Runtime Contracts Checker: Increasing Robustness of Component-Based Software Systems

IOP Conf. Series: Materials Science and Engineering **575** (2019) 012006 doi:10.1088/1757-899X/575/1/012006

Runtime Contracts Checker: Increasing Robustness of Component-Based Software Systems

M Illarramendi, L Etxeberria, X Elkorobarrutia and G Sagardui

Mondragon Goi Eskola Politeknikoa S.Coop, Loramendi 4, Mondragon, Spain

millarramendi@mondragon.edu

Abstract. Software Systems are becoming increasingly complex leading to new Validation &Verification challenges. Model checking and testing techniques are used at development time while runtime verification aims to verify that a system satisfies a given property at runtime. This second technique complements the first one. This paper presents a runtime contract checker (RCC) which checks a component-based software system's contracts defined at design phase. We address embedded systems whose software components are designed by Unified Modelling Language-State Machines (UML-SM) and their internal information can be observable in terms of model elements at runtime. Our previous research work, CRESCO (C++ REflective State-Machines based observable software COmponents) framework, generates software components that provide this observability. The checker uses software components' internal status information to check system level safety contracts. The checker detects when a system contract is violated and starts a safeStop process to prevent the hazardous scenario. Thus, the robustness of the system is increased.

1. Introduction

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

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

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

Current runtime checking solutions as shown in Figure 1, are specified at different abstraction levels: system, component, class, method or statement. In most of the approaches, the checking is performed at the same level, that is, only by checking system level properties could system misbehaviour be detected. Nevertheless, component or class level properties can give valuable information in detecting system level problems and undesired emergent behaviour.

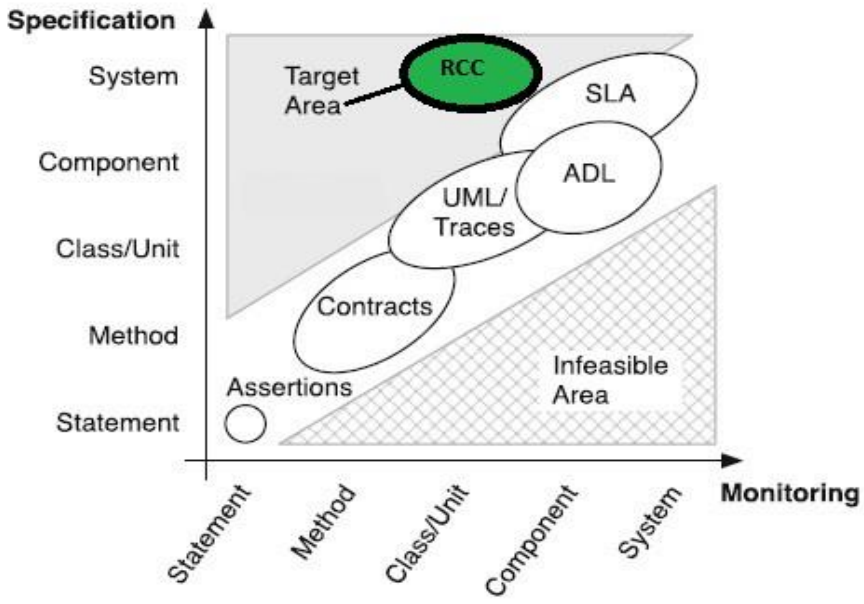

Figure 1. Abstraction level of specifications vs. runtime monitoring abstraction levels based on [3].

Most runtime software checkers require modifying the source code of the observed system by instrumented code. However, it is desirable that runtime checkers for testing safety properties of the systems should be isolated from the target system to minimize any disruption of the system being tested [4].

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

Our approach (RCC) is classified in the target area of Figure 1. Specification is made using system level safety contracts and the monitored information is the internal status of the software components of the system. It verifies the system level safety contracts based on the internal status information received from each of the systems' software components' observable states at runtime. These contracts are checked when the internal status of the system components is going to perform a state transition. In this sense, our software checker shares the advantages of hardware checkers because we can detect the fault before a change in components' state happens. RCC is developed in an isolated way and therefore there is no need to change any code, nor the design of the system.

Main contributions of RCC are:

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

It is worth noting that the RCC has been developed taking into account a previous work called CRESCO framework [5]. This framework is able to generate reflective software components that provide information about the internal status of the software components in terms of UML-SM elements at runtime. The contracts that the RCC checks are defined using the information provided by these software components at runtime. Nonetheless, the RCC solution can be used independently of CRESCO software components. In any case, the software components we are addressing have to fulfil 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.

RCC also requires consistent snapshots of the system and, to this end, for example, the observed system has to be a synchronous one or the system's messages must be causally ordered.
Section II introduces the Runtime Contract Checker. In Section III related work is detailed and, finally, section IV ends the paper with our conclusions and future lines of action.

2. Runtime contract checker

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 errors and hazardous scenarios of component-based software systems at runtime. To this end, system level runtime safety contracts, based on internal status information of the software components, are defined and the specific checker is generated automatically.
This section presents the process of generating the RCC, the architecture of the contracts checking system, the internal status information to be checked by the RCC and how the runtime state-based safety contracts are specified.

2.1. Process for defining Safety Contracts and generating the RCC

The process to generate the RCC 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 safety contracts starts. This process has four steps (see Figure 2):

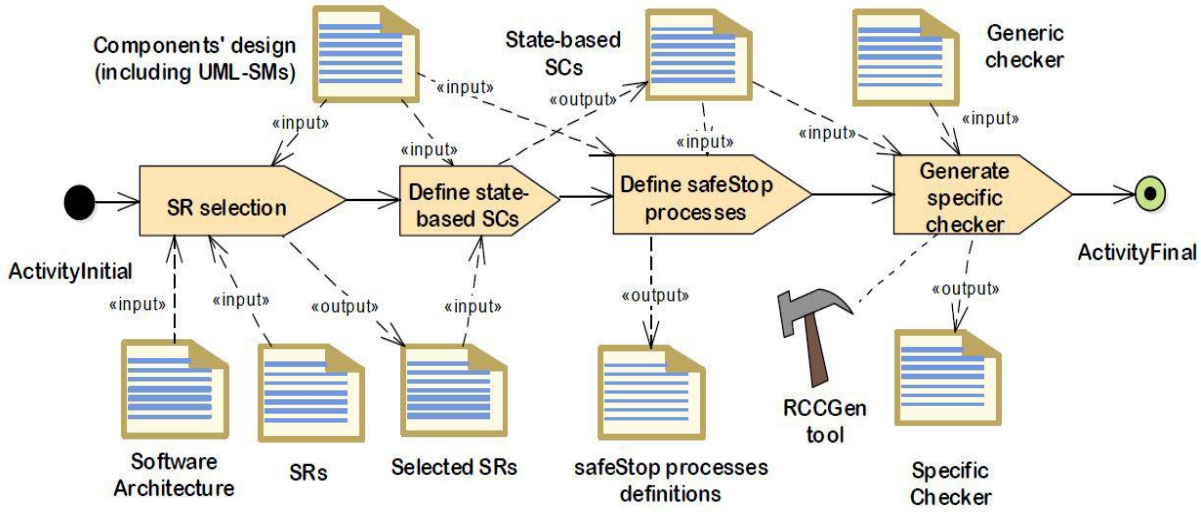

Figure 2. Process for defining state-based safety contracts.

- Select safety requirements (SR) to be used at runtime verification. Not all safety 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 safety requirements (SR_i).
- Define state-based safety contracts (SC) based on selected safety requirements.
- Define safeStop processes to be launched in case safety contracts are not fulfilled at runtime (a process for each safety contract).
- Generate the checker: RCCGen tool transforms the safety contracts to RCC Code (checker, in C++) automatically. RCCGen uses a generic checker as a basis and adds the specific safety contracts to the checker.

After these steps, the development continues with the implementation, verification and validation and integration steps.

2.2. RCC Architecture

In Figure 3, the overview of component-based software system's safety contract checking architecture is shown. The software components of the system are modelled by UML-SM and the runtime contract checker has two main components: the runtime contracts checker (RCC checker) and the safeStop processes manager.

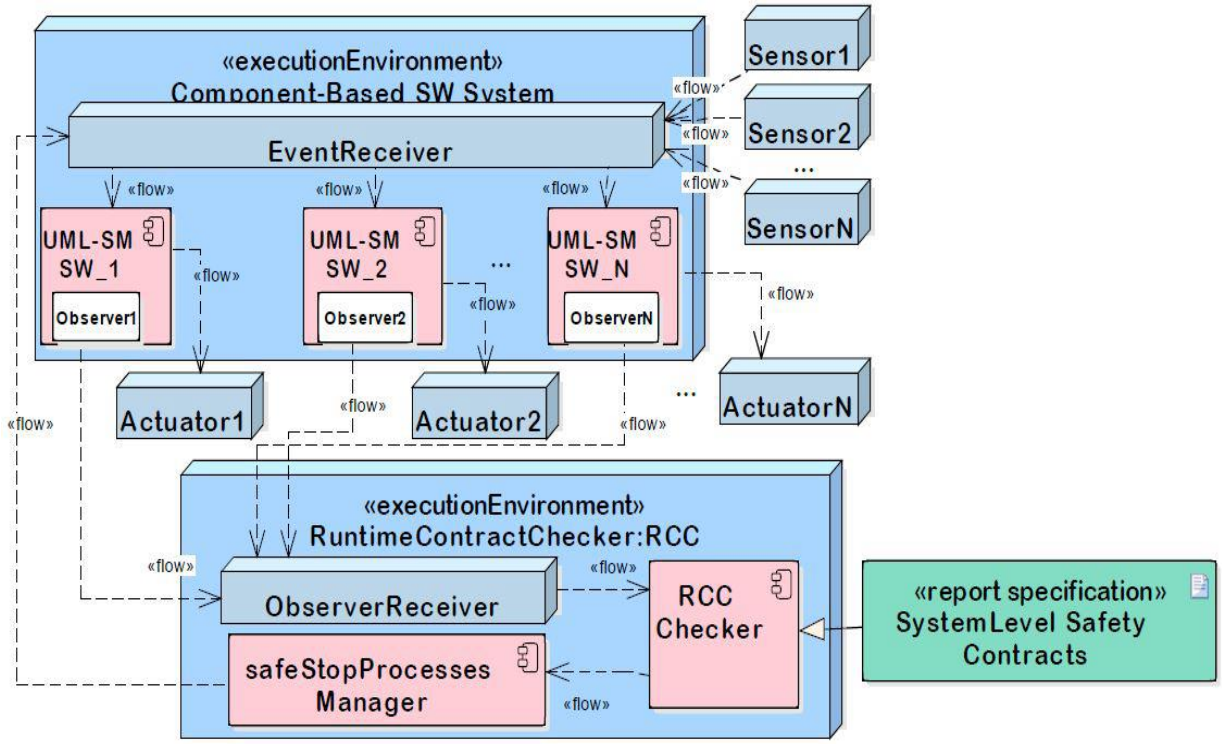

Figure 3 General Architecture of the Contracts Checking System.

RCC behaviour at runtime: The RCC 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 observers of these software components send their internal status information before performing a state transition. The RCC checker compares this information with the system level contracts. If system contracts are fulfilled, the system status information is updated and it waits for new updates.

In the event of the contracts not being fulfilled, the RCC checker notifies that circumstance to the safeStop processes manager. This manager starts the predefined safeStop process. This process sends safeStop events to the software components that are involved in the safeStop process. The process is dependent on the system and it is not in the scope of our work.

2.3. Internal status information of the monitored software components

This work addresses software components that are modelled by UML-SM and systems that are composed of several different of these components. Many tools can generate UML-SM based software components, one of which is the CRESCO framework. In addition, CRESCO creates software components that can reflect at runtime the UML-SM model from which they come.

For runtime checking, some decisions must be taken about what to observe. The RCC needs to receive runtime information from the software components of the system to be checked, in particular from the states annotated as observable.

2.4. Specification using Safety Contracts

We need to prove that the composite implementation of the system guarantees system level contracts at runtime. In our case, a safety contract specifies properties related to the internal behavior of the software components that are part of the system in terms of their UML-SM model (active states). That is what we call a state-based safety contract.

In our approach a system (Sys) may be composed of subsystems (that could be further decomposed) and primitive components (C) that cannot be further decomposed. Furthermore, the primitive components have a behavior specified using a state machine.

Safety requirement will be allocated to the system and a safety requirement may be satisfied by a safety contract or a set of safety contracts. A safety contract will be related to the states of the components involved in the contract. A grammar with regular expressions is used to specify what to check.

contract := *constraint* | *timedConstraint;*
constraint := *condition* **implies** *condition;*
condition := *activeState* | **not** *condition* | *(condition)* |
condition **or** *condition* |
condition **and** *condition*;
timedConstraint := *constraint* **in** *timeUnits*;
activeState := *ComponentName:StateName;*

With this grammar, it is possible to specify constraints 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}.

CTraction:SOn **implies** *CDoor:SClosed*

3. Related work

Creating a runtime checker for checking system level properties is an important research problem. Existing approaches to runtime verification require specifications that not only define the property to monitor, but also contain details of the implementation, sometimes even requiring the implementation to add special variables or methods for monitoring.

In [6], 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, among others CoMA, RV-MONITOR and AspectC++, BEE++, DN-Rover, HiFi etc. The abstraction level of specification and monitoring of the above solutions is the same: specifications and the monitor's checking properties are at the component or class level.

LuMiNous [3] framework's target is the same area as the checker presented in this paper: system level specifications are checked 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 solution is for Java (AspectJ based solution), which is not suitable for embedded systems.

4. Conclusion and future work

This paper presents an RCC that is automatically generated and which considers system level specific safety contracts to be verified at runtime.

The main conclusions are that the checker is able to detect all the safety related faults at runtime, thereby increasing the robustness and reliability of the system. As it uses components' internal

information, it has the ability to prevent faulty scenarios before having changed the system output signals. This is a benefit compared with software monitors that can only check the output signals.
Our last conclusion is that the process to implement and generate the RCC is cost-effective as it is generated automatically. The safety engineer has to define just the safety contracts (following the defined grammar to this end) and the safeStop processes. The rest of the process is automatic. The software developer of the software components only considers the functional aspects of the system.
As future research lines we might consider the following topics:

- Expand the empirical evaluation by using realistic use cases in different industrial domains and projects.
- Provide a tool or mechanism that will assist in adding the specific part of the safeStop processes systematically to the safeStop process manager.

Acknowledgments
The project has been developed by the Embedded System Group of MGEP and supported by the Department of Education, Universities and Research of the Basque Government under the projects Ikerketa Taldeak (Grupo de Sistemas Embebidos) and TEKINTZE (Elkartek 2018) and the European H2020 research and innovation programme, ECSEL Joint Undertaking, and National Funding Authorities from 19 involved countries under the project Productive 4.0 with grant agreement no. GAP-737459 - 999978918.

References
[1] IEC 2010 *61508: Functional safety of electrical/electronic/programmable electronic*
[2] B. Schön, M. Brini and P. Crubillé 2017 *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
[3] J. Wuttke and M. Pezzé 2016 *Model-driven generation of runtime checks for system properties* Int J Softw Tools Technol Transfer.
[4] P. Koopman, A. Kane and T. Fuhrman 2014 *Monitor Based Oracles for Cyber-Physical System Testing* in Dependable Systems and Networks: Practical Experience Report
[5] M. Illarramendi, L. Etxeberria, X. Elkorobarrutia and G. Sagardui 2017 *Increasing Dependability in Safety CPSs Using Reflective Statecharts* in Computer Safety, Reliability, and Security
[6] G. Chang-Guo, J. Zhu and X.-L. Li 2011 *A Generic Software Monitoring Model and Feature Analysis* Journal of Software, vol 6, no 3, pp 395-403