xabier Elkorobarrutia, and Goiuria Sagardui. Increasing dependability in safety critical cpss using reflective statecharts. In *International Conference on Computer Safety, Reliability, and Security*, pages 114–126. Springer, 2017.

[ISO12] ISO. *ISO26262:Road vehicles- Functional Safety*, 2012.

[ite18] itemis. Yakindu statechart tools, 2018.

[Jam12] Pooyan Jamshidi. Computational reflection: Smith's theory of computation and causal relations, 2012.

[JJ94] Pankaj Jalote and P Jalote. *Fault tolerance in distributed systems*. PTR Prentice Hall Englewood Cliffs, New Jersey, 1994.

[JK10] Daniel Jackson and Eunsuk Kang. Separation of concerns for dependable software design. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 173–176. ACM, 2010.

[JSZ10] Mark James, Paul Springer, and Hans Zima. Adaptive fault tolerance for many-core based space-borne computing. In *European Conference on Parallel Processing*, pages 260–274. Springer, 2010.

[KDRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[KFK14] Aaron Kane, Thomas Fuhrman, and Philip 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*, pages 148–155. IEEE, 2014.

[Lap92] Jean-Claude Laprie. Dependability: Basic concepts and terminology. In *Dependability: Basic Concepts and Terminology*, pages 3–245. Springer, 1992.

[Lib] Libfiu. Libfiu: faultinjection in userspace.

[Mae87a] Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987.

[Mae87b] Pattie Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987.

[MAJJ08] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. In *RAM*, pages 736–742, 2008.

[MBJ+09] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run. time to support dynamic adaptation. *Computer*, 42(10), 2009.

[MGE] Monitoring oriented programming.

[MIn19] PM: Process MIning. Process mining, 2019.

[MSS93] Masoud Mansouri-Samani and Morris Sloman. Monitoring distributed systems. *IEEE network*, 7(6):20–30, 1993.

[Mue18] Peter Mueller. sinelaborert. Technical report, 2018.

[MWPB16] Alexandra Mazak, Manuel Wimmer, and Polina Patsuk-Bösch. Execution-based model profiling. In *International Symposium on Data-Driven Process Discovery and Analysis*, pages 37–52. Springer, 2016.

[MZ16] Pieter J Mosterman and Justyna Zander. Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems. *Software & systems modeling*, 15(1):5–16, 2016.

[np15] nSafeCer project. Safety Certification of Software-Intensive Systems with Reusable Components. Project Grant Agreement n^o 295373, 2015. http://safecer.eu/, (2015, accessed 15 February 2015).

[NYA+13] Kunming Nie, Tao Yue, Shaukat Ali, Li Zhang, and Zhiqiang Fan. Constraints: The core of supporting automated product configuration of cyber-physical systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 370–387. Springer, 2013.

[OMG17] OMG. Precise semantics of uml state machines (pssm), v1.0, beta 1. ptc/2017-04-01 (http://www.omg.org/spec/PSSM/About-PSSM/), 2017.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE Computer Society, 1998.

[Pap19] Papyrus. Papyrus, 2019.

[PG05] Antonis Paschalis and Dimitris Gizopoulos. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *IEEE Transactions on Computer-aided design of integrated circuits and systems*, 24(1):88–99, 2005.

[PM03] Gergely Pintér and István Majzik. Program code generation based on uml statechart models. *Periodica Polytechnica Electrical Engineering*, 47(3-4):187–204, 2003.

[PM05] Gergely Pintér and István Majzik. Runtime verification of statechart implementations. In *Architecting Dependable Systems III*, pages 148–172. Springer, 2005.

[Pra16] Prashant Kaliram Pradip. Commissioning and performance analysis of whispergen stirling engine. Technical report, University of Windsor, 2016.

[PRGL17] Van Cam Pham, Ansgar Radermacher, Sébastien Gérard, and Shuai Li. Complete code generation from uml state machine. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 208–219. INSTICC, SciTePress, 2017.

[Pul01] Laura L. Pullum. *Software Fault Tolerance: Techniques and Implementation*. 2001.

[PW16] Mauro Pezzé and Jochen Wuttke. Model-driven generation of runtime checks for system properties. *Int. J. Softw. Tools Technol. Transf.*, 18(1):1–19, February 2016.

[Qua18] Quantum. Quantum platform in c++, 2018.

[RI16] Alexander Romanovsky and Fuyuki Ishikawa. *Trustworthy cyber-physical systems engineering*. CRC Press, 2016.

[RKFTF03] Juan Carlos Ruiz, M-O Killijian, J-C Fabre, and P Thvenod-Fosse. Reflective fault-tolerant systems: From experience to challenges. *IEEE Transactions on Computers*, 52(2):237–254, 2003.

[Saf13] Safecer. Deliverable d2.1.4 & d4.1.2: Standards-oriented, domain-specific aspects in reusing software in safecer domains. http://www.safecer.eu/images/pdf/pSafeCer_Deliverable_D4_1_2.pdf, March 2013.

[Sam02] Miro Samek. *Practical statecharts in C/C++: Quantum programming for embedded systems*. CRC Press, 2002.

[Smi82] Brian Cantwell Smith. *Procedural reflection in programming languages*. PhD thesis, Massachusetts Institute of Technology, 1982.

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35. ACM, 1984.

[Som07] Ian Sommerville. *Software Engineering*. 2007.

[SSL+14] Mehrdad Saadatmand, Detlef Scholle, Cheuk Wing Leung, Sebastian Ullström, and Joanna Fredriksson Larsson. Runtime verification of state machines and defect localization applying model-based testing. In *Proceedings of the WICSA 2014 Companion Volume*, page 6. ACM, 2014.

[Sto96] Neil R Storey. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[Sys15] Sparx System. Enterprise architecture v11. Technical report, Sparx Systems, 2015.

[SZ16] Michael Szvetits and Uwe Zdun. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Software & Systems Modeling*, 15(1):31–69, 2016.

[TAFJ07] Mario Trapp, Rasmus Adler, Marc Förster, and Janosch Junger. Runtime adaptation in safety-critical automotive systems. *Software Engineering*, pages 1–8, 2007.

[TFCB90] Jeffrey 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*, 16(8):897–916, 1990.

[VDAADM+11] Wil Van Der Aalst, Arya Adriansyah, Ana Karla Alves De Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter Van Den Brand, Ronald Brandtjen, Joos Buijs, et al. Process mining manifesto. In *International Conference on Business Process Management*, pages 169–194. Springer, 2011.

[VGB00] Jilles Van Gurp and Jan Bosch. On the implementation of finite state machines. *Variability in Software Systems The Key to Software Reuse*, page 45, 2000.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous monitoring of software services: Design and application of the kieker framework. Research report, Kiel University, November 2009.

[VK04] Vijay Vaishnavi and William Kuechler. Design research in information systems. 2004.

[VRG+16] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, Klaus Seyerlehner, Stefan Wallner, and Helmut Zeisel. Reminds: A flexible runtime monitoring framework for systems of systems. *Journal of Systems and Software*, 112:123–136, 2016.

[VSG10] Thomas Vogel, Andreas Seibel, and Holger Giese. The role of models and megamodels at runtime. In *International Conference on Model Driven Engineering Languages and Systems*, pages 224–238. Springer, 2010.

[Wea08] Rob Weaver. Systematic and random failure, 2008.

[Wol09] Wayne H Wolf. Cyber-physical systems. *IEEE Computer*, 42(3):88–89, 2009.

[Zer19] ZeroMQ. Zeromq, 2019.

[ZOKR06] Yuhong Zhao, Simon Oberthür, Martin Kardos, and Franz-Josef Rammig. Model-based runtime verification framework for self-optimizing systems. *Electronic Notes in Theoretical Computer Science*, 144(4):125–145, 2006.