Nermin Kajtazovic, Dipl.-Ing. BSc

A Component-based Approach for Managing Changes in the Engineering of Safety-critical Embedded Systems

DOCTORAL THESIS

to achieve the university degree of

Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Em. Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhold Weiß
Dipl.-Ing. Dr.techn. Christian Kreiner

Institute for Technical Informatics

Graz, December 2014

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date

Signature

Kurzfassung

Die Entwicklung eingebetteter Systeme ist ein komplexes Unterfangen mit vielfältigen Herausforderungen. Gründe dafür sind etwa ständig wachsende Anforderungen des Marktes, wie Produktivität, Performance und Qualität; weiters ist die Entwicklung auch organisatorisch aufwändig, etwa durch Anforderungen infolge einzuhaltender Gesetze und Standards, aber auch durch das Management und die Integration komplexer Zulieferketten in vielen Disziplinen im Zuge der Entwicklung des Gesamtsystems. Als Folge werden eingebettete Systeme immer komplexer auch in Hinsicht auf Ihre Funktionalität. Eine besondere Herausforderung stellt hier die Software dar, weil immer mehr Funktionen software-technisch realisiert werden – die jährliche Wachstumsrate der Softwarekomplexität beträgt je nach Anwendungsdomäne 10% bis 30%. Dieser Komplexitätstrend stellt auch für sicherheitskritische Systeme die größte Herausforderung dar. Im Unterschied zu anderen eingebetteten Systemen werden hier höhere Ansprüche an Sorgfalt, Kontrolle und Strenge in der Entwicklung gestellt, um zu vermeiden, dass gefährliche Fehler auftreten und potentiell Schäden verursachen (z.B. Verletzung von Menschen, die mit solchen Systemen operieren). In den letzten Jahren hat die steigende Softwarekomplexität bedauerlicherweise auch dafür gesorgt, dass auch mehr Fehler während des Betriebs sicherheitskritischer Systeme auftreten. Die Folgen sind ungeplante Rückruf- bzw. Wartungsaktivitäten. Allerdings ist die Durchführung derartiger Aktivitäten nicht trivial, weil oft adäquate und kosteneffektive Methoden im Engineering dieser Systeme fehlen.

Diese Arbeit stellt eine Methode vor, mit der für sicherheitskritische Systeme die erwähnten Wartungsvorgänge während der Einsatzphase unterstützt werden. Aus technischer Perspektive werden solche Reparaturen durch den Austausch von Softwarekomponenten im Rahmen der Softwarearchitektur realisiert. Ein wesentliches Merkmal ist hier, dass auch relevante Standards zur funktionalen Sicherheit zur Anwendung kommen, die den gesamten Lebenszyklus dieser Systeme regulieren. Um dies zu erreichen, stellt die Arbeit folgende Beiträge vor: (i) eine Methode, die auf architektonischer Ebene die benötigten Softwareänderungen erlaubt, um die Softwarewartung technisch zu ermöglichen – dazu wird das Component-based Software Engineering (CBSE) Paradigma verwendet; (ii) eine entsprechende Modellierungs- und Analyseunterstützung um sicherzustellen, dass die Auswirkungen der Änderungen die Systemintegrität nicht beeinträchtigen – hier wird das Prinzip von Contract-based Design (CBD) eingesetzt; und (iii) die Anwendung von Regeln der IEC 61508, einem generischen Standard für Funktionale Sicherheit von Elektrischen, Elektronischen und Programmierbaren Systemen, um Änderungen standard-konform unter Aufrechterhaltung der Systemintegrität für Funktionale Sicherheit durchführen zu können.

Durch einen ausgewogenen Trade-Off zwischen möglichen Änderungen, Analyseunterstützung für Auswirkungen solcher Änderungen, sowie den Regeln des Standards wird eine kosteneffektive Durchführung von Wartungsvorgängen auf Softwareebene ermöglicht, d.h. ohne eine neuerliche Sicherheitszertifizierung infolge der Änderung. Dieses Konzept wurde in dieser Arbeit für eine Steuerungsarchitektur für Wasserkraftwerke realisiert, ist aber insbesondere auch anwendbar in Domänen, die sicherheitsgerichtete Produkte in Massenfertigung herstellen, wie zum Beispiel die Fahrzeugindustrie oder Medizintechnik.

Abstract

The development of today's embedded systems is becoming a cumbersome and challenging task for engineers in many application fields. These challenges originate, on the one side, from the increasing market demands on various system attributes such as productivity, performance and products quality, and on the other side, from the organizational reasons, for instance the need to follow certain regulations, standards, to maintain the development in the supply chain, and so forth. In response, embedded systems are becoming complex in terms of implemented functions, their heterogeneity, and their inter-connections. For example, the average increase rate of the software complexity in embedded systems engineering lies between 10% and 30% per year. This complexity trend is currently also a major problem facing the engineering of safety-critical systems. Safety-critical systems in contrast to general embedded systems require more rigorous and controlled development, in order to meet quality requirements, and in this way to prevent causing failures that might lead to severe consequences, for example to pose threats to humans operating with those systems. In response to the aforementioned complexity trend, more and more failures, in particular software-related failures, are being manifested in the operation of these systems, and therefore repairs and changes have to be performed at an increased rate. However, safety engineering generally lacks methods and procedures to address such repairs in a systematic and cost-effective way.

This thesis introduces an approach to perform repairs on safety-critical systems, by allowing to replace faulty software units (i.e., software components) in their operation and maintenance phase. A very important characteristic here is that repairs and necessary procedures are aligned with safety standards, which regulate the safety lifecycle, so that the system integrity can always be maintained. To this end, the thesis proposes the following three contributions: (i) an architectural support to perform repairs by allowing to introduce changes in software – here, a Component-based Software Engineering (CBSE) has been utilized to make performing changes possible; (ii) a modeling and analysis support to conduct the impact of changes – a Contract-based Design (CBD) paradigm is applied to ensure that the system integrity is not compromised due to incorporated changes, and (iii) the alignment with the safety standards – corresponding processes, requirements, measures and techniques defined in standards are followed. To this end, the IEC 61508, a generic standard applied in the industry sectors, is used as a reference.

The main outcome here is that specific type of changes can be maintained in a cost-effective way, in particular, without introducing costs required for the re-certification. The approach proposed here has been realized and applied to support managing changes in controllers for the hydro power plants, in particular, for maintaining the control software. However, the contributions it provides can also bring benefits to other application fields, especially to fields that have mass production in focus and thus have rather frequent repair demands, as it is the case with the automotive sector and biomedical engineering for example.

Acknowledgements

This thesis has been carried out at the Institute for Technical Informatics, Graz University of Technology, in cooperation with the industrial partner Andritz Hydro GmbH from Vienna, within the scope of the HIPASE project.

I would like to use this opportunity to mention some persons, who substantially contributed to the success of my work and supported me during my PhD studies. First of all, I would like to thank my supervisor Prof. Reinhold Weiß and my mentor Christian Kreiner for their great cooperation and support. Prof. Weiß helped me in solving many issues in the beginning of the project from the scientific point of view and in finding the right direction towards my research problem. He was open for many discussions and he guided me with a lot of valuable suggestions. Especially, I appreciate his readiness to share his skills and knowledge that helped me a lot to produce the scientific work. I express many thanks to Christian Kreiner, who offered me the PhD position, and motivated me to focus my work on component-based systems and safety engineering. He continuously supported me with a number of discussions and workshops that helped me to easily get in touch with our industrial partner. One of the most notable lessons I could learn from him is the way on how to apply some concepts from the academia in the industry. Besides, from the Christian's special way of mentoring I could learn how to deal with stress situations and how stay on the right track all the time. I am very delighted that I had the opportunity to work with Christian and Prof. Weiß.

Next, I would like to thank Rudolf Neuner from Andritz Hydro, who was my first contact person within the company. He spent a lot of time with me during our workshops, and was always open for my suggestions and ideas. I also thank other colleagues from the company, including Thomas Kirchmair, Michael Cech, Edwin Fruehwirth, Norbert Lange, Jari Fritz, and Basha Erion. With their support and quite intensive communication, we were able to realize some new ideas and concepts and in response achieve some important milestones in the project. I also want to express many thanks to the Andriz Hydro company for the financial support.

For their great support and cooperation I would like to thank my colleagues from the institute, Christopher Preschern, Andrea Höller, and Tobias Rauter, i.e., the A-Team. We spent a lot of time together in the project, participated in diverse workshops organized with our industrial partner, and supported each other. I was able to share the great experience with them. I also express special thanks to my students Johannes Iber, Thomas Hinterkircher and Mathias Blum for their great contributions in implementing diverse features for the HIPASE project.

In the end, I would like to express acknowledgement to my parents, my brothers, and other members of the family for their continuous support during my education. In particular, I address my special thanks to my girlfriend Belgin. She really was my largest motivation generator. In many stressful situation, especially in the beginning, she believed in me more than I did, and motivated me during the whole PhD time. With her, I always had someone with whom I could share my experience and frustration.
Thank you all again, and for Belgin, *çok teşekkürler, cac*!

Graz, December 2014 Nermin Kajtazovic

Extended Abstract

Motivation

Development of today's embedded systems is becoming a cumbersome and challenging task for engineers in many application fields. Due to impulsive market demands on improvements of various system attributes such as productivity, quality and performance, the complexity of embedded systems in terms of implemented functions, their inter-connections, and heterogeneity is continuously growing. For instance, when just considering the functional complexity, the average growth rate of embedded software lies between 10% and 30% per year (depending on concrete field) [EJ09].

This complexity trend has even more implications on developing safety-critical systems. Safety-critical systems require more rigorous and controlled development, since they operate in environments in which often humans, equipment or environment are in some way related to the system, in particular, they depend on correct functioning of such a system. Failures in the hardware, software or mechanics may therefore lead to severe consequences, including potential threats to human life, or damages of equipment or environment. To ensure that occurring of such failures is unlikely, the general principle of developing this class of systems is to align systems and safety engineering, within a complete development lifecycle in a systematic way. This alignment is mainly provided by safety standards, which represent a common knowledge base for engineers and for safety assessors that aids them to achieve the desired system's quality (i.e., the safety integrity in the notation of standards), and to provide a means to guarantee that certain integrity level has been achieved respectively.

The consequences of the aforementioned challenges in systems and safety engineering are manifold. On the one hand, it becomes increasingly difficult to verify and to validate the system, in particular the predominant software functions, i.e., to inspect their behaviour with respect to functional requirements, but also, to verify the conformance to non-functional requirements that set constraints on various system quality attributes such as safety, security, and performance for example, which is in practice often only partly achievable. On the other hand, considered safety standards are often missing some technical concepts and guidelines needed for the practical realization. For example, systems engineering is currently massively based on reuse of existing, pre-fabricated components, in form of hardware, software or complete devices, and the system is basically built by integrating those components. To date, only few guidelines in the context of safety standards have been developed, which provide a systematic way on how to design such components for reuse and how integrate them together, to build so called composite systems [GS05]. However, the concrete techniques that in particular take into account non-functional requirements while building such composite systems are not yet mature enough and can be rarely found in the practice. Moreover, the topic of the compositional analysis and

reasoning is still an open research field.

In response to issues summarized here, more and more faults are remaining unidentified in the development, and represent a potential cause for system failures in the operation. According to some recent studies made in fields of automotive and biomedical engineering, the major cause for recalls of products in related fields are systematic faults in software [Qi 14,AIKR13]. In addition to challenges posed to development, this trend raises an emerging problem to maintainability of safety-critical systems – repairing a system from such faults becomes a cumbersome and cost-intensive task, in particular, because procedures and changes required for repairs have to be communicated with assessors, and, according to safety standards, such actions often require a complete system re-verification, re-validation and re-certification. Obviously, introducing faults in the development is inevitable, and in addition to their rigorous handling in the development, safety engineering needs complementary methods to manage those faults in the post-development phases systematically.

Thesis Contributions

This thesis presents an approach to perform changes on safety-critical systems in their operation and maintenance phase, with the objective to repair those systems from certain faults in a cost-effective way. To meet safety regulations while performing changes, a trade-of between the flexibility in terms of supported change types and the ability of methods to analyse the impact of such changes on system integrity has been made. Thus, the scope of the supported change types is reduced to replacements of software units (i.e., software components), thereby enabling engineers to repair a system from faults on a level of individual software components.

To support performing changes on an architectural level, a Component-based Software Engineering (CBSE) has been utilized [Crn02]. This paradigm is currently a key tool used for building safety-critical systems in many fields. According to principles of CBSE, software architecture is fragmented into (re-usable) parts, i.e., software components, that implement portions of system functions. The foundation for the correct system development in CBSE is a component model, which specifies what components are in terms of interface syntax, semantics and rules on how they can form a composite system, among a number of other aspects. The component model presented in this thesis provides basic definitions on software components and syntax of their interfaces, in order to allow performing supported changes at runtime.

Another, and more challenging issue addressed in this thesis is to ensure to which extent the incorporated changes may impact the system integrity, and whether such changes may compromise the system. In other words, it must be possible to verify whether new components, which have to be put into the hierarchy of the system design, match with the remaining parts of that system. This matching, or the compositional analysis, has to consider many dimensions, in particular, the behaviour of components and system, their interfaces with respect to syntax, semantics, and other aspects. To this end, a Contract-based Design (CBD) paradigm has been applied [SVDP12]. With CBD, the introduced component model is extended to capture various non-functional (or extra-functional) properties on a level of software components and their execution platform. These properties are derived from the top-level system requirements and represent just a portion of those requirements on a component level. Contracts are used here to structure properties, and to allow to link properties within the complete hierarchy of the system

design. In this way, the integrity information is maintained and changes in any contract can be tracked throughout the design hierarchy, and their impact on requirements can be analysed. In summary, the contributions of this thesis are the following (see Figure 1):

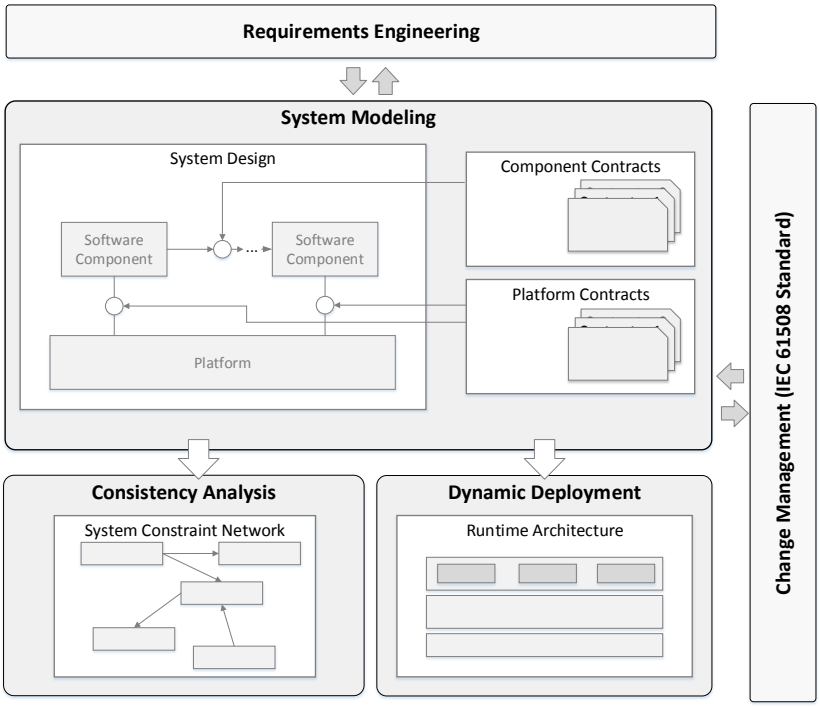


Figure 1: Overview of the change management proposed in this thesis: (a) system modeling using contracts, (b) analysis of changes introduced into the system design, and (c) dynamic deployment of the analysed system (or software component)

1. *System modeling and analysis* [1] [2] [3]
   In the context of modelling a formalisation is provided on how to build the system design in terms of contracts. In particular, the modelling here includes specific types of contracts required to capture the aforementioned non-functional properties, and it also includes basic relations that contracts must support to build the system hierarchy (i.e., relations between individual components and relations to their platform). Further, to conduct the analysis, a novel method, based on Constraint Programming paradigm (CP) [Apt03], has been proposed.

2. *Runtime support to perform changes (dynamic deployment)* [4] [5] [6]
   Software components that satisfy contracts within a composite system are the subject

[1] Constraint-Based Verification of Compositions in Safety-Critical Component-Based Systems, *SCI 2015*
[2] On Design-time Modelling and Verification of Safety-critical Component-based Systems, *IJNDC 2014*
[3] Towards Pattern-based Reuse in Safety-critical Systems, *EuroPLOP 2014*
[4] A Component-Based Dynamic Link Support for Safety-Critical Embedded Systems, *ECBS, 2013*
[5] Inversion of Control Container for Safety-critical Embedded Systems, *EuroPLOP 2013*
[6] Towards Predictable Dynamic Linking for Safety-critical Component-based Systems, *WiP@SEAA 2013*

to dynamic deployment. To enable this deployment, a runtime support has been provided. This mechanism allows to link components in their binary form into the real-time operating system (RTOS) at load-time or at runtime. The distinct feature here is that the proposed mechanism meets software safety regulations, and therefore can be used in the context of safety-certified RTOSs.

3. *Alignment with safety standards* [7] [8]
   To allow performing changes on safety-critical systems, it is necessary to align their management with the regulations of safety standards. For this purpose, the generic industry standard IEC 61508 has been analysed. The results of this analysis are certain design limitations that have to be set in order to prevent performing changes that cannot be supported. In addition, for such a limited design, a list of common properties that have to be considered in the modelling has been proposed.

The introduced approach to manage changes brings benefits in terms of effort and costs required to repair systems from faults in the operation and maintenance. Following the regulations of safety standards, the approach can be aligned with the conventional change management process, for example the one proposed in the IEC 61508 standard, thereby allowing the engineers to perform certain repairs without having to communicate the required procedures and changes with the safety assessors. Managing certain repairs as introduced in this thesis can be especially beneficial in fields that have rather frequent and cost-intensive repairs, and that, in particular, have an emphasis on mass production, as it is the case in the automotive or biomedical engineering for example.

[7] Towards Assured Dynamic Configuration of Safety-Critical Embedded Systems, *SAFECOMPW 2014*
[8] Reducing Certification Costs Through Assured Dynamic Software Configuration, *ISSREW 2014*

Contents

List of Figures

List of Tables

List of Abbreviations

OMG	Object Management Group
UML	Unified Modeling Language
CBD	Contract-based Design
PBD	Platform-based Design
CBSE	Component-based Software Engineering
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
SAT	Boolean Satisfiability Problem
SMT	Satisfiability Modulo Theories
IMA	Integrated Modular Avionics
AUTOSAR	AUTomotive Open System ARchitecture
DbC	Design by Contract
MDE	Model-driven Engineering
NFP	Non-functional Property
RTOS	Real-time Operating System
OS	Operating System
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen
ARM	Advanced RISC Machines
RISC	Reduced Instruction Set Computer
SIL	Safety Integrity Level
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
HIPASE	High Integrity Protection Automation Synchronization and Excitation

Chapter 1

Introduction

The use of computers to control various processes in physical environments, machines in the industrial control or even processes of other computers has been practised since the last five decades. In response to the successive progress in hardware technologies at that time, in particular, the introduction of integrated circuits in the late 1950s, the engineers were able to produce hardware devices in increasingly smaller configurations, which in turn enabled their direct embedding in the environment they had to control [BS93]. In parallel, more powerful and more reliable hardware was a motivation to realize flexible and generic products, by allowing to execute some control functions in software. Although the incorporation of software for the processes control was not widely accepted by the control systems community, especially in the application fields such as nuclear power plants and some sectors of the industrial automation, where the computer-based controllers were able to damage the equipment (i.e., mechanical parts, installation, etc.), or even to pose harm to humans in cases of malfunctioning, the real expansion of the software started in the early 1990s. This trend introduced many opportunities, but also many challenges for systems engineering. On the one hand, the companies were able to speed-up their development and production cycles, to give quick response to changes and to better maintain systems; whereas, on the other hand, they were confronted with the complexity issues from the beginning. The correctness of the hardware with respect its function and quality is due to its simple and standard logic more easily to analyse and to verify than software, e.g., using formal methods, based on mathematical models of the behaviour, reliability, and other system attributes. In contrast, software is not only error-prone because of the low-level languages for the functional specification such as the machine code or assembler, or even because of higher-level languages that were introduced later, but rather because of the size and heterogeneity of the state space that it could incorporate (i.e., the representation of numbers, and their use in state variables).

Today, software is a major innovation driver in many application fields, but, at the same time, is represents the main source of problems, basically the old problems related to the complexity management – and as pointed out by Leveson [Lev86]:

> *"... most of problems are not new, but are only of a greater magnitude."*

This trend of dominating software kept progressing in the last decades, and impacted systems engineering in many aspects. In the following, this impact on systems engineering is discussed more in detail and, thereafter, the problem statement for the thesis is defined.

1.1 Motivation

1.1.1 Embedded Systems: Computational Models and Applications

To date, different definitions of a term *embedded system* have been provided in the literature. According to a very generic one given by Barr and Massa [BM06]:

> *"An embedded system is a combination of computer hardware and software – and perhaps additional parts, either mechanical or electronic – desired to perform a dedicated functions."*

A *dedicated function* in this context may have different meanings, mainly deepening on the nature of the considered application field. One such a function is for example controlling a process of a physical world – e.g., the temperature of a chemical reactor in the context of the process control engineering; the engine control in the automotive or avionics sector, and so forth. These are the typical systems implementing the basic models of the control systems theory (see Figure 1.1). Generally, the *physical process* in a model depicted in figure corresponds to a set of operations that are intended to influence the physical world [OP92]. This influence is made on basic process parameters: materials, energy, and information. Materials and energy are the main values that characterize some physical process, such as heat and cooling in the aforementioned chemical reactor example; whereby the information provides a means on how the embedded system influences the state of the process, in terms of the material and energy, and vice versa. The embedded system in this case provides a function of the controller using the information channels.

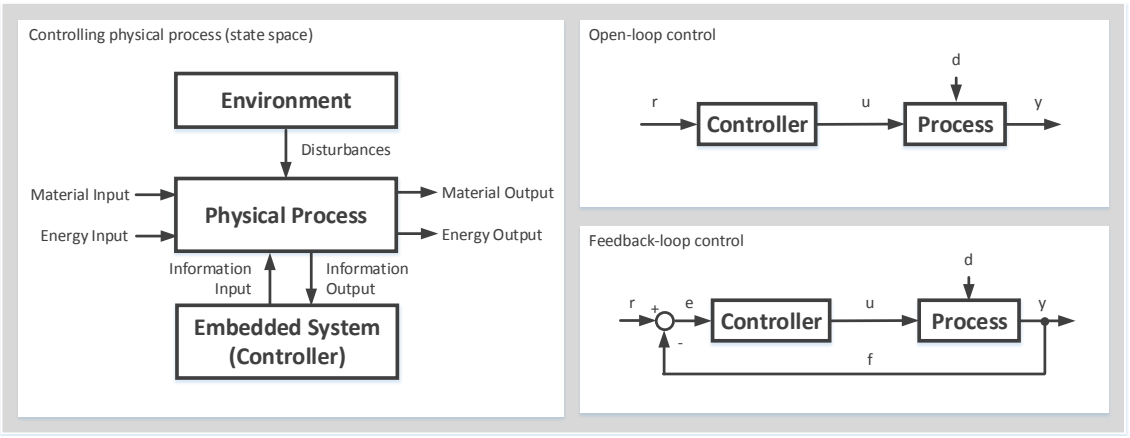


Figure 1.1: Controlling a physical process using an embedded system: the implementation (left) – adopted from [OP92], and basic models of the process control theory (right)

Another, and more general class where embedded systems apply are reactive systems. In contrast to the previous system model with the narrowed control loop behaviour, these systems perform their function based on occurrence of specific events (cf. untimed computation model [Kaz09]). Distributed, event-based architectures, such as AUTOSAR for automotive devices [KF09], or IEC 61850 for devices in electrical substation automation systems within smart grids [YVNC11], are examples here[1]. Events correspond to actions,

[1]Both architectures implement timed and untimed computation models [Kaz09].

i.e., input actions from the environment or actions performed by users like switching the car lights in AUTOSAR architecture, or even internal actions of systems which implement the behaviour shown in Figure 1.1, with the aim to trigger the reaction.

In general, there is a number of other, different dimensions to characterise models of embedded systems. Their concrete form particularly depends on the nature of the *desired function* and the strategy on how such a function has been realized, i.e. the characteristics of an embedded system. Jantsch [Jan03] summarizes these dimensions and provides a taxonomy of possible system models (see Figure 1.2).

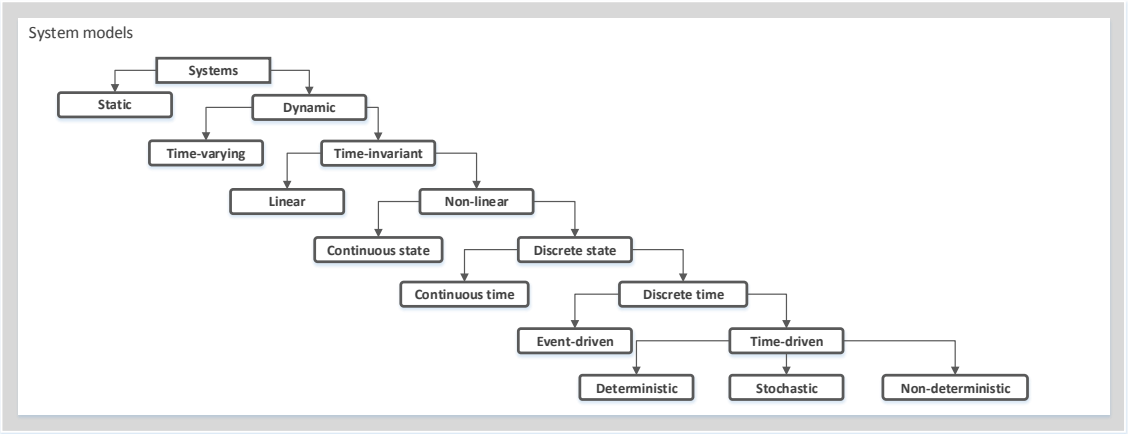


Figure 1.2: Taxonomy of system models, [Jan03]

These system models roughly describe the nature of a function or a physical process to control and an embedded system, but they do not provide details on how the embedded system has to realise such a function. Generally, when designing embedded systems, their concrete implementation, in terms of hardware or hardware/software synthesized functions that are coming from requirements, has to fit to a particular system model. To success in this intent, a number of details about a system have to be considered. They include functions to implement, the way on how those functions are executed, models on their communication, refinement from requirements to implementation, and resource models and constraints such as timing and memory, among many other aspects [Mar10]. This is a part of embedded systems engineering, which becomes increasingly important as the complexity of embedded systems turned to be an issue. All the outlined different engineering aspects require the distinct engineering disciplines, in order to correctly map a system implementation into the desired system model. For example, when realizing an interaction between individual control functions, a model of processes, such as Kahn Process Networks or Petri Nets may help to first define the communication at some abstraction level, on which the essential system properties can be analysed, before any concrete implementation has been provided and mapped onto the system model. Similarly, the system behaviour can be abstracted using State Transition Systems (STS) [BK08].

Generally, various aspects are required to consider when specifying and refining a system, in order to correctly conduct its synthesis. Jantsch [Jan03] summarizes relevant aspects of the embedded system engineering into the following main groups:

- *Computational model.* It describes an observable behaviour of the system, or that of

the system parts (hardware, software components), i.e., a relation between system inputs and outputs. The representative notations are various techniques that support specifying behaviours for both hardware and software in different aspects and at different abstraction levels – i..e, from the low-level hardware transistor and gate level functions, software instructions and algorithms, to the system-level relations and functions. Those representations are based on theoretical models of computation, including sequential (e.g., Finite State Machines (FSM)), concurrent, parallel, and timed models, among many others [Fer09].

- *Data model.* This model provides a notation for data required by the system to perform the *desired function*, i.e., to act with the physical system using information channel (see Figure 1.1). A number of different formats are used to represent the desired data, including floating-point numbers for approximating continuous values, integer or boolean numbers for logical values, and so forth.

- *Time model.* The notion of time is a very important characteristics of embedded systems. Time not only provides a reference value for the reaction on certain events, but also the causality of such events, or the order of execution of individual functions within a system. Again, there are many notations of time, based on considered abstraction type. For hardware functions on gate level, usually the clock is the trigger of transition within a circuit. On the other side, for the software system on the application level, the execution time is often a reference value to analyse the real-time constraints. Except of that, many state-full systems use time as part of their data model to correctly compute the outputs (e.g., controllers, filters).

- *Communication model.* Since the system, per definition [JAY84], comprises a set of smaller, individual components, each of them implementing a distinct function, the interaction among those components has to follow the specific communication model, according to the concurrency and parallelism of their interactions. The communication model forms, in fact, the top-level system behaviour.

One of the main functions in the engineering embedded systems is first to specify a system model, in its all necessary abstractions and aspects, based on given requirements, and to synthesize it correctly using various notations. As pointed out by Jantsch, the system is often a mixture of different computation and communication models, rather than a single one, which even more complicates the tasks in the engineering. Due to difference of application fields, different objectives in systems modeling, but also different communities, the challenge in the engineering today is to provide sound technologies to consider all necessary models in a holistic way.

1.1.2 Challenges in Engineering Safety-critical Embedded Systems

Embedded systems, in their form introduced previously, are also used to automate and to control processes in critical environments. Such environments include various control tasks in the industrial automation sector, nuclear power plants, electrical substation automation systems and some processes in smart grids, transportation, and so forth. The distinct characteristic of these so called safety-critical systems is that system failures, that might arise from any functional part, i.e., hardware, software, or mechanics, may influence the

controlled process in an unplanned way and thus may cause severe consequences – damage the equipment, introduce the environmental damages, or even pose threat to humans that eventually operate with those systems [SS10]. Since the last few decades, the society had to undergo a number of disasters, caused by malfunctioning of embedded systems [Air14, Qi 14, AIKR13, Lev86].

To build such systems more safe and reliable, the community has established a number of methods, standards, and disciplines, with the aim to characterize the system in terms of its quality, in particular, the probability of failures occurring in particular context, which is a value characterized by the *safety* property, and to control that property systematically. According to the International Electrotechnical Commission [Bel06], safety is:

> *"... the freedom of unacceptable risk."*

The risk in this context corresponds to the probability that an accident happens (i.e., harm). In general, the goal of the engineering to this end is to achieve the desired system quality by reducing that risk. From the technical viewpoint, and as mentioned previously, this achievement is possible only by specifying the system model correctly and by conducting its complete synthesis correctly. According to Henzinger [Hen08], such a correctness can be achieved if this class of systems can feature the two main characteristics, the *predictability* and *robustness*. The predictability relates to the determinism of the system model, i.e., all outcomes of the communication and computation models must be analysable beforehand. The robustness, on the other hand, is the ability of the system to react on uncertainty of the environment. In fact, this characteristic is not part of the functional aspect of the system, but it must be provided by the system. Examples here are various functions that implement fault tolerance mechanisms [Dub13].

Even though the predictability and robustness have been considered thoroughly in systems development, the practice has shown that the engineering often fails to achieve its main goal, and failures become an inevitable part of systems lifecycle. There are many reasons for this, but, one has to be highlighted – the complexity, which is today of few magnitudes larger than in the last decades, and it currently represents a major challenge for systems engineering in many application fields. In the following, some relevant field reports on complexity issues are outlined, and thereafter, related consequences are discussed.

Managing Complexity

From the discussion above it becomes apparent that the engineering of safety-critical systems has to be rigorous and quality-centred. On the other hand, this engineering was constantly influenced by the market demands, as in other domains, to improve the productivity, to shorten costs in development cycles, and, at the same time, to develop high quality products. One of the consequences here is that more and more functions are being shifted to software, and the complexity management is mainly focused there.

Today, software functions are inevitable part of safety-critical systems. According to Ebert and Jones [EJ09], the average increase rate of software complexity, in terms of its size (lines of code, instruction, interface complexity, etc.), lies between 10% and 30% per year (concrete value depends on specific application field). In some fields that in particular base their production on mass customization, this growth rate is up to 50% (e.g., avionic embedded systems).

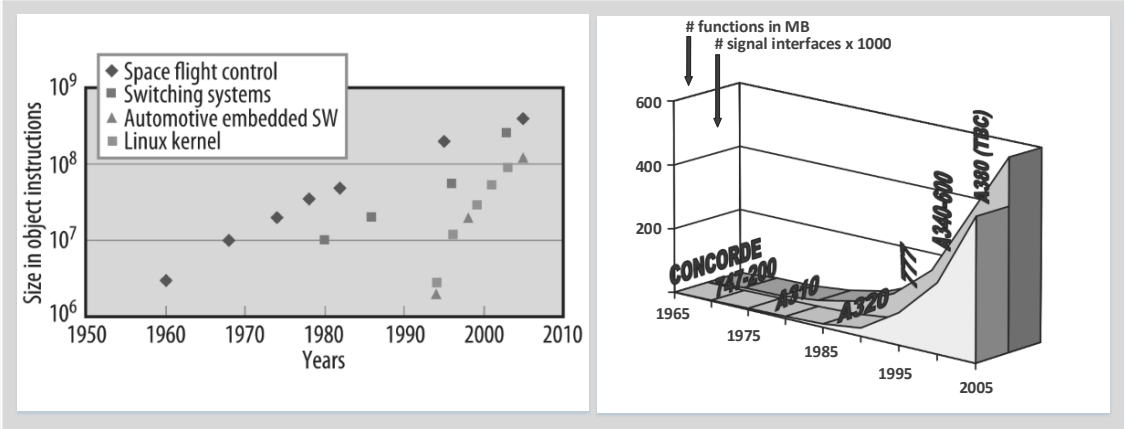


Figure 1.3: Complexity development: embedded software in automotive, switching systems, and space flight control [EJ09] (left), and software and interfaces in avionic systems [But08] (right)

Figure 1.3 illustrates the development of software complexity in some application fields. As depicted on the left, the size of embedded software in automotive systems starts to grow rapidly in the late 1990s, parallel to the complexity of the Linux kernel[2]. The right distribution shows the exponential growth of software-implemented functions in some avionic systems.

Consequences

To achieve the desired quality, safety-critical systems need to undergo a number of rigorous processes in the engineering: development, verification, validation and certification (assessment or system qualification by the independent authority) [SS10]. With the increasing complexity in software, it becomes difficult to verify and to validate those systems. On the other side, development in many application fields is massively based on reuse of existing parts, as it can be observed from Figure 1.3, on the left (automotive software and Linux kernel). In some fields, this reuse goes beyond the organizational boundaries – for example, automotive companies mainly reuse (software) parts from their suppliers, in order to realise the top-level system functions. Unfortunately, the engineering in general lacks strong theoretical background and practical methods on how to construct or to compose the system out of parts, and therefore this construction is in the practice done more or less in an add-hoc way.

In response to these issues, more and more failures are occurring in the operation of safety-critical embedded systems[3]. The engineering mainly fails to accurately synthesise the system, including all its models of computation, communication, time and data. In fact, the real problem lies in the lack of the engineering support to perform such a synthesis while respecting non-functional requirements, such as resource constraints, interface syntax and semantic between system parts, power consumption, the quality and configuration of

[2]Note that this distribution comprises a complete software, i.e., safety-relevant one and software that has no influence on system safety)

[3]There is a correlation between the complexity (in function point metric, fp_N) and the number of expected failures $fail_N$. For embedded software, it is given by: $fail_N = fp_N^{1,22}$ [EJ09].

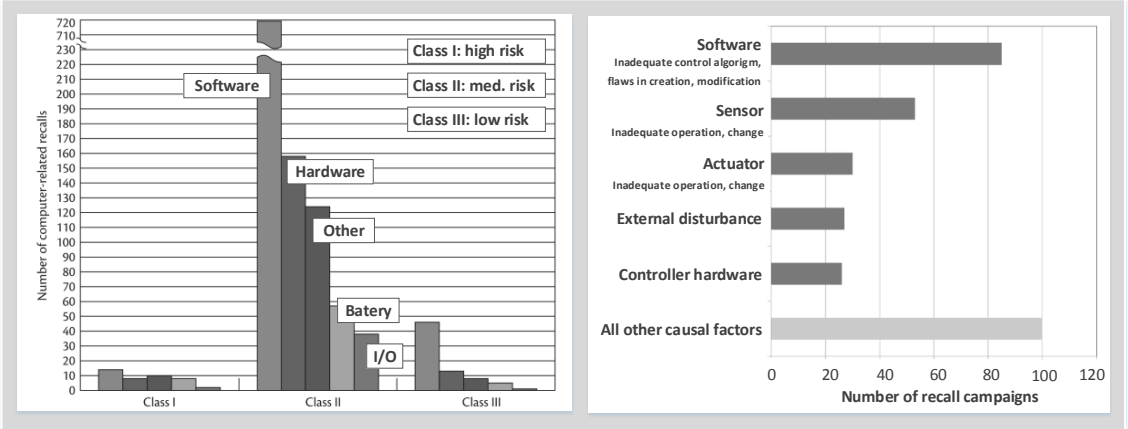


Figure 1.4: Distribution of recalls campaigns according to causing components: medical engineering [AIKR13] (left), automotive engineering [Qi 14] (right)

used hardware platform, etc. (cf., reactive and execution requirements mentioned by Henzinger [Hen08]).

Figure 1.4 illustrates the distribution of failures in two considered application fields, biomedical engineering and automotive, based on their cause. According to both studies, software represents a major cause for product recalls. In biomedical engineering (left part in figure), more than 60% devices had to be returned due to various faults in software [AIKR13]. The considered period for collecting data was (2006-2011) and (2002-2013) for biomedical engineering and automotive respectively.

1.2 Problem Statement

The consequences on the complexity trend summarized above pose challenges not only to development, but also to many activities in the operation of safety-critical embedded systems. One of such activities are repairs that have to performed to remove faults from system components. Repairing this class of systems from faults is usually a cumbersome and cost-intensive task, because, they are often certified by the independent authorities, to prove that they have achieved the claimed quality level and that they have followed certain guidelines of safety standards[4], and incorporating any kind of changes to perform repairs requires a lot of re-engineering effort, in particular, to communicate with the associated authorities to make plans for the change management, but also to follow related guidelines in standards, which very often require to re-verify and re-validate the complete system.

Generally, the engineering of safety-critical embedded systems lacks necessary methods and procedures to systematically address repairs or changes required for repairs. In response, even minor changes such as configuring and calibrating data models, or maintaining algorithms, may be difficult to perform. On the other side, the motivation to have methods to address such changes in a more cost-effective way is that software complexity keeps growing, and software-caused failures, which correlate with this complexity trend,

[4]Safety standards provide guidelines to conduct systems and safety engineering together, in order to systematically achieve the desired quality level, i.e., the safety integrity.

are becoming inevitable part of systems lifecycle.

The problem statement: One way to reduce the engineering costs in the overall lifecycle of safety-critical systems is to effectively handle their repairs. The challenging part that has to be addressed by the supporting method is to analyse to which extent the system can be changed, i.e., to identify the possible supported change scenarios that can be performed in the operation and maintenance, while at the same time, it has to be ensured that the necessary regulations of safety standards with respect to change management are followed. In fact, a trade-of between flexibility and rules posed by standards has to be found.

1.3 Thesis Contributions

The approach presented in this thesis allows to perform changes on software in the operation and maintenance phase of safety-critical systems, with the objective to repair those systems from certain faults in a cost-effective way. To meet safety regulations while performing changes, the following two objectives have been achieved: (i) a trade-off has been found between the flexibility in terms of supported change types and the ability to analyse whether the impact of such changes can compromise system integrity, i.e., whether system requirements are still satisfied, and (ii) necessary procedures to perform changes are aligned with requirements, measures and techniques of safety standards. In summary, the approach provides the following contributions:

1. *System modeling and analysis*
   In the context of modelling a formalisation is provided on how to build a model of the system that captures information about non-functional system requirements on a level of individual design elements (system components). To allow performing changes, like exchange of instances of control algorithms and libraries for example, the system architecture is defined following the Component-based Software Engineering (CBSE) [Szy02, Crn02]. Using this paradigm, well-defined interfaces between the application that implements various control functions and the remaining software layers, i.e., the container, the operating system and hardware abstraction layer, could be established. For both parts of software, i.e., software components that form an application, and the platform (lower layers), the modeling support provides a means to capture portions of the non-functional requirements, so that impact of changes in any of those parts on system requirements can be estimated. This helps to take the decision whether changes can be performed without the extensive re-verification and re-validation effort, and without involving associated certification authority (assessor) or not. As a background technology, the Contract-based Design [SVDP12, BCN+12] has been applied.

   Finally, to conduct the analysis, a novel method, based on Constraint Programming paradigm (CP) [Apt03], has been proposed.

2. *Runtime support to perform changes*
   Software components that satisfy related requirements within a composite system[5]

[5] A system that is formed by wiring components, using related composition operators [GS05]

are the subject to dynamic deployment. To enable this deployment, a runtime support has been provided. This mechanism allows to link components in their binary form into the real-time operating system (RTOS) at load-time or at runtime. The distinct feature here is that the proposed mechanism meets software safety regulations, and therefore can be used in the context of safety-certified RTOSs.

3. *Alignment with safety standards*
   To allow performing changes on safety-critical systems, it is necessary to align their management with the regulations of safety standards. For this purpose, the generic industry standard IEC 61508 has been analysed. The results of this analysis are certain design limitations that have to be set in order to prevent performing changes that cannot be supported. In addition, for such a limited design, a list of common properties that have to be considered in the modelling has been proposed.

1.4 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 summarizes relevant related studies. In particular, it first introduces some background with respect to safety engineering and various software and system paradigms applied in that engineering discipline. Then, the chapter further continues with a literature review in the three main areas: configuration and change management in safety engineering, CBSE in safety engineering, and modeling and analysis methods for safety-critical systems. Finally, it ends up with a summary and the difference of the work proposed in this thesis to selected related studies.

Chapter 3 introduces the proposed approach. To this end, the chapter starts with the analysis of the considered safety standard with respect to requirements engineering and configuration and change management. The result of this analysis is the design of the change management support, with respect to procedures and supported types of changes. In the end, the chapter describes the remaining contributions: system modeling and analysis, and the runtime support to perform changes.

Chapter 4 summarizes the evaluation of the proposed approach. To this end, the chapter first introduces a case study, which is taken from the analysis of recall distributions that have been conducted on a typical embedded system in two considered application fields, and, later, the chapter introduces a metric for the evaluation, and ends up with the evaluation results, that show possible reduction in costs (effort) when applying the approach.

Chapter 5 concludes this work and shows the possible ongoing directions.

Chapter 6 shows a list of publications made during the work in this thesis.

Chapter 2

Background and Related Work

This chapter summarizes related studies relevant to target the problem statement given in this thesis. In particular, it first introduces safety engineering, basic principles and some applied paradigms. In the remainder, it outlines related studies targeting two distinct topics: methods for change management applied in domains of safety-critical systems, and modeling and analysis support for safety-critical systems. The former summarizes some recent works that target change management for this class of systems, and procedures on how safety standards are dealing with changes. The later shows a comparative view of formal modeling and analysis techniques, which provide essential supporting foundation for analysing the impact of changes.

2.1 Safety Engineering and Related Paradigms

2.1.1 Overview and Safety Lifecycle

Many physical processes controlled by embedded systems are critical in terms of consequences they may pose if such a controlling fails. Examples are chemical processes in nuclear power plants, various technical processes in automotive, and processes controlled by medical devices. Failures in embedded systems, caused by faults in their components, e.g., hardware or software, may in some cases pose threats to humans operating with those systems and processes, or/and cause some costs due to potential damages in the equipment or environment in which the processes are being controlled. Systems used for such a control have to be therefore developed according to rigorous and quality-centred engineering. Safety engineering, to this end, is a discipline which usually in conjunction with system engineering aims at conducting this rigorousness. In other words, the goal of the integrated safety and system engineering is to drive the system development while taking into account safety[1] thoroughly in systems lifecycle, i.e from conception and planing to realisation, operation and disposal [SS10].

In parallel to phases of system's lifecycle, there are distinct activities that are provided for achieving goals of the functional safety[2]. These lifecycle activities are generally provided in safety standards, which act as a common knowledge base for engineers that guides them on developing this class of systems, i.e., safety-critical systems.

[1] Safety definition: "freedom from unacceptable risk", [SS10].

[2] Part of system safety that only depends on correct functioning of the control system, [SS10].

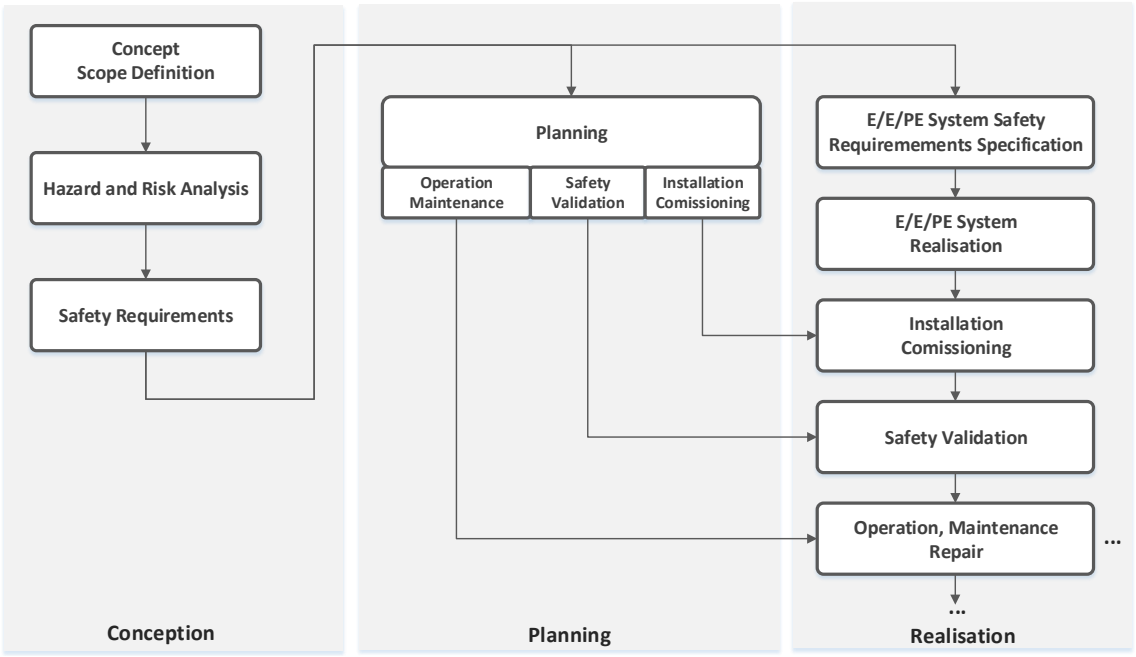


Figure 2.1: Safety lifecycle, according to IEC 61508 standard (excerpt, [IEC10a])

One of such standards is the IEC 61508, developed for various industry sectors, and used as a base for developing many domain-specific standards (a brief overview is given in Section 2.2.3). Figure 2.1 illustrates the relevant parts of safety lifecycle according to IEC 61508. Basically, the safety lifecycle from the conceptional view, i.e., conception, planning and realisation, is quite similar in derived, domain-specific standards.

The lifecycle starts with the concept phase, where the scope of the system is defined, in relation to analysing the context (or situation) in which the system shall operate. This context is very important, since it gives a feedback on *criticality* and consequently, on the source of damages or harms (i.e., hazards) that may be caused by the system during the operation. Next, possible risks are identified, in terms of severity[3] of individual harms and their probability of occurrence. They further serve as a source of information to define safety requirements, i.e., requirements that a system must implement to reduce risks to defined, tolerable level. This is actually the main objective of safety engineering, i.e., (i) to characterize the risks by considering system and related context (situation, environment), and (ii) to reduce risks by providing means in terms of safety requirements. It is important to note that tolerance criteria for risk reduction is to some extent a balanced trade-off between costs invested in development and severity of identified risks, i.e., it is not possible to reduce risk to absolute zero, but instead the tolerance thresholds are set for specific contexts and environments (in terms of probability of failures per time unit).

Safety requirements are further refined and allocated to design elements (in some domains, functional and technical architecture), In further development phases, planning is performed on different aspects, such as operation and repair, safety validation and

[3]Severity usually expresses a strength of damage some harm may produce (e.g., for humans: levels up to loss of life).

installation. The validation, which is one of the most cost-intensive activities in the engineering, is for higher criticality levels performed by an independent safety assessors who also provide certificate for products.

Methods and Measures

Standards in addition to lifecycle activities also guide engineers with methods and measures that they may apply when developing safety-critical systems. These methods, on the one hand, target the predictability as mentioned by Henzinger [Hen08] such as following modular design, static scheduling, and static memory management for example. On the other hand, they suggest protective measures that target runtime safety mechanisms, such as self-tests and hardware fault tolerance for example.

Usually, such methods and measures are recommendations that are allocated to specific criticality level (i.e., Safety Integrity Level, SIL) the system has to achieve.

Supporting Paradigms

Introducing software in controlling critical processes caused lot of problems in validating safety. Unlike hardware, whose safety can be quantified during the whole lifecycle, for software different measures have to be applied. Basically, those measures are oriented towards eliminating design, systematic faults introduced in development.

As for general purpose and embedded engineering, time to marked push is also targeting development of safety-critical systems. Software engineering for this class of systems becomes a grand challenge, because, and as given in the introducing section, it becomes difficult to manage its complexity. Currently, engineers are applying different software and system paradigms to cope with this challenge. In particular, Model-based Engineering (MDE) in conjunction with many other paradigms, such Component-based Software Engineering (CBSE), are seen as promising support for software engineering in safety domains.

In the following, relevant paradigms and their basic principles are introduced.

2.1.2 Component-based Software Engineering

General

Component-based Software Engineering (CBSE), or in some parts of literature, alternatively – Component-based Development, is a paradigm that facilitates modular development of software systems, by providing a means to systematically design system parts (software components, frameworks) for use and re-use, and necessary processes and rules to build systems out of such components. Like object-oriented programming paradigm, the objectives are put towards providing a sound engineering support to address managing complexity, improved productivity, and system quality. On the other side, in contrast to traditional paradigms, the focus in CBSE is given on systematic design of software artefacts for their specific operation and specific context.

During the last decade, a number of different domains have adopted CBSE, and were able to experience some benefits, but also some challenges. The challenges are mainly reflected in lack of basic principles and standards in CBSE to target specific characteristics

and needs of different domains [CSVC11]. An experience report provided by Panunzio and Vardanega [PV11] shows, for instance, that much details need to be considered to develop a component technology for specific domain, without being supported enough by CBSE principles and standards. To this end, the CBSE community is trying to provide those principles and standards. As one of the consequences, several definitions on what builds a component-based system and software components alone, have been proposed. One of the first widely accepted definitions targeting software components, provided by Szyperski [Szy98], states that:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party.*

Later, as pointed out by Crnkovic and colleagues [CSVC11], this definition was complemented by Heineman and Councill [HC01], to target not only design of software components and their role in the system, but also necessary activities required during the lifecycle, which are captured in a so called component model. Chaudron and Crnkovic [CC08] define the component model as:

> *... standards for 1) properties that individual components must satisfy and 2) methods for composing components.*

In other words, component models provide rules on how to build individual software components to target particular domain and context by restricting their behaviour using properties (i.e., fragments of system requirements), and how to build composite systems (i.e., compositions [GS05]) out of such components taking in account different phases in systems lifecycle [CSVC11]. In contrast to definition of Szyperski, domain experts are responsible for defining what their software components are, and what rules to apply to build their systems.

Recently, towards more detailed but still generic view of CBSE, Crnkovic and colleagues in [CSVC11] developed a set of dimensions that characterize essential commonalities shared among existing component models, *lifecycle*, *construction* and *non-functional properties*:

Lifecycle. Separated development process for software components and systems is a distinct feature of component-based systems [CLC05]. Lifecycle dimension characterizes different forms of software components during the development and operation, i.e., from the specification and modeling to the final deployment and execution. In each of these steps, different artefacts are used to represent software components (e.g., object files in the Implementation phase). One concrete variant of such a lifecycle is illustrated in Figure 2.2, in which software components undergo three steps in the lifecycle to be deployed on the embedded RTOS: modeling, mapping to the real-time model of RTOS, and deployment.

Construction. This dimension characterizes the specification of interfaces and interaction capabilities for software components. Interfaces may have further specializations, such as levels – syntax, semantics; type – operation-based, port-based; used language, etc. The interaction between software components is not necessarily in the

responsibility of components, but it strongly influences the way on how components access their interfaces. For example, in the pipe&filters architectural style, software components have standard behaviour and are accessing interfaces without being aware of the neighbouring components (exogenous style of binding [CSVC11]). In contrast, in CORBA Component Model [WSO01], client-side software components may explicitly contact the naming service to get the reference to the component providing a required function (endogenous style).

Non-functional Properties. Non-functional (or extra-functional) properties (NFPs) in context of CBSE correspond to characteristics of software components that in some way contribute to the top-level system requirements. In fact, they represent the quality stamp for software components [LAC12]. Examples of NFPs are the properties related to real-time, safety and resource requirements.

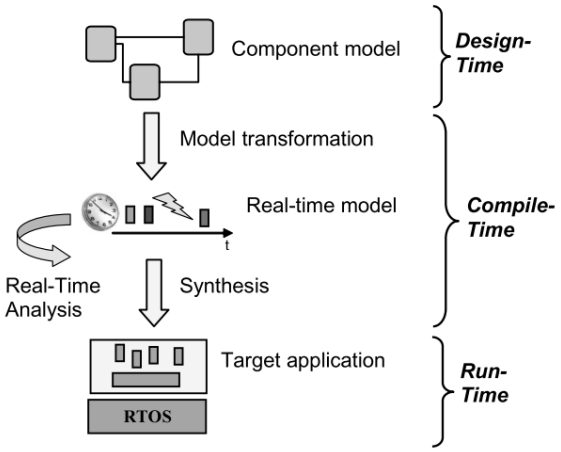


Figure 2.2: Synthesis process of component-based embedded system, according to AutoComp component model, [SFA04]

NFPs play a very important role in system construction, since they provide an important source of information to correctly build the system, in terms that considered system properties hold. A grand challenge here is currently to reason about system properties out of properties set on levels of software components. This topic and the state of the art are discussed in Section 2.2 mode in detail.

The outlined definitions and classification framework give some orientation when building component-based systems for specific domain and specific context. Obviously, concrete properties, construction, and the lifecycle have to be defined by the domain experts.

CBSE for Embedded Systems

Applying techniques for modular system construction in embedded system engineering has been practised since 1980s, first in form of Function Block Diagrams, for example in the Standard IEC 61131 (that was released later), then, and currently, in many application fields that rely on principles of CBSE. Figure 2.3 summarizes some relevant component

models, released in various sectors in industry, avionics, transportation, and electrical substation automation systems.

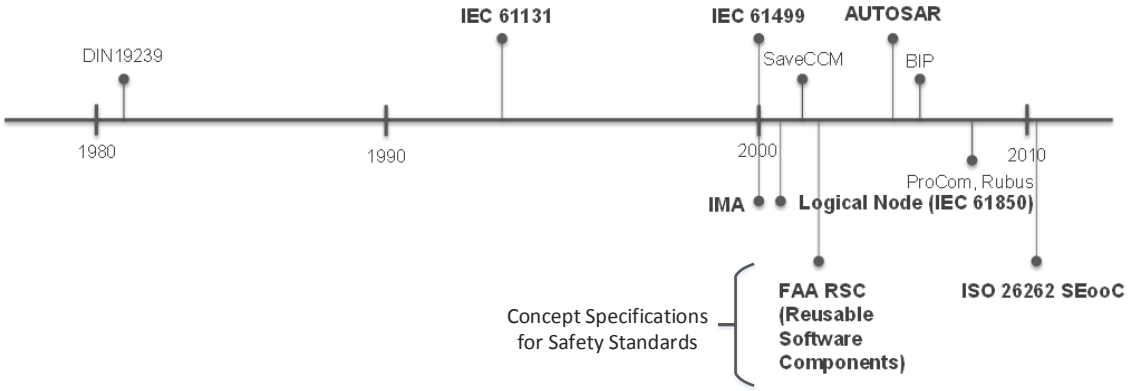


Figure 2.3: A time-line of some relevant component-based systems and concepts applied in the engineering of (safety-critical) embedded systems

The below illustrated RSC (Reusable Software Component, [FAA04]) and SEooC (Safety Element out of Context, [SBBK12]) concepts are the first guidelines targeting safety standards that consider CBSE in the system development, in particular, the design for reuse (RSC), and separation of the lifecycle in component development (i.e., SEooC Development) and system development (i.e., so called Item Development).

2.1.3 Model-driven Engineering

As mentioned in the introducing Chapter 1, managing complexity in system engineering has its roots in introducing software and its continuous growth. The common approach to effectively cope with complexity issues is maintaining abstractions of the system, in different aspects, and successive synthesizing such abstractions to the desired implementation model. As pointed out by Schmidt [Sch06], in the early 1980s, the engineering followed this strategy, but at that time it was not equipped with necessary language features and tools that can precisely capture abstractions. Instead, various existing imperative languages were used. However, such sparse abstraction models could not target the complexity issues, in particular, growing size of existing frameworks and platforms on the one hand, and mapping different abstractions, for example to support configuration and deployment, on the other hand.

Today, this intent of abstracting complex systems is supported by the distinct paradigm, Model-driven Engineering (MDE) [Sch06]. MDE aims at providing means to precisely describe desired abstractions and to synthesize them to desired implementation models (see Figure 2.4). According to Schmidt [Sch06], MDE achieves this by utilizing principles of Domain-specific Modeling (DSM, [KT08]), i.e., (i) domain-specific languages (DSLs) – that provide necessary formalization (abstract, concrete, formal, or semi-formal), and (ii) transformation engines and generators for the synthesis.

Currently, MDE has been applied to different domains, to capture different abstractions, and to serve different purposes. Persson [Per09] summarizes MDE domain classes into the following three groups: Information Modeling – organizing documentation, i.e.,

replaces the traditional way of informal documenting (e.g., class diagrams in OMG UML); Executional Modeling – various executable models (e.g., Matlab Simulink); and Formal Modeling – systems with formal semantics (e.g., automata, axiomatic models, etc. – see Section 2.2).

In the following sections thorough this thesis, the focus is oriented towards describing abstractions using formal models.

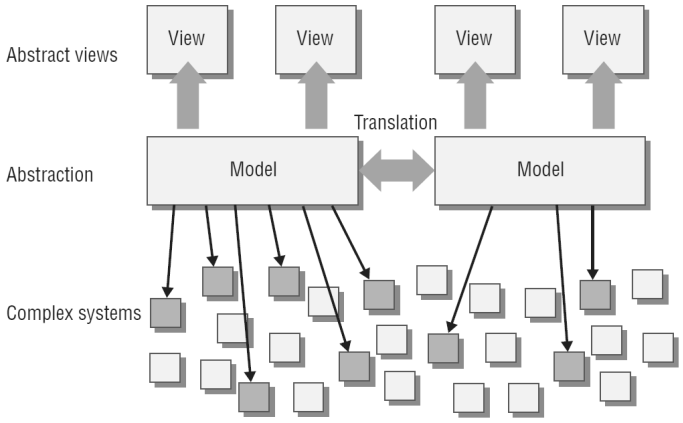


Figure 2.4: Managing complexity using MDE, [Sch06]

2.1.4 Platform-based Design

Using modular approaches to build systems, combining the top-down and bottom-up design flows, has been practised in the engineering of electronic systems for many years, in particular, for the synthesis of integrated circuits (ICs) [SNWSV09]. To cope with complexity issues associated with such an integration-oriented development, the community in this domain has established a new engineering paradigm, the Platform-based Design PBD [SVCDBS04].

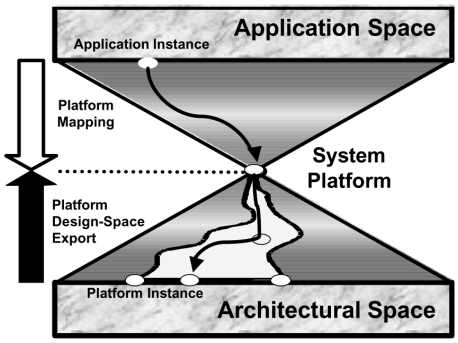


Figure 2.5: Concept of the Platform-based Design [SVCDBS04]

The PBD paradigm has been applied to balance the development costs and system performance by providing a stack of individual abstraction layers that can be successively refined. According to the PBD stack illustrated in Figure 2.5, there is on the one hand, an abstraction stack that captures the essential information about system functions or

application, and, on the other hand, the set of abstractions that describe the platform on which that application shall run. Basically, PBD abstractions allow the customizations of the design, by providing a means to rigorously validate every step in refinements.

As pointed out by Sangiovanni [SVM01], PBD is a promising approach to support rigorous development of embedded systems. The standard form of PBD illustrated Figure 2.5 can be applied to different types of systems (not only ICs), for example to perform allocation of the application software onto standardized AUTOSAR middleware, or on some standardized OS interfaces such as that of the OSEK RTOS for example. These basic principles of PBD have been used as a foundation to build component models for safety-critical systems. In particular, the Contract-based Design (CBD) paradigm, which is currently considered a promising tool for engineering safety-critical systems, inherits PBD principles with regard to abstractions and refinements. The following section introduces CBD more in detail.

2.1.5 Contract-based Design

Applying CBSE in embedded systems engineering was a challenging task since beginning of CBSE era, and is today also the case. Many existing component technologies recently applied in various sectors of industry[4] are still based on interface specifications that only consider the functional view of the system, and mostly ignore much details about the context, i.e., the platform and environment in which the system shall operate, or in other words, the non-functional system properties (in some parts in literature [VSC+09], extra-functional properties). To enable rigorous development and to prevent introducing design faults by only applying CBSE (e.g., due to add-hoc reuse and lack of techniques for compositional modeling and analysis), non-functional system properties have to be put as first class entities in component models for embedded systems.

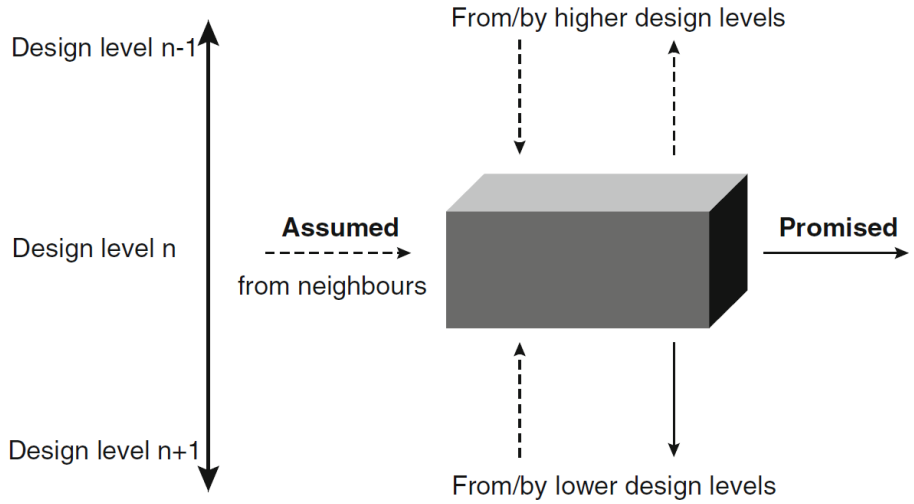


Figure 2.6: Concept of contracts according to CESAR project [PSS+13]

On the other side, from the academical point of view, a lot of effort has been invested in the last decade to provide sound technologies towards integrating those properties in

[4] AUTOSAR, IEC61850 Logical Node, IEC61499, Avionic IMA, just to name few.

component models (more details about concrete projects are given in Section 2.2). One of the success stories in this intent is the Contract-based Design (CBD) [SVDP12]. The CBD paradigm has its roots in classical Meyer's Design-by-Contract principle for object-oriented software [Mey92], which is in turn based on Hoare logic on formal reasoning of computer programs, in terms of setting pre-conditions and post-conditions on program executions [Hoa69]. During the last decade, this classical view of Design-by-Contract (DbC) has been adopted to CBSE by several authors, including Benveniste et al. [BCN+12], Sangiovanni-Vincentelli [SVDP12], and Damm et al. [DVM+05], and in the scope of several research projects (see Section 2.2).

The main difference to DbC is that in addition to capturing component behaviour, contracts for CBSE also provide means to integrate (software) components in the design hierarchy. That is, they also capture the context in which (software) components shall provide their services, in terms of provided and required services to/from other components, to/from neighbouring abstraction layers, and in terms of interfaces with respect to the platform on which those components shall function. This basic principle of contracts is depicted in Figure 2.6. In short, a contract describes, in terms of properties, what particular (software) component provides (promised services), while it also sets necessary requirements which must be satisfied by the context, i.e., assumptions, in order to provide promised behaviour. This notation of contracts is not the only one, for example Benveniste and colleagues use assumptions/guarantees, which is also the notation used in Henzinger's Interface Automata [dAH01].

The outcome of the recent studies associated with CBD is a so called meta-theory on contracts, which, according to Benveniste and colleagues, provides a common model for contracts targeting composite designs. This model sets formal semantic foundations on basic operators that have to be supported when developing such composite designs. In other words, to make strong integrity between system requirements and implementation, the following operators have to be provided:

Composition. Integration between (software) components in horizontal dimension (i.e., from neighbourhoods, to neighbourhoods in figure).

Abstraction and Refinement. Like in PBD, this operator allows to represent the system in different abstraction levels, and to build refinement relations between those levels (i.e., from/by higher/lower design levels in figure).

Platform Mapping. Mapping of the system design to the model of its execution platform. In the case of software, the execution platform may correspond to hardware devices in terms of resource models for example.

Viewpoint Fusion. Support to fuse different system views with respect to non-functional system properties (e.g., safety and real-time).

The contract meta-theory provided by Benveniste and colleagues is a general form CBD, and can be refined depending on the type of the system model considered. For example, in Henzinger's Interface Automata [dAH01], the properties on composition between two automata are describing the order of operations the automata have to respect during their interaction. In contrast, Sun et al. in their work about contract-based support for

PBD of analogue circuits [SNWSV09] are using algebraic properties on data models (i.e., variables used to characterize some analogue element).

2.2 Change Management Support for Safety-critical Embedded Systems

This section introduces some basic engineering concepts, methods and paradigms used to conduct the software change management that particularly address safety-critical systems.

2.2.1 General

Change management in the context of software engineering comprises a process and techniques to perform, to approve and to track change requests [BD00]. It is part of the configuration management, which basically provides engineering disciplines required to release software products (i.e., approved configurations, or *baselines*). According to Crnkovic et al. [CAD03], change management has the following two objectives: the first one is the process support that incorporates several steps and responsible roles, in order to conduct changes systematically; and second objective is the traceability support that is required to perform the change impact analysis.

In embedded system engineering, in particular, for dependable and safety-critical systems, it is much more difficult and error-prone to conduct such processes, and due to cost reasons as given in introducing section, changes are often omitted in many products. One of the first approaches to enable and to improve change management for this class of systems, in particular, to enable system upgrades, was started by the Software Engineering Institute (SEI) – Dependable System Upgrade Initiative, in the late 1990s [GW97]. The most important outcomes here were the challenges to be addressed by the systems engineering. The most relevant challenges are the following:

Design for Changes. or Design for Dependable Evolution [SEI01], includes the architectural support that can facilitate cost-effective change management, tools to analyse the impact on system components, methods to isolate the impact of changes[5], etc. In fact, specific types of supported changes have to defined in the early phases of systems lifecycle and incorporated into system architecture. The main characteristics of such an architectural support are: design simplicity, high cohesion, and low coupling [Rie01].

Required Processes. In original version the "Online Upgrade", includes procedures and methods that have to be developed to plan upgrades, to certify and re-certify upgrades, to develop criteria for accepting upgrades (also from the assessment point of view), etc. This includes the alignment of processes with regulations of considered safety standards.

In summary, assurance in performing changes is essential objective here. On the one hand, all necessary types of changes must be planned beforehand, which includes not only architectural support, but also processes and related responsibilities (e.g., parts that

[5] In SEI Dependable System Upgrade Initiative [GW97], upgrades are the only supported change types.

have to be communicated with assessor), and on the other hand, it must be ensured that changes are isolated and that they cannot compromise the initial or currently established system integrity. Also, the strategies on how to certify and to re-certify various types of changes have to be developed.

Characterising Changes

Changes may have an impact on different artefacts in the project landscape, with different strengths of that impact. Further, different operations may be required when changing the system, like adding/removing class members in the object-oriented software for example. During last decades, several approaches have been developed with the aim to generalize some aspects of change management and to provide a common framework to characterize changes.

According to Lehnert et al. [LFR12], there are few generic dimensions that characterise how changes influence the software system, in particular, its *representation*. The representation in this context corresponds to a system model that can be used to analyse the impact of changes, like graph-like representations for example [Leh11].

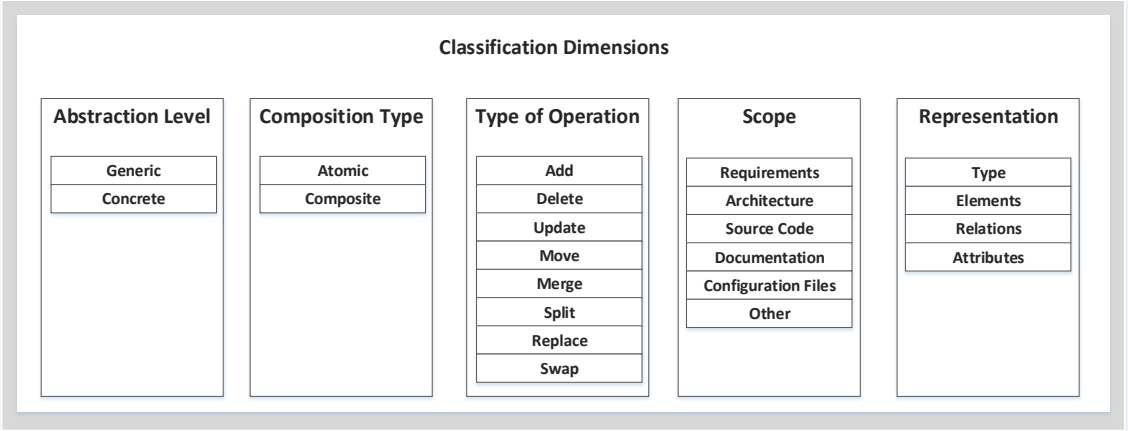


Figure 2.7: Classification framework for types of software changes [LFR12]

Any of the artefacts, shown in the scope dimension in figure, may therefore be part of the graph, represented in terms of elements, relations and attributes. Another important classification framework was proposed by Buckley et al. [BMZ+05] . In addition to only focusing on artefacts and representation, it also includes some dimensions relevant for the change management process:

Object of Change : Like the scope in the taxonomy from Figure 2.7, the object of change is any project artefact. In addition, object of change is characterized with the granularity of change, its impact, and propagation to other objects.

Temporal Properties : Time, frequency and history of changes.

Change Support : Process support – e.g., change management in IEC61508); degree of formality – a formal way to define the system and to analyse changes; type – operations required to perform changes; degree of automation – automated, partially-automated, and manual change support.

Technology/paradigm	Description
Conventional Programming Practice: Adaptors	Simplest form of incorporating changes, using *copy & adapt* approach (e.g., adapter design pattern for object-oriented (OO) software systems).
Scripting and Glue	Using scripts to build composite systems (e.g., pipes and filters architectural pattern).
Adaptation by Name	Changes supported by reflection (e.g., Microsoft .NET Common Language Runtime).
Linking and Interconnection Languages	Linkers, Loaders, Dependency injection, Component containers (e.g., .NET and Java runtimes).
Aspects, subjects and decentralised modularisation	Patching, Aspect-oriented Programming (AOP), Subject-oriented Programming, etc.
Program transformation	Runtime transformation of managed code (Java).
Software connectors	Connectors for component-based software, [Kel07].
Packaging	Packaging of component (object) executables.
Orchestration	Linking services (context: Service-oriented Architectures, SOA) using notations for business process modeling.
Coordination	Techniques that define communication models (e.g., synchronisation between components).
Specialised adaptation	Binary rewriting such as in Virtual Machine Monitors (VMMs).

Table 2.1: Technologies and paradigms that enable to realize various change scenarios in software [Kel08]

System Properties : affected non-functional requirements (e.g., safety, in the case when technical safety requirements are affected).

Supporting Techniques

Different technologies and paradigms are used to enable change management. Basically, they provide a support to perform changes on a level of code, models (architecture, requirements), and other artefacts such as documentations and configurations [Leh11]. Table 2.1 summarizes some relevant applied paradigms [Kel08]. For example, the representative mechanisms of the Linking and Interconnection Languages are the low-level operating system mechanisms such as linkers and loaders [Lev00], some middleware services or patterns – Dependency Injection [Fow04], or more advanced loaders in component frameworks.

Except of the concrete techniques, some of relevant engineering paradigms that may help in designing system for changes, in particular, the changes that have to be performed in the operation and maintenance (e.g., for so called dynamic architectures [Crn02]), are the following [MSKC04]:

Separation of concerns : Separation of the system's functional view and the cross-cutting aspects like safety, security, quality of service, etc. One of the supporting technique is the Aspect-oriented Programming (AOP), where different parts of the system, each representing particular system aspect, are combined (weaved) at some point of time. Separation of concerns simplifies the change management process by allowing to focus only on particular aspects of the system.

Computational reflection : This is the ability of the system to reason about its own

behaviour. It is more relevant for change requests that have to performed online (e.g., online maintenance), or for adaptive systems. Basically, the reflection provides knowledge about relevant traceability information inside the system, so that the system can decide whether to perform changes or not.

Component-based design : CBSE provides a sound basis for performing architectural changes. The interfaces between components enable low coupling on an architectural level, whereas on a component level, they provide information about functional and non-functional aspects of components (i.e., for component-based systems that have specified syntax and semantics, to some extent). The integrated description of various functional, non-functional properties on interfaces allows to identify how changes may propagate from software components to system requirements. CBSE is one of the promising paradigms that can provide some answers to the challenges set by SEI Initiative, in particular, the Design for Changes.

Runtime Support (Middleware) : For specific type of changes, in particular, the ones that have to be performed online, such as upgrades or adaptations, various services have to be provided by the runtime (middleware). Those services basically rely on low-level communication mechanisms such as function pointers, middleware interception (e.g., CORBA, .NET, Java RMI), aspect weaving, proxy pattern, virtual component pattern, and other.

Technologies Used in Embedded System Engineering

In embedded systems engineering, there are several approaches that provide a runtime support to performing changes, i.e., that facilitate the use of linkers and loaders, and specific component frameworks. One of such approaches is the THINK Component Model [PS08, POS06], that implements dynamic linker and provides an adequate representation of component binaries to be used by the proposed linker. The linker design is based on a typical Unix-like model, which basically loads binaries in ELF format (Executable and Linking Format) and performs finding necessary interfaces (functions) and connecting them (relocating) inside of component binaries. Thus, changes in the architecture are supported on a level of individual software components.

Another class of embedded systems where such a runtime support is provided are Wireless Sensor Networks (WSN) [HKS+05, DCL+09]. Here, due to more restricted memory and computation resources, simple linkers with so called indirection tables are provided [KPK13]. These tables are used to route the function calls between involved binaries. The entries inside of tables are memory addresses, for example contained in function pointers.

There is also a trend on implementing runtime services for change management in modern embedded systems. For component-based architecture AUTOSAR, the linking support is proposed that facilitates the usage of indirection tables, like in aforementioned WSN approaches [MFRV13, MVFR14, AK13, NKA14]. However, the linking support is still not part of the AUTOSAR specification (standard), but this topic is currently very present in ongoing AUTOSAR workshops.

More advanced linking support has been provided in the scope of the RECOMP project (Reduced Certification Costs for Trusted Multi-core Platforms) [PTV+13], which uses

annotations at the source level to setup the links between binaries and the remaining system. One of the project goals is to reduce the cost for certifying mixed-critical systems by splitting their functionality based on criticality and by allowing to change some parts dynamically.

2.2.2 Change Impact Analysis - Architecture and Requirements

Depending on the concrete type of changes and applied process, change impact analysis is performed on different levels: source code, models – i.e., architecture and requirements, and other artefacts such as documentations and configurations [Leh11]. In the following some recent and relevant studies that in particular handle change impact analysis on level of models and are focusing on the engineering of safety-critical systems are summarized.

One of the latest approaches that address the complete change management for automotive safety-critical systems was proposed by Oertel and Rettberg [OR13]. The approach uses formal representation of non-functional requirements, in particular, safety requirements to synthesize the system. On a level of individual components, those requirements are represented in form of so called Safety Contracts, which capture the failure model of individual components. Thus, every contract defined properties (assumption), that have to be satisfied by components in order to prevent firing specified faults. Faults are derived from the specification of safety requirements, that are results of the hazard and risk analysis. The essential part of the approach is the link between requirements and their implementations on a level of components. To this end, the authors propose the formal definition on how to verify the conformance between the implementation and the specification [OKB14]. Any change in the system model, in terms of change in properties, can be therefore tracked to the high level requirements and in this way necessary change requests can be derived. The analysis of the system is performed using tools for model checking.

Another approach to analyse changes on a level of requirements is proposed by Montano [Mon11]. Here, the approach targets the embedded avionic systems that implement IMA (Integrated Modular Avionic) architecture which standardizes hardware and software platforms (middleware) so that many software functions can be hosted by different devices. This especially brings benefits in availability, since in case of device failures, required functions can be migrated to functioning devices and can continue their operation. To enable this, Montano proposes a model of a system based on constraints. Constraints describe different system aspects, such as timing and memory budgets for example. Every constraint is connected to top level constraint that describes certain requirement – i.e., the non-functional system requirement. In case of failures in certain device, another device (so called LRU - Line Replaceable Module), is inspected with respect to available resources, including changes that have to incorporated due to migration. The impact on changes is automatically analysed by inspecting constraints. As a background technology, Constraint Satisfaction Programming is used [Apt03]. The advantage of using CSP for this purpose is that not only the conformance to constraints of certain device can be analysed, but also the possible configurations that fit demanded resource constraints of particular function can be identified.

In the scope of automotive engineering, the formal approach for analysing the impact on changes was proposed by Adler et al. [ASTPH11]. The approach utilizes CBSE to design the system in a way that individual implementations of software components can be

changed, however, only statically defined implementations. The motivation of performing changes at runtime is to increase the availability by incorporating the design diversity principle [Dub13]. To this end, every software component provides different implementations, each having specific quality level. In case of failures, for example in sensors – which is more often undergo failures that controllers or actuators, the system has to adapt all components so that the provided quality of sensors can be processed. To enable this, the authors define a global type system, which comprises static quality levels, which can be assigned to individual software components. In response to changes of quality in certain software component, the impact is performed to find a proper configuration of the whole system. The software components are then configured according to the given quality level.

Another method to improve the productivity and to reduce the certification costs in domain of nuclear power-plants was developed in the project PINCETTE (Validating Changes and Upgrades in Networked Software) [ZOF13, Cho10]. The idea behind PINCETTE project is to analyse the feasibility of formal methods, in particular, model checking approaches, to enable the validation of changes. Changes are here performed at runtime, like in many modern Dynamic Software Updating systems (DSU) [HN05], and allow to modify the behaviour of the system, which is represented in form of State Transition Systems. Although the approach is far away from real practical use, it shows the feasibility of available formal approaches to support such disruptive changes of systems behaviour.

Summary

In addition to summarized studies above, a number of approaches have been proposed to analyse the change impact on the scope of models in software engineering in general. However, most of the those approaches are focusing on syntactical view of the system, such as inspecting the model instance with respect to its meta-model, e.g., relations between components, ports, and interfaces [Leh11]. As pointed out by Henzinger [Hen08], very important prerequisite to correctly synthesize the system is to align its implementation to related execution and platform requirements (i.e., non-functional requirements). The introduced approach by Oertel and Rettberg [OR13], and Montano [Mon11] as well provide a sound technologies to analyse changes when considering such non-functional requirements. This analysis is on the other side considerably simplified by applying the basic principles of CBSE, which allow to fragment the essential requirements into properties on a level of individual software components and to track the impact of changes by considering changes on a level of software components.

2.2.3 Supporting Processes in Safety Standards

Some safety standards have defined guidelines on how to conduct the the configuration and change management for the system and software. These guidelines are basically set of processes, requirements, and techniques that have to be applied in order to be conform with the standard. In the following, the change management support in some common safety standards is briefly introduced.

IEC 61508 Configuration and change management support is provided in parts 2 and 3 of the IEC 61508 safety standard. Basically, the standard provides processes,

requirement on achieving functional safety, techniques and measures for system and for software, depending on considered safety integrity level the system has to reach.

Figure 2.8 illustrates the flow of the change management process, that is triggered due to failures in operation. The first step in the process is to determine the type of failures in order to build a change request. For systematic failures, caused by development faults introduced in software for example, the change request is built immediately – for random hardware failures, on the other side, the performance of the system has to be evaluated first, in order to get the delta to the initial system performance. The changes request is further processed to perform the impact analysis. After performing the change impact analysis, the current hazard and risk data is updated (i.e., the technical safety and other non-functional requirements). The complexity of changes and related costs depend on the impact on this data.

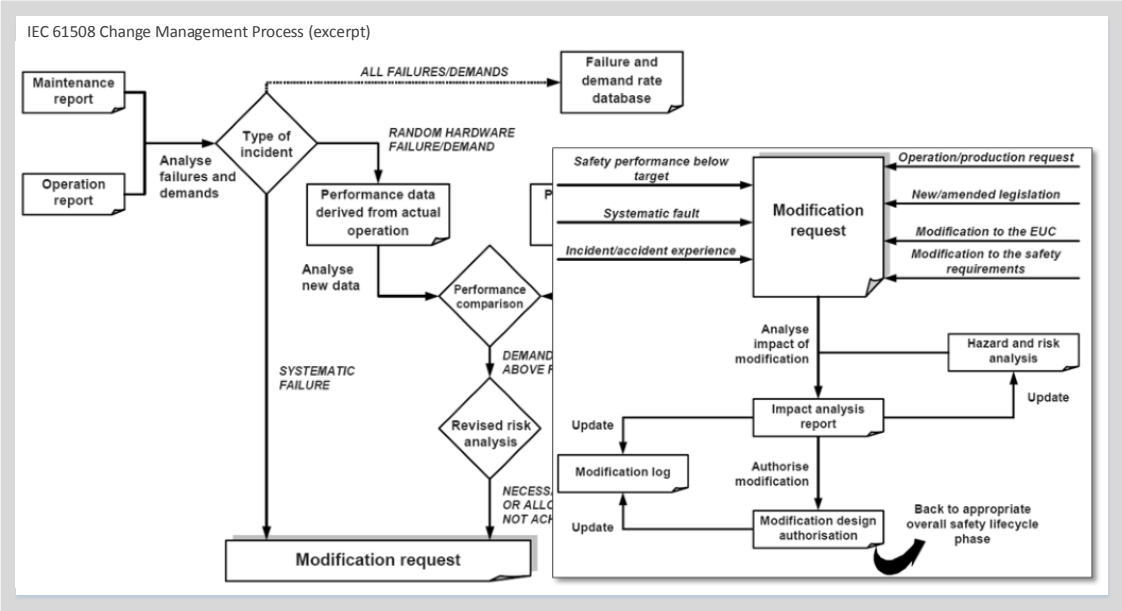


Figure 2.8: Change management in IEC 61508 safety standard [IEC10a] (acc. to IEC 61508, the term "modification" is used instead of "change")

The process illustrated in figure is accompanied with a number of requirements, techniques and measures to apply. For example, one of the requirements states that for systems with SIL 2, only the affected parts of the system have to be re-verified and re-validated, in order to achieve the desired safety integrity (more details about requirements, techniques and measures are given in Section 3.1).

Change management in IEC 61508 is related with processes for configuration management and maintenance.

IEC61508-derived Standards The domain specific standards, that are based on IEC61508, provide the same or to some extent refined concepts for change management and related processes. For example, the automotive standard ISO 26262 provides refined configuration and change management, with more details about input documents

Standard	Domain	Change Management Support
IEC 61511	process industry	modification procedures for software ; modification procedures for safety-instrumented system[6]; part 1, clause 16-17
ISO 26262	automotive	change management part 8, clause 7; configuration management part 8, clause 8; operation and service (maintenance and repair) part 7, clause 6 (all for hardware, software and system level)
IEC 60601	medical	modifications of PEMS (Programmable Electrical Medical Devices), part 1, clause 14.12
IEC 62061	machinery	modification procedures; configuration procedures clause 9 (safety-related electrical control system)
IEC 61513	nuclear power plants	maintenance part 1, clause 8; system design modification requirements part 1, clause 6.2.8; system maintenance plan part 1, clause 6.3.8

Table 2.2: IEC 61508-derived safety standards and their support for change management

and work products each of the processes has to generate. On the other side, some of the derived standards provide such a support only for certain level, i.e., system or software. The summary of these differences to IEC61508 are provided in Table 2.2.

DO-178B/C The DO-178B standard and its successor, the DO-178C, are guidelines for software safety in avionic systems. The standards are developed by the RTCA (Radio Technical Commission for Aeronautics), the US organisation that provides technical guidelines used in industry or by certification authorities such as US FAA (Federal Aviation Administration) or European EASA (European Aviation Safety Agency). The standards are accompanied with several advisory guidelines that address particular engineering aspects, like for example DO-331 – Model-Based Development and Verification, or DO-333 – Formal Methods.

One of such documents are guidelines on change management, which focus on characterization of changes with respect to their impact (major, minor), and types with respect to impact on non-functional system requirements [FAA00]. The document is released by the FAA, for the older standard DO-178B, and allows the engineers to easily conduct the impact of changes and to identify whether required change request has an adversary impact on functional safety [Bre05]. The document provides a list of properties or non-functional requirements that are commonly affected by changes in software.

2.2.4 Discussion

The main challenge facing change management in the context of safety-critical systems is to find a trade-off between costs, i.e., especially costs related to re-engineering, and the characteristics or the complexity of supported changes. As introduced by the SEI Initiative on Dependable System Upgrade [SEI01], the design for changes and an adequate process support are the key success factors for the effective change management. To this end, the key facts about related studies described above can be summarized as follows:

- *Lack of a runtime support to perform changes.* Many change scenarios intended to repair systems from certain faults can be performed more cost-effectively, if they could be conducted on the site, e.g., in the system maintenance mode for example, without having a need to re-compile the complete software. However, current commercial real-time operating systems (RTOS) for safety-critical applications do not provide such a support. Some linking models mentioned in this section, i.e., the model introduced in the THINK component framework, or some new linking concepts in AUTOSAR or RECOMP project, do not consider the certification issues associated with such mechanisms. The behaviour of the most standard dynamic linkers is not predictable, and is difficult to verify and therefore difficult to certify. Since such mechanisms are an integral part of the operating system, they have to be developed according to software safety regulations, like any other RTOS services.

- *CBSE and CBD as supporting paradigms.* As introduced in several studies, including Adler [ASTPH11], Montano [Mon11] and Oertel [OKB14], CSBE allows to perform simple, but effective changes. With loose coupling between components, the essential parts of software architecture can be maintained, in particular, the application software.

  Further, as proposed by Adler, separation of concerns can help to simplify the specification of various non-functional properties. He splits the functional view that allows system designer to define basic component functions, whereas so called adaptation view allows them to focus on change scenarios for each component, thereby leaving all other details hidden.

  Last, CBSE hierarchical composition helps in making refinements with fine-grained steps from requirements to atomic (software) components. As mentioned in contract frameworks in Section 2.1.5, stepwise refinement allows to setup relations between individual levels in component-based design, which may help in analysing conformance of implementations with requirements.

  In summary, CBSE in conjunction with CBD and their views can provide sound technologies to effectively analyse the impact of various change types, from their cause in (software) components to the corresponding specifications in the top-level requirements.

- *Supporting processes provided by safety standards.* Most safety standards provide guidelines on how to conduct change management. To this end, they provide very rough descriptions of processes, work products that have to be provided, and set some requirements and techniques that have to applied. These guidelines can help in taking decisions on defining and planning changes and their concrete processes. Especially, the FAA supplemental guidelines for the DO-178B standard, which list a common non-functional requirements, may help in building a common repository of properties or contracts that can be further used to improve and, in some cases, to automate the support for the change impact analysis.

2.3 Modeling and Analysis of Safety-critical Embedded Systems

This section provides an overview of modeling and analysis techniques commonly used in engineering of embedded systems. In particular, component-based systems are targeted here. Modeling and analysis in context of CBSE is mainly focused on (i) characterising software components in an isolation, i.e., on their properties that capture portion of non-functional system requirements and that represent a kind of quality stamps for software components, and (ii) on trying to reason about the system level properties or the top-level requirements out of such individual, but composed, properties. Many existing techniques associated with compositional modeling and analysis are based on formal methods (at least on the specification level).

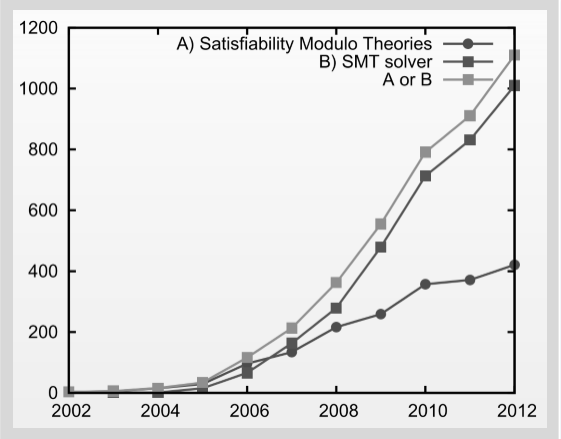


Figure 2.9: Expansion of problem solvers in the literature (number of publications) [Bar13]

Although the use of formal methods in the practice is still rare, their expansion in the last years showed that the industry cannot answer on many relevant challenges in the engineering. Figure 2.9 illustrates the situation that shows the expansion of approaches that use automated problem solvers (proof tools) to analyse systems in various application fields (tools are described later). This is to some extent motivated by the modern safety standards such as ISO 26262 and DO-178C (its advisory document, DO-333 – Formal Methods) which explicitly recommend the use of formal methods. The reason is that today's complex, safety-critical systems cannot be any more considered as a whole, in a classical top-down refinement process, but rather as a composition of parts, and currently, the industry is not enough supported by the sound theoretical background and practical tools to perform such compositional construction and analysis.

In the following, the section first gives some introducing material about different system modeling techniques to setup the context. Thereafter, the summary of related component technologies that provide specific modeling and analysis support is provided.

2.3.1 System Modeling: An Overview

Different system models with respect to communication, computation, time and data are briefly introduced in Section 1.1. Each of these models require specific modeling notations

to be analysed or executed. For example, in the continuous-time model the system behaviour is represented in form of equations over continuous variables. In simulation tools, such as Matlab Simulink for example, the dynamics of the system is represented using differential equations over real-numbers [Kaz09]. In the context of formal modeling and analysis, different models for describing system behaviour are used. The most commonly used models fall into the following two groups [AFPdS11]:

State-based Modeling. The behaviour of the system for this class of modeling notations is represented in form of *states* and *operations* or *actions* that perform changes on such states.

Automata-based Modeling. Automata are commonly used notation to model synchronous, reactive, concurrent and communicating systems. In the standard form, Finite State Machine (FSM), the system behaviour is defined in terms of states and transition functions (informally). This basic notation allows to model typical sequential systems. To date, several extensions of automata have been proposed, in order to model and to analyse the essential properties of (distributed) real-time systems. Timed automata and hybrid automata such as priced timed automata, which allow to model and to analyse dynamic resource consumption, are examples here. The extended versions of automata are supported by several recent model checking tools.

Abstract State Machines (ASM). ASM is a modeling notation that describes the system behaviour in terms of states, state transitions, and rules on how such transitions shall be activated. In contrast to Finite State Machines (FSM), it provides a more precise description of the system behaviour, and can be used to define different abstraction levels, with refinement facilities that enable to synthesize high-level abstraction of the system down to implementation model that can be in turn translated to concrete notation in some programming language for example.

Set and Category Theory. This class of modeling is similar to ASM, but in contrast, states are represented as functions, mathematical relations or sets. The abstraction level is usually close to the implementation so that synthesis to concrete programming language is possible (e.g., like in some VDM (Vienna Development Method) tools [AFPdS11]).

Axiomatic Modeling. The system behaviour is specified using the data model only. This data model corresponds to a collection of data types, their instances, possibly inter-related, and a set of constraints or axioms on those variables. A very important class of system modeling notations here are logic-based languages. They are used as a supporting modeling technique in this thesis to realise necessary system modeling tasks (see Section 3.2).

2.3.2 Specifying System Properties

To analyse the system described in one of the notations introduced previously, the corresponding formulations of certain system properties to inspect have to be set. For automated analysis, these formulations are typical expressions that rely on logic formula

(see Table 2.3). The corresponding tools, proof systems or model checkers, inspect the behaviour of the system model with respect to given formula (property).

Using a collection of logic languages from Table 2.3, many different system properties can be specified. For example, *safety property*[7], which has a usual format that "nothing bad will happen" [Hen92], is a common specification for embedded systems, among few others. The word "bad" can refer to a system failure, caused by the fault when system reaches certain, invalid state, or in axiomatic models, when certain value is not within valid intervals for example. Various system properties, including safety, liveness, and real-time, can be represented in form of states, events, and transitions, using quite similar language constructs and patterns. Figure 2.10 shows a collection of commonly used properties for the formal specification, that are particularly designed for automata-based models. However, some of the properties are also applicable in some axiomatic system models.

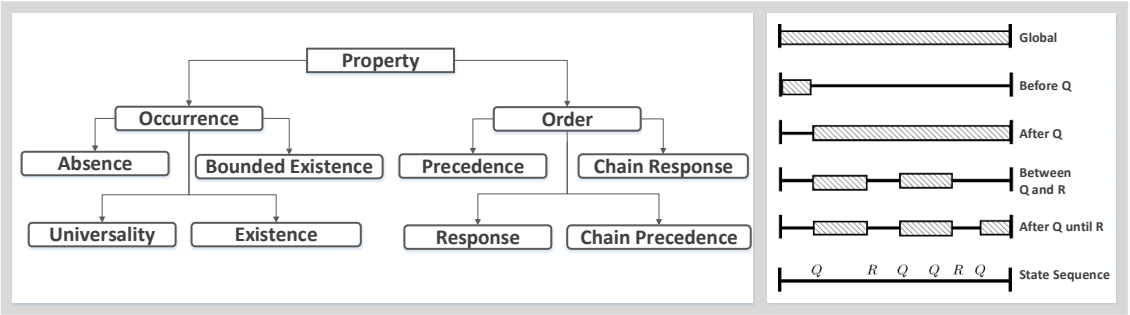


Figure 2.10: A taxonomy of property specifications (left), and different scopes for events used in properties (right) [DAC98] (Research Group at Kansas State University)

Basically, these properties can be classified into the following two categories:

Occurrence : Properties which describe the occurrence of events or states. There are several refinements of this category that generally include some temporal quantifiers: *never* (Absence) – certain state will never be reached or event will never occur, *always* (Universality) – state property holds forever, *eventually* (Existence) – system has at some point of time certain set of events and states, and Bounded Existence – certain event or state must occur for a given number of times.

Order : Properties which describe the order of occurrence of specified events and states (if there are multiple events and/or states specified). As for occurrence properties, there are few refinements (see Figure 2.10)[8].

These basic forms of properties can be represented by many property specification languages. For example, Contract Specification Language (CSL) that was developed in the context of the SPEEDS project [PHB+09] provides a set of specification patterns for components contracts, which are derived versions of properties from Figure 2.10. Contracts in SPEEDS approach are customized to specify various temporal properties on timed automata models. In contrast to SPEEDS approach, which targets specific types

[7]Not to mix with the notion of functional safety.
[8]Meaning of these properties can be observed from their names.

Specification	Description
Propositional logic	Logic formulas (boolean expressions) with boolean operators (e.g., $3 < 5$).
First order logic	Contains in addition functions, predicates and quantifiers. Evaluation of formula is based on values of operands, e.g., $x < (x + 1)$.
Higher order logic	Contains quantifiers over more complex structures (e.g., functions).
Temporal logic	Include notion of time (see Fig. 2.10).
Derivatives	Instance of SAT (Boolean Satisfiability Problem), SMT (Satisfiability Modulo Theories), etc.
Analysis Technique	**Description**
Proof Tools	Automated theorem provers (e.g., constraint solver), Proof assistants.
Model Checking	In contrast to proof tools, they can verify finite-state systems.

Table 2.3: An overview of some common formal specification types and standard analysis techniques [AFPdS11]

of embedded systems, there is a number of other languages which target different system models and more general scenarios, such as IEEE Property Specification Language (PSL) [IEC10b] for asserting hardware design models and various property specification languages for COMPASS Modeling Language (CML) for analysing component-based systems [Rid12]. Finally, some approaches build properties from basic operators to form logic expressions from Table 2.3, cf. [SVP09]; some of them use extended operators such as function symbols in expressions to define resource constraints in SMT logic for example [Mon11].

2.3.3 Related Component Technologies

Modeling and analysis techniques discussed above are an integral part of many component frameworks developed for safety-critical systems. The differences among those frameworks are basically reflected in a formal model they consider, and analysis capabilities, i.e., which parts they consider when composing a system (behaviour, data models, refinement of that behaviour, etc.). In the following, an overview and comparison of the most recent and relevant component frameworks is given (see Table 2.4). The comparison is made based on relevant attributes that must be considered to provide strong traceability support from individual components (implementations) to the top-level requirements. This support is, as introduced in Section 2.2, important for the change management and thus, provides a basis for the work in this thesis.

The basic principles of Contract-based Design (CBD), proposed by Benveniste et al. [BCN+12], Sangiovanni-Vincentelli [SVDP12], and Damm et al. [DVM+05], are used here to characterize the traceability support. As introduced in Section 2.1.5, CBD provides basic rules or operators that component-based systems have to support, to rigorously enable building systems out of individual (software) components. The following are the attributes used to characterize the modeling and analysis capabilities of considered component technologies:

Composition. Support for the formal syntax and semantics on building hierarchical composite systems.

Abstraction and Refinement. Ability to represent the system in different abstraction levels in a formal way, and to link those abstraction levels (refinement).

(Platform) Mapping. Ability to map the functional system design to its execution platform (e.g., in case of software components, the platform may correspond to an embedded system in terms of resources).

Views. Ability to represent different views of the system, with respect to non-functional system properties, and to link those views.

Formal foundation for these attributes is essential, in particular, when changing parts of the system, or even some aspects, to precisely identify impacted system components and requirements. In the following, some recent projects targeting CBSE for safety-critical systems are briefly introduced (the complete list is provided in Table 2.4).

MEDEIA Model-Driven Embedded Systems Design Environment for the Industrial Automation Sector, MEDEIA project, is targeting development of embedded systems in Europeran industrial automation sector, by utilizing synergies between MDE and CBSE paradigms [SRH+09]. The MEDEIA approach provides a common model (meta-model) that comprises the integration of the component model, i.e., the Automation Component Model (ACM), and the model for the corresponding execution platform (hardware).

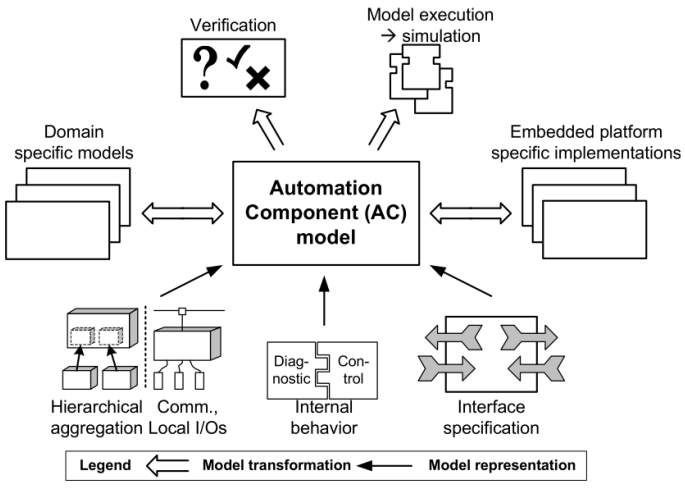


Figure 2.11: Automation Component Model (ACM) according to MEDIA approach [SRH+09]

Figure 2.11 shows the conceptional view of ACM. Basically, the approach does not provide semantics for composing ACM software components and their refinements, and is mainly focused on modeling, rather on analysis.

SPEEDS Heterogeneous Rich Components is a model-based approach for rigorous development of component-based systems, developed in the scope of the SPEEDS EU

Apprach	C	R	M	V	Sys. Model	Comments
BIP [BBB+11]	x		x		automata (FSMs)	Composition of FSMs (behaviour, interaction, priority). Platform mapping by code generation.
Palladio [BKR09]	x*		x	x	axiomatic models	*Not analysis, but performance prediction (use of solvers for stochastic processes).
HRC SPEEDS [PHB+09]	x	x		x	hybrid automata	Formal model for composition, refinement, and views. Support for hybrid system models: continous, discrete; timed and untimed.
ProCom [VSC+09]	x		x		FSMs, timed automata, axiomatic models	Composition: syntax only. Different add-ons in context of PRIDE environment [BCF+11] (e.g., resource analysis in REMES [SVP09], attribute framework [SvCC09], synthesis [BC11]).
SaveCCM [CHP06]	x				timed automata	Behaviour of atomic and composite components as timed automata (standard semantics for composites).
RCM [DVM+05]	x	x	x	x	automata (interface automata [dAH01])	RCM provides formalism incremental, vertical analysis, horizontal composition, and platform mapping. Multiple views of NFRs are supported.
Rubus [HMTN+08]	x				–	Composition: only syntax.
X-MAN [LNRT12]	x	x			automata, axiomatic models	Uses model checker and proof tools for analysis.
ComFort (PECT) [CISW05]	x	x			FSMs, axiomatic models	Uses C code as input to generate FSMs and processes.
COMDES II [KSA07]	x	x	x	x	hybrid automata	Synthesis views (platform independent, implementation, and deployment aspect).
Autofocus3 [KRSV13]	x	x	x	x	FSMs, axiomatic models	Provides functional, logical and technical (deployment) view.
Frescor [MAP+07]					–	No formal semantics. Contracts are defined for resources, and evaluated at runtime.
CESAR [PSS+13]	x	x	x	x	automata, axiomatic models	A comprehensive modeling support for component-based embedded systems. They underlying formal notation is based on X-MAN and RCM component models.
RECOMP [PTV+13]	x	x	x	x	automata, axiomatic models	CESAR extensions for functional safety.
COMPASS [Rid12]	x	x			axiomatic models	Different types of semantics for composition and refinement. The underlying model: process algebra.
VEST [Sta01]	x*		x		–	*No formal semantics.
MEDEIA [SRH+09]			x	x	–	Only syntax for platform mapping.
DECOS [Gru04]					axiomatic models	No formal semantics (composition, refinement). Allocation and scheduling using constraint solvers.

Table 2.4: An overview and comparison of component technologies according to their modeling and analysis capabilities (**C** – composition, **R** – refinement, **M** – platform mapping, and **V** – views)

project (SPEculative and Exploratory Design in Systems Engineering) [PHB+09]. The essential part of the approach is the comprehensive modeling and analysis support, which rely on CBD. The formal model of contracts is defined that provides important background for rigorous system design: composition of contracts taking in account the atomic behaviour of corresponding components, refinement of the system design, in terms of behavioural refinements, and explicit definition of views that allows to focus on individual non-functional aspects of the system, such as real-time, timing, and behaviour for example. The approach provides the thorough modeling support on top of hybrid automata, and therefore allows to model timed and untimed system models, but also to mix continuous and discrete system models.

The concept of "components" goes beyond the traditional CSBE concepts (software components, framework, and component models), since "component" may refer to any architectural part that may perform certain system function (hardware, software, mechanics, but also plant models).

PRIDE PRIDE is a modeling and analysis ecosystem for the ProCom component model – i.e., ProCom Integrated Development Environment [BCF+11]. It provides a number of different modeling facilities, with the objective to rigorously model, synthesize and analyse component-based embedded systems, from their abstract definition to the synthesis on RTOS platform. Like in many formal component models, syntax is provided on defining composite and atomic software components, their interaction and execution. On the other side, semantics is provided in form of external add-ons, such as the attribute framework [SvCC09] – allows to model the distinct non-functional system properties, REMES [SVP09] – resource analysis framework, and synthesis semantics on real-time tasks proposed by Borde and Carlson [BC11]. The underlying modeling technique are FSMs, timed automata, and various axiomatic models, on which various properties can be analysed, including value intervals on ports, resource consumption, among many others.

CESAR Cost-efficient Methods and Processes for Safety-relevant Embedded Systems, CESAR project, targets rigorous development of safety-critical embedded systems by utilizing synergies between MDE, CBSE and various analysis, test and verification techniques [PSS+13]. It provides a common model (meta-model) that captures various aspects or views of embedded system, and techniques and tools to synthesize the system through stepwise refinements and fusions of different views.

Figure 2.12 shows an overview of the CESAR approach. The system is fragmented into different views, each corresponding to certain abstraction level: operational, functional, logical and physical. In each of these views, CESAR provides distinct modeling and analysis facilities. For instance, modeling and analysis of non-functional system properties is provided on functional view. In this view the approach utilizes the formal semantics for compositional modeling and analysis of RSC and XMAN component model [LPC+13].

Except of formal modeling and analysis, different analysis facilities are provided on other levels – for example, tools for schedulability analysis are used to map the functional and the physical view. Further, a modeling means is provided to define

artefacts of safety engineering, for example elements required for the hazard and risk analysis and their traces to design elements.

In addition to technical aspects, the CESAR approach provides the lifecycle support that manages all these views, and corresponding modeling and analysis techniques. In summary, CESAR approach is a distinct engineering approach for developing safety-critical, component-based embedded systems.

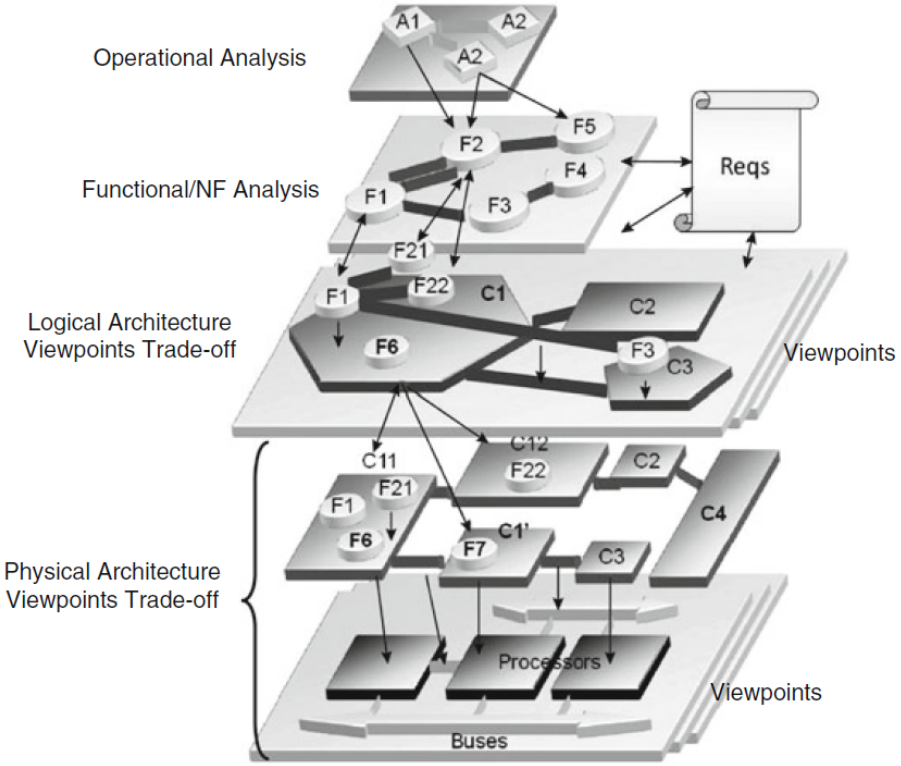


Figure 2.12: Development lifecycle for component-based embedded systems according to CESAR approach [PSS+13]

RECOMP Reduced Certification Costs Using Trusted Multi-core Platforms, the RECOMP approach, is a successor of CESAR, and inherits some features of the AutoFocus3 component model. In contrast to CESAR, the RECOMP approach provides support for defining data relevant for certification of individual components, such as Safety Integritiy Level, traceability links to tests and evidence, and other modeling elements related to safety engineering. Further, it extends the configuration and deployment support of Autofocus3 to multi-core embedded systems [PTV+13].

COMPASS Similar to the strategy of PRIDE and SPEEDS, but with the broader scope, the COMPASS project [Rid12] targets model-based development of System of Systems (SoS) [Mai98]. For this class of systems, the compositional reasoning is a first class entity in systems engineering, and components are, as for SPEEDS, software, hardware, mechanics, physical processes, or combinations of them. To provide sound modeling and analysis support, COMPASS approach relies on Unifying Theory of

Programming [HJ98] -– a set of different formalisms to enable strong formal semantics for systems.

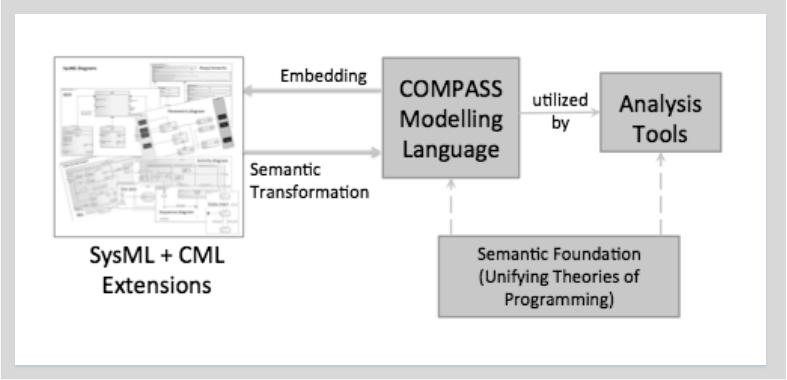


Figure 2.13: A COMPASS approach to modeling and analysis of component-based systems (COMPASS Homepage: http://www.compass-research.eu)

Figure 2.13 shows the overview of the COMPASS approach. The system modelled using SysML (System Modeling Language) is enriched with contracts, which are specified using COMPASS Modeling Language (CML). The CML provides the syntax and semantics for SoSs that rely on some axiomatic system models, in particular, process algebra (e.g., Communicating Sequential Processes, CSP), and related refinement calculus [HMC13]. Basically, such system models are more process-centric, i.e., they describe the interaction between heterogeneous system components, while the formalism for building compositions (syntax, semantics) are provided by CML. Thus, the COMPASS analysis ecosystem, i.e., the Semantic Foundation block in figure, allows to analyse composition and refinements of data and behaviours of communicating components (processes).

Other Component Technologies In addition to introduced projects that target the use of CBSE for the engineering of safety-critical systems, a number of different approaches have been proposed, with mainly similar goals and directions. Some of the approaches are using (hybrid) automata to define system models for the purpose of analysis of non-functional system properties [AM07, CB08], safety requirements [OR13, OKB14] or analysis of the adaptation behaviour [ASTPH11], whereas, on the other side, some of them are using axiomatic models, in particular, constraint solving techniques to analyse adaptation behaviour with respect to resource requirements [Mon11], deployment constraints [BSAB14], or combinations of both [PV14]. In the end of this landscape, there are some approaches that do not rely on formal analysis but only on formal specification [PG13]. Except of difference in the representations of the system model, these approaches differ in strength of the modeling support for CBD attributes. For example, the Choi and Bunse [CB08] define a formal semantics for composition and refinement operators between component specifications and realizations, whereas Montano [Mon11] does not consider these operators. Further, in contrast to all mentioned approaches, Panunzio and Vardanega [PV14] provide a component model with comprehensive process support and ability to define various system views.

2.3.4 Discussion

During the last decade, a considerable number of different approaches relying on CBSE have been proposed, with the goal to enable rigorous development of safety-critical systems and to better utilize CBSE principles for those systems. In particular, several research projects have been realized, and have provided interesting results, both in applying CBSE and related lifecycle to safety domains, and in providing various analysis methods for component-based systems. The following are the main key points of the introduced studies:

- *CBSE, CBD and MDE as supporting paradigms for rigorous development.* The issues targeting system development from individual (possibly heterogeneous) parts and related compositional reasoning are present since the beginning of CBSE era and are today one of the major challenges facing the engineering of safety-critical systems. As proposed in several projects, in particular, SPEEDS, CESAR and COMPASS, the synergy between CBD and MDE is the promising approach to cope with such challenges. Model-driven engineering allows to manage system complexity, in terms of capturing its different aspects and abstractions in a holistic way, while CBD allows to setup the links between those aspect, abstractions, and to rigorously synthesize system designs. A very important concern the approaches are targeting here is the development of the formal semantics to enable different reasoning tasks, in particular, building hierarchical compositions, design refinement, platform mapping, and fusion of different system views.

- *CBSE: From Software Engineering to System Engineering* One important concern that could be observed from SPEEDS, CESAR, and COMPASS approach is that applied engineering principles for component-based systems go beyond the traditional CBSE. Components correspond in these approaches to any part of the system, especially in COMPASS, where a system is a composition of heterogeneous communicating sub-systems. Considering not only software, but rather arbitrary system parts as components, is to some extent motivated by the increase of system complexity, and thus mandated by some standards. For example, in automotive industry, the supplier chain is sharing different kinds of system functions with the manufacturers, including software layers, or complete devices. The SEooC principle defined in the ISO 26262 standard, Safety Element out of Context [SBBK12], is one of the first standard-based guidelines that help in developing such supplier parts, and for manufactures, to compose them together. However, such guidelines are just high-level description for the involved roles. The results of the projects introduced here provide therefore some concrete techniques on how to build system of systems using existing principles of CBSE.

- *Adaptation of existing formal methods to component-based systems.* Analysis techniques used by introduced component technologies mainly rely on existing formal methods and tools, which are in turn adopted to formal model of contracts, and their related semantics. For example, the BIP framework utilizes solvers for Mixed Integer Linear Programming[9] to analyse composition of FSMs; X-MAN uses CBMC model

[9]MILP solver homepage: http://sourceforge.net/projects/lpsolve

checker[10]; CESAR project uses Isabelle proof tool[11] to analyse processes in system of systems environments; Approach by Montano [Mon11] uses the Choco constraint solver [YVNC08] to analyse resource availability in IMA architectures, etc. The adaptation is basically done by defining a template for the problem statement that describes the contracts and related properties using the notation of the considered tool.

- *New directions: integrating CBSE and Safety Engineering* From the projects CESAR and RECOMP, it can observed that the trend in research is directed towards an integrated system and safety engineering, i.e., providing modeling means and necessary traceability support. First steps have already been done, for example the traceability between design elements and tests (evidence) and safety lifecycle in CESAR. Currently, this topic is handled in the context of SafeCer [SMLA13] and OpenCoss [dlVPW13] projects, which put more focus on reuse and certification for component-based systems by linking the two engineering principles.

2.4 Summary

Related studies discussed in this chapter focus on the change management support for safety-critical systems, in particular, on methods to analyse the change impact on non-functional system requirements (and related properties), and later, they focus on various techniques for the formal analysis applied to these systems. Some recent studies showed that there is a trend towards building dynamic architectures, to improve the flexibility and maintenance of safety-critical systems, as proposed for example in several approaches targeting the automotive sector [MFRV13, MVFR14, AK13, NKA14]. However, much details have not been considered yet, from the technical, process and organizational point of view. In summary, according to the problem statement given in Section 1.2, the following are the key points for improvements targeting discussed related studies:

- *Focusing on specific changes.* Reports on recalls given in introducing Section 1.1 show that different kinds of changes need to be conducted to perform necessary repairs. Different change types have possibly different impacts on development, re-verification and re-validation effort and costs, and therefore, may require different procedures and techniques to conduct repairs. In general there is a lack for such a specific support.

- *Runtime support to perform changes.* As discussed in Section 2.2.4, current operating systems applied in safety domains lack a support to perform changes on software (i.e., linkers, loaders). Some mechanisms, proposed as add-ons to existing RTOSs, ignore to consider safety standards and related software safety regulations. As such, they cannot be certified as part of RTOSs.

- *Relations with safety standards.* Safety standards provide not only processes regarding the configuration and change management, but also they set necessary requirements that have to be followed, and techniques that have to be applied. The

[10] CBMC model checker homepage: http://www.cprover.org/cbmc/
[11] Isabelle tool homepage: http://www.cl.cam.ac.uk/research/hvg/Isabelle

alignment with the standards also gives a feedback about required effort to perform changes, since the supported change types have to defined in the planning phase of the safety lifecycle, and have to be communicated with safety assessors. This is essential for defining different strategies for different change types, and may help to better optimize effort/costs required to conduct changes. However, recent approaches ignore considering these regulations in general.

2.5 Thesis Objectives

The objectives of the thesis are oriented towards providing a solution that can answer the questions on the three issues outlined previously. These objectives are the following:

Objective 1: Identify supported change types (trade-off analysis). As mentioned in the previous section, different types of changes may generate different costs, and in the worst case, may require the complete system re-verification, re-validation and finally the re-certification. This decision additionally depends on applied safety standards. Therefore, regulations of standards have to be analysed, and a trade-off has to be found between possible types of changes and costs on performing such changes.

Objective 2: Provide a support and corresponding system model to perform changes. Based on change types identified in Objective 1, an adequate support to their performing has to be provided. This support has to include necessary mechanisms and tools, the adequate system model that can support changes (cf., SEI initiative on design for changes [SEI01]), and finally, it has to be ensured that such a support is conform with related safety regulations.

Objective 3: Provide a support to analyse the impact of changes on system integrity. The system integrity must be re-established after performing changes, either in the initial or in the new configuration. A challenge that has to be addressed here is to identify whether and to which extent particular changes may have impact on system integrity.

Chapter 3

Managing Changes in Safety-critical Embedded Systems

A very important prerequisite to address managing changes for safety-critical systems it to ensure that the initially established system integrity cannot be compromised when incorporating changes. This is usually achieved by the extensive verification and validation of affected system components and requirements, or even in some cases, by the validation of the complete system and consequent validation of the functional safety [SS10].

Design for changes, traceability and an adequate processes support are the essential factors that can strongly influence the effort required to perform such verification and validation activities. The approach proposed in this thesis utilises some principles of paradigms discussed in previous sections to enable managing changes while ensuring that system integrity remains maintained. In particular, software architecture, which is according to repair statistics given in introducing Chapter 1 the system component that most frequently undergoes changes and repairs, is defined in a way that certain software components are systematically designed for changes, in terms of their interfaces, incorporated interaction styles and non-functional properties required to characterise system integrity.

The basic principles of Component-based Software Engineering (CBSE) have been utilized here, supported by the Contract-based Design (CBD) paradigm [Crn02,BCN+12]. Applying CBSE allows to define the granularity of changes and interfaces of changing artefacts, in terms of software components, and it also allows to isolate such components in order to support structural changes by incorporating specific interaction styles. On the other side, the information that captures the way on how the individual software components can influence system integrity is provided using CBD. The integrated CBD and specific component-based architecture allow to perform certain, but rather frequent repairs and to analyse the impact of required changes on system integrity.

Figure 3.1 shows an overview of the proposed approach to manage changes. The model of the system design, that captures the application software and essential parts of the corresponding execution platform (e.g., an embedded device and necessary component container or framework), is represented as a hierarchy of contracts. Component contracts structure specific non-functional properties (NFPs) that are relevant when wiring software components into a composition, whereas the platform components are used to analyse the platform mapping, as introduced in CBD Section 2.1.5.

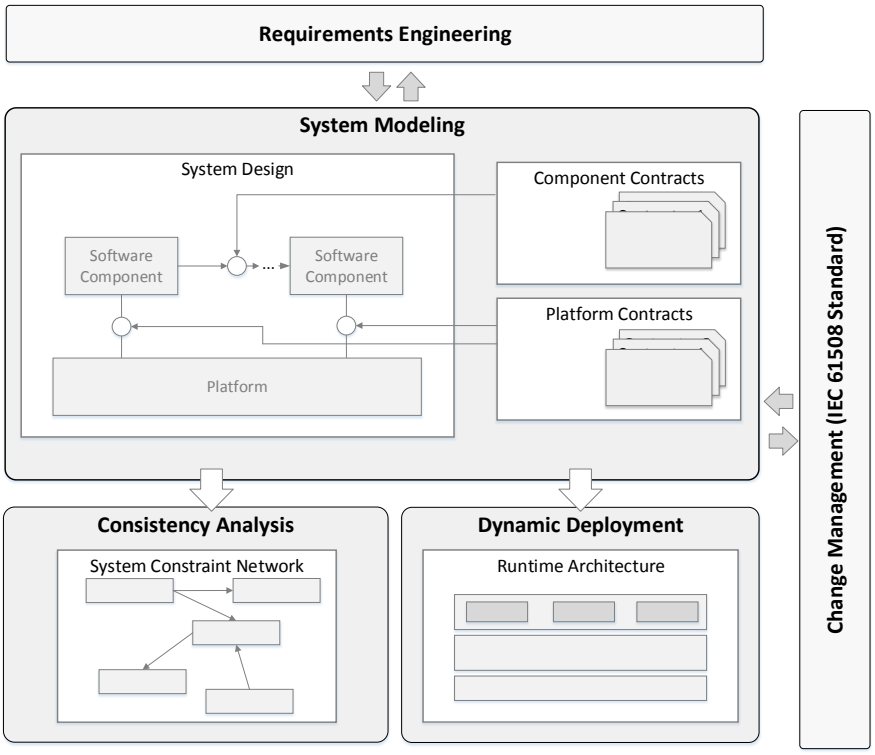


Figure 3.1: Overview of the approach to manage changes proposed in this thesis: (a) system modeling using contracts, (b) analysis of changes introduced into the system design, and (c) dynamic deployment of the analysed system (or software component)

The information that is captured in contracts is originating from the top level system requirements, in particular from safety and other non-functional requirements, such as those that represent resource and quality constraints for example. Using contracts, properties are linked throughout the abstraction levels of the system design to these requirements. In fact, the two introduced types of contract are the essential traceability links between software components and system requirements. More importantly, they hold the system integrity information, because requirements they link are resulting from the risk reduction performed in the conception phase of safety lifecycle (i.e., safety requirements). In this way, violation of system requirements can be identified when some of the contracts are violated, for example due to incorporated changes.

In the remaining parts of the workflow illustrated in Figure 3.1, the system modeled in form of contracts is analysed against violating system integrity (*Consistency Analysis*), and finally, changes are incorporated into the system (*Dynamic Deployment*), according to rules and procedures defined in the change management of the considered standard, in this case the IEC 61508. These are the main contributions of the work in this thesis, and are briefly introduced in the following sections.

3.1 Identifying Characteristics of Changes: The Role of Standards

Changes and necessary procedures for the operation and maintenance have to be defined in the planing phase of the safety lifecycle, as introduced in Section 2.2.3. The benefit of targeting these operation activities systematically is that safety assessors can be early involved in the lifecycle, which in response enables that allowed change scenarios and procedures with respect to requirements of the considered standard can be agreed in advance, and thus the effort when performing such activities can be reduced.

In the following, guidelines on change management defined in the IEC 61508 standard are briefly analysed to identify potential change types that can be supported by the flow illustrated in Figure 3.1.

3.1.1 Supported Changes

Requirements, Measures and Techniques

As discussed in Section 2.2.3, many safety standards provide guidelines on how to support performing some processes in the operation, including maintenance, configuration and change management. Regarding processes related to change management, the IEC 61508 standard sets some requirements to consider, and measures and techniques to apply. Basically, requirements are oriented towards ensuring that the current evidence data (e.g., tests and reports) or in other words, the system integrity is not compromised, or that new integrity has been established, for example degraded integrity due to incorporation of complex changes [SS10]. Regulations are defined both on software and on system level.

The most relevant part here with respect to the effort or costs required to perform changes is actually a decision on when the system has to be re-engineered, in terms of repeating lifecycle activities, and consequent verification and validation activities. Basically, according to certain requirements in the standard[1], there are two options in managing changes: (**i**) if changes to incorporate have an impact on system integrity, i.e., they modify the current evidence data and violate safety requirements, it is mandatory to return to the appropriate lifecycle phase, and to perform necessary re-engineering work, that is often followed by the re-verification, re-validation, and re-certification; (**ii**) on the other side, if changes do not compromise system integrity, there is no need to return to development phases.

This second option is actually the motivation to provide a support for specific, minor, changes that can be managed in a cost-effective way. Many recalls from the field statistics introduced in Chapter 1 can be solved by incorporating such minor changes, for example exchange of faulty algorithms and libraries. Another, very important benefit here is that certain minor changes can be usually planned in the safety lifecycle, including necessary detailed procedures, so that safety assessors can approve their management in the operation and maintenance without having to be involved in assessing that process. The approach in this thesis focuses on managing this type of changes.

[1]Detailed analysis can be observed from **Publication [1]**.

Architectural Support

Software architecture proposed in this thesis utilizes CBSE, the classical form with software components on the top layer, the container (or framework) and underlying OS support. The distinct feature is that it provides a support to load software components dynamically (cf., dynamic software architecture in [Crn02]). The benefit of this feature is that in case of changes incorporated in software applications there is no need to re-build and to re-deploy the complete software system, and therefore no need for the comprehensive configuration management for different software layers, as mandated by the standard.

From the technical point of view, different types of changes can be incorporated into software applications, ranging from simple changes on single software components to dynamic reconfigurations of complete software applications. However, to be able to define all necessary procedures in the planing phase of the safety lifecycle, so that change management process can be conducted as described before, certain design limitation have to be set. For example, if the behaviour of a software component is changed (e.g., like in PINCETTE approach [Cho10]), or the sequence of events that influence the communication between software components (e.g., like in computational model of AUTOSAR [MFRV13]), it is very difficult to analyse the impact of changes, especially on functional requirements, without manual intervention. Except of that, in such cases, it is very difficult to define procedures on how to ensure that changes have no impact on system integrity, because there are no general procedures that can apply to all possible outcomes of such changes.

In the approach proposed in this thesis, supported changes are limited to replacements or exchanges of individual software components only. This has the following benefits: (**i**) procedures to conduct managing such changes can be planned, and are generally applicable to replacements; and (**ii**) the impact on system integrity can be analysed in an automated way. Although replacements are rather simple type of changes, many recall scenarios can be managed by replacing faulty software components.

3.1.2 Process and Responsibilities

The system integrity can also be compromised by incorporating simple replacements of software components. As mentioned in the beginning of Chapter, contracts are used to prevent such situations. Basically, the information that contracts capture has to be defined in the planning phase of the safety lifecycle, and communicated with assessors. The intent of such planning is to show that this information completely supports exchanges of all involved software components, in terms that incompatible software components can be identified and their deployment prevented. A list of commonly used attributes that build this information for contracts is given in **Publication [1]**.

In summary, to enable supported changes from the viewpoint of the change management process, the information captured in contracts must be complete, in terms of being able to analyse the impact of such replacements on system integrity, on the one side, and on the other side, the engineering has to ensure that targeted software components, i.e., the faulty and repaired ones, implement the same functional requirements.

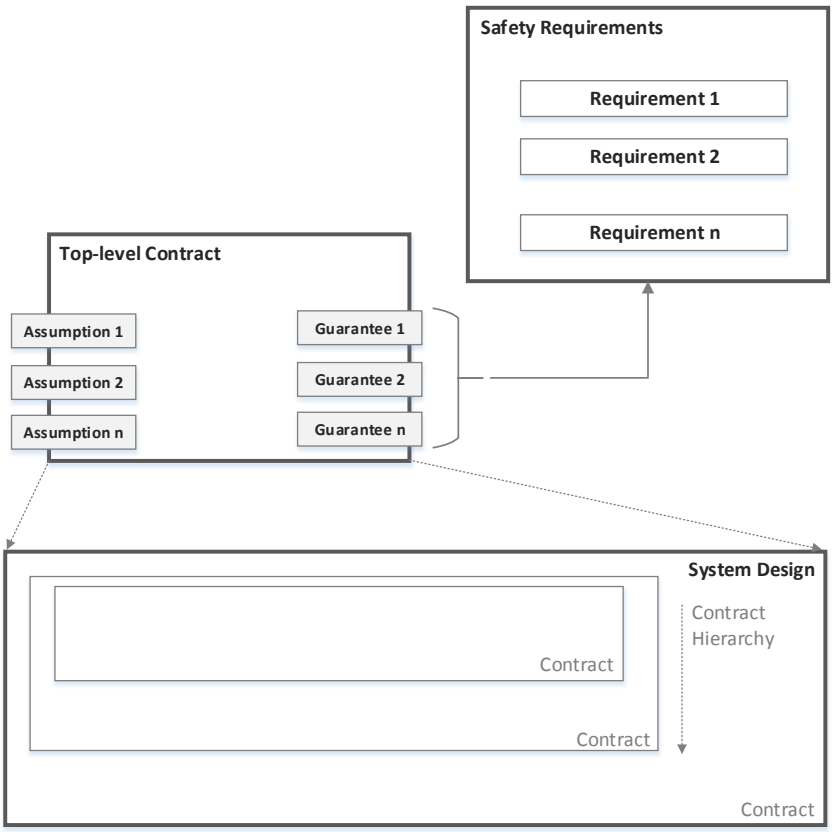


Figure 3.2: Linking system design and requirements using contracts

3.2 System Modeling and Analysis

This section introduces the modeling support for the system, and the analysis of that system built out of contracts. Figure 3.2 shows the system design from Figure 3.1 in a conceptual view, where the source of information that is captured in contracts is illustrated.

Contracts structure properties in terms of assumptions and guarantees [BCN+12], which are expressions, for example represented as logic formula, like first order logic or even informal statements like in some requirements (see Section 2.2 for a list of commonly used expressions). Guarantees correspond to the desired behaviour of a contract, and therefore the desired behaviour of a (software) component or a system which implements that contract. They can be considered as typical requirements. On the other side, in contrast to requirements, contracts provide assumptions that define conditions under which the underlying (software) component or system can guarantee the desired behaviour. Assumptions are very important constructs because they ensure the correct integration between contracts. In fact, they serve as requirements that neighbouring (software) components or layers have to satisfy in order to communicate with related (software) components.

In order to link the individual software components with the system integrity information, every element of the detailed system design, i.e., software components and the platform, is expressed in terms of contracts, thereby forming the design hierarchy out of contracts. Further, the contracts are inter-related so that the top-level contract links all the underlying contracts of the system design, and is related to system requirements, in particular, to safety requirements[2]. Safety requirements, as introduced in Section 2.1.1, set functions that must be implemented by the system in order to target risks identified in the hazard and risk analysis phase. Contracts does not directly specify the information defined in requirements, but instead, the risk information associated with those requirements. This relation between contracts and relation between contracts and requirements is described in the following sections in more detail.

3.2.1 Component Model: Modeling Aspects

As discussed in the previous section, changes with the broad scope, such as those that influence the behaviour or the communication between processes have more disruptive consequences on system requirements, are rather difficult to manage. They also require to construct a component model with the explicit modeling support and means for the analysis, like proposed in CESAR or COMPASS projects for example [PSS+13, Rid12]. Ultimately, such kind of analysis is difficult to plan and to perform in the operation and maintenance phase without involving safety assessors. The benefit of replacements of software components, on the other side, is that their impact on non-functional requirements can be captured in a data model, which is for example used as a basis in several component technologies to analyse various requirements (cf., axiomatic formal models in Section 2.2). Such a data model can be considered as a rough description of the system, from the non-functional aspects, and allows to analyse the system in an automated way in cases when requirements are represented using formal notations.

Elements of the system design illustrated in Figure 3.1, i.e., software components and their platform, are from the modeling perspective defined in way that they can capture necessary information from non-functional requirements. This is achieved by the simple data model that describes software components as a set of data variables. Such a model is typical and commonly used in many data-flow systems, like for example Matlab Simulink or classical IEC 61131-based systems. Here, data variables are used to characterize, on the one side, the syntax of interfaces and of the internal state for software components and the platform, and on the other side, the syntax of the information that is shared between dependent modeling elements. The later are the variables used to set constraints or properties on quality and resources, for example required and provided safety integrity levels, memory and timing margins, operational characteristics and other (for more detailed and formal semantics, see **Publication [8]**).

Contract Model

The information about non-functional requirements on level of software components and their platform is captured in properties and structured in contracts. To support the data

[2] Also other non-functional requirements that do not contribute to the system integrity are linked in the same way.

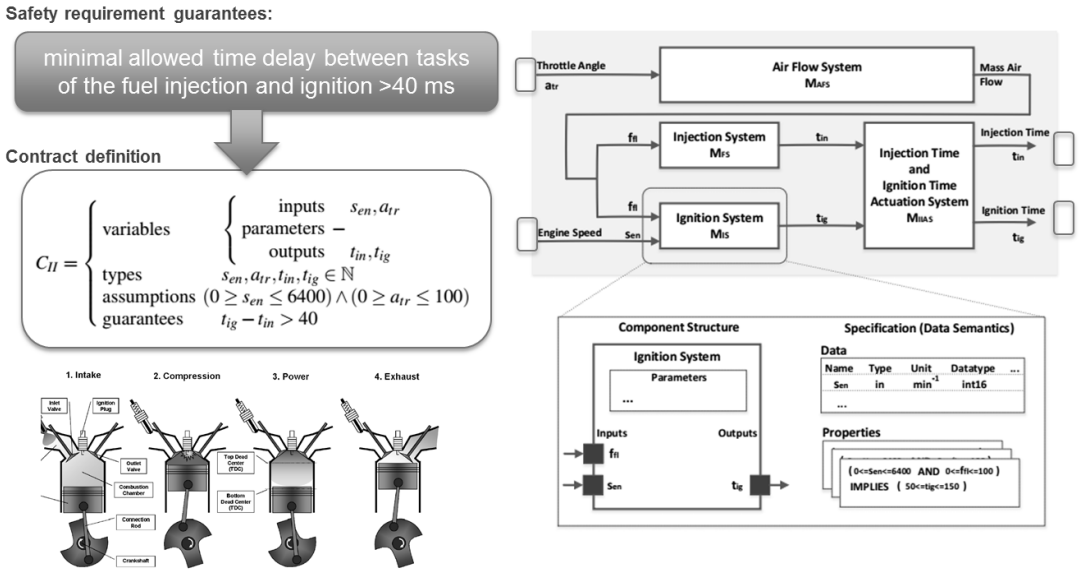


Figure 3.3: Specifying requirements as contracts shown on an exemplary use case: controlled process of the car engine adopted from [Fre10] (left); and software system (right)

model introduced previously, so called *data flow contracts* are applied [BCN+12]. This class of contracts sets in assumptions and guarantees the assertions on data variables of software components (or the platform). Requirements are therefore represented in guarantees as specific logical expressions (more details are following in System Analysis section).

Basically, the information related to requirements that is describing the risk, is defined formally as a logical expression. A trivial example is illustrated in Figure 3.3. On the right, the figure shows an engine controller, implemented as a component-based software system. In short, the controlling software function here is to make a decision on when the engine actuators shall trigger the fuel injection and thereafter the ignition, which is actually nothing else than computing the time distance that depends on several factors, including the engine rotation speed and some information about requests from the driver. Although the function is rather simple, it is critical, in terms that it can damage the engine mechanics if the time distance is too short. Thus, the associated risk here is a concrete consequence when the timing is below the allowed threshold, i.e., $40ms$ in this case. The corresponding safety requirement specifies the safety function that the system has to implement in order to reduce posed risk. Ultimately, the information captured in the contract of the software system is a guarantee that this requirement has been considered, independently on how the system implements related safety function.

To propagate this contract information to the individual, atomic software components, standard contract operators are utilized: *composition*, *refinement*, and *platform mapping* [BCN+12]. Since contracts are self-contained elements, i.e., do not depend on other contracts, these operators provide the connecting features – they inter-relate contracts on the same or different system abstractions, or different models, thereby allowing to analyse the impact of contract properties on the system level.

3.2.2 System Analysis

Different formal methods and tools can be used to analyse the contract system described previously. However, concretely for the system that shall support the introduced types of changes, the following two characteristics are important: (**i**) providing a system configuration that satisfies requirements, and (**ii**) scalability. To target the former, in addition to providing a decision on satisfying requirements, it is often necessary also to provide the concrete values of data variables for which the contracts are valid. For example, the platform mapping can generate different configurations, e.g., due to alternative contracts, and concrete values may be required in deployment plans to configure software application. Scalability is, on the other side, relevant here because the data model can comprise a lot of information that is spread in the contract hierarchy.

To target these concerns, a Constraint Programming (CP) paradigm is used as an underlying analysis technique [Apt03]. For the analysis, the system model defined as a hierarchy of contracts is translated into the network of constraints, i.e., the Constraint Satisfaction Problem (CSP). The CSP is a typical axiomatic model, in which constraints and variables form a problem statement. Basically, the problem solving tool performs the analysis by searching a domain space (variables) so that constraints are satisfied.

To enable the translation of the component-based system into CSP, the corresponding CSP representation of the targeted component model is defined. This definition is a mapping of contracts, their assumptions and guarantees, and operators to constraints and variables of CSP. Concretely, the representation consists of the type system, the expressions used in assumptions and guarantees, software components (and platform), and representation of composite structures. Constraints in CSP are in fact contract assumptions and guarantees, and are with special constraints inter-related with the neighbouring contracts (for more details, please refer to **Publication [6]**).

The proposed CSP-representation of the component-based system allows to automatically translate the system design into CSP and to analyse that design using problem solvers. Another benefit of using CP is that many problem solving tools exist nowadays, with a comprehensive support for defining expressions (e.g., SAT, SMT, functions, etc.), and targeting different domains such as integers, real numbers, and sets. Moreover, some of the tools from the palette are better performing than the other, for particular kinds of problems, for example when solving allocation problems, or when working with specific domains only.

3.3 Runtime Support to Perform Changes

As summarized in related work Section 2.2, there is currently a trend towards providing a support to change or to upgrade software for safety-critical systems, and in response, few approaches have already been proposed. In contrast these approaches, the runtime support proposed in this thesis particularly addresses software safety regulations so that such mechanisms can be applied in the context of safety-certified operating systems. The essential factor that enables this application is a strategy on how the platform and software components are managing their interactions, from the viewpoint of their linking and interconnection mechanisms. This strategy has to some extent the influence on how the software components are designed and how they use their interfaces and communication services.

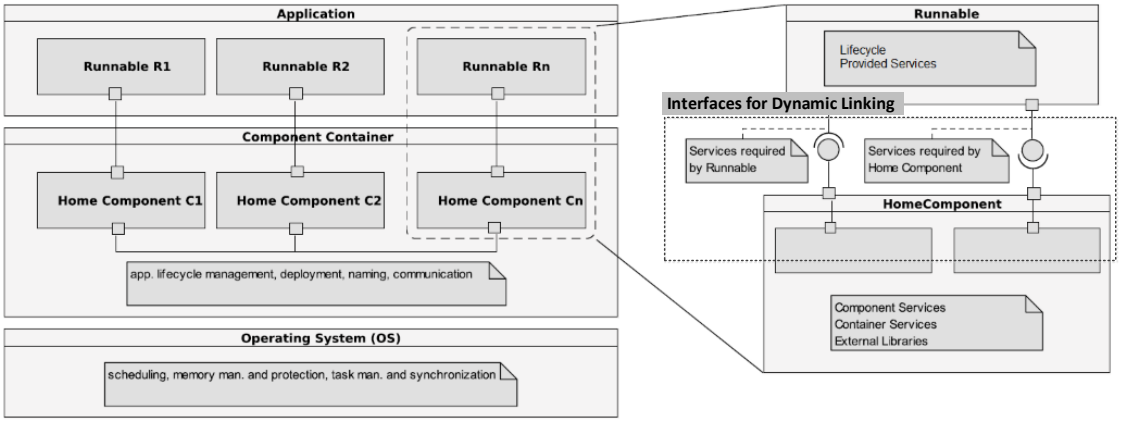


Figure 3.4: Software architecture in the proposed component model (left), and design of a software component (right), adopted from **Publication [4]**.

In the following, some relevant details on such a design are given.

3.3.1 Component Model: Software Components and System Architecture

Figure 3.4 shows an excerpt of the runtime architecture from Figure 3.1, with some details on interfaces between software components and the platform. Here, a software component is designed in a way that functions (i.e., the implementation or business logic, *Runnable* in figure[3]) it provides are the only part that is dynamically linked, and from the physical or deployment view it corresponds to an object file (executable) with relevant interface information, whereas the another part of that component is *Home Component* which runs on the platform, and concretely, it is statically linked with the component container. This element has the role of providing interfaces to the container in order to manage the lifecycle of software components, which is a concept that is commonly used in some general purpose component models, such as the Home Interface construct in EJB or CCM for example [WSO01]. In fact, the container can be considered here as a client using services from software component implementations.

The realization of the binding between interfaces of both parts of a software component is based on function/data symbol addresses or function pointers[4], with minimal degree of indirection, to avoid runtime performance penalties. These pointers in fact connect or bind symbols (interfaces) of software components and component container, as it is done when linking object files and libraries for example. To this end, the linking process has to identify and to bind required and provided interfaces from the viewpoint of software components. In the proposed approach, this is done by utilizing the indirection or routing tables. These tables are provided to software components by the container at runtime, every time when software components are executing and when they require particular interface operations. This is quite different strategy compared to standard dynamic linkers, for example the

[3]Notation *Runnable* is used because of the analogy to the AUTOSAR concept of Runnables [KF09].
[4]Please refer to Kell [Kel07] for different possible realizations of software binding mechanisms.

GNU Linux dynamic linker, which generally do not have static tables, but are computing interface addresses at load-time or at runtime. The reason for this strategy is explained in the following, and has a strong correlation with software safety regulations.

3.3.2 Addressing Software Safety Regulations

Many standards for functional safety provide methods, techniques and measures as recommendations for developing software. Examples are using modular design to manage complexity, limited use of pointers, and static scheduling, among many other [SS10]. Basically, these recommendations shall assist the development to produce software that can be verified, that is easily maintainable and that has predictable behaviour. In the end, it should be possible to provide an evidence that software is functionally correct and that it conforms to safety regulations, in form of test, verification and other kinds of reports.

The problem with standard dynamic linkers for their application in safety-certified RTOSs is that they have a rather complex behaviour for computing the locations of symbols required by object files (i.e., compared to required interfaces of software components). For every required interface (or function and data symbol), the linkers require the runtime memory layout of the software system, and based on specific processor architecture, they compute the final location (address) of that interface. The validity of this final location cannot be easily verified, because the machine code is not analysable, i.e., has no metadata or an adequate reflection mechanism as it is the case with the intermediate code in some virtual machines such as .NET or Java runtime for example. In response, it is very difficult to test and to provide an evidence that such a mechanism works as expected, for possible operational situations (e.g., for different memory layouts). In contrast, using static routing tables as proposed in this work, no computation is required at runtime. The locations of symbols or interfaces are statically estimated and linked with the component container, and can be verified at design-time, with offline tests. These tests are valid during the complete system lifecycle, because their absolute location is always constant (for details, please refer to **Publication [3]**).

Chapter 4

Case Study and Evaluation

An important outcome of managing changes as introduced in this thesis is that specific repairs can be performed on systems, in particular repairs on software applications. Such repairs are supported by planned and systematic procedures, so that in response the extensive communication with safety assessors and consequent re-certification of the system can be avoided. The supported types of changes that are required for repairs are rather simple, but may to some extent contribute in many application fields to save the effort or costs during the systems lifecycle. In the following, some of these fields are targeted and a brief evaluation is performed, to show how much they can benefit from applying the proposed approach. This evaluation is based on the material from **Publication [2]**.

4.1 Objectives, Field Data and Used Metrics

The aim of the evaluation is to show the possible reductions in effort or costs that can be achieved when performing repairs as proposed here.

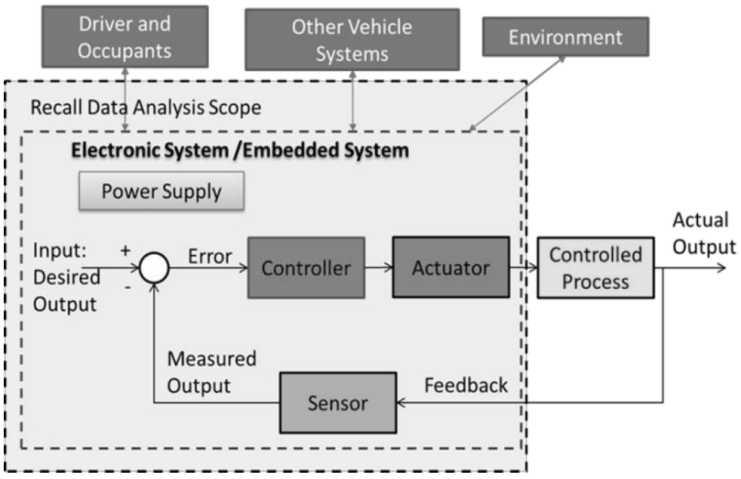


Figure 4.1: Field data: considered scope of analysed system failures that led to product recalls in automotive, [Qi 14]

Recall data for Use Case 1	
Faulty system components	**# of recalls**
Software (control algorithm, flaws in creation, change)	82
Sensor (inadequate operation, change)	52
Actuator (inadequate operation, change)	30
External disturbance	26
Controller hardware (faults in hardware, change)	25
Other	100
Considered period	2002-2013

Recall data for Use Case 2	
Faulty system components	**# of recalls**
Software (software, application, function, code, version, backup, database, program, bug, java, run, upgrade)	778
Hardware (board, chip, hardware, processor, memory, ...)	179
Battery (battery, power, power-up, discharge, charger, ...)	70
I/O (sensor, alarm, screen, interface, monitor, connect, wireless, ...)	41
Other	142
Considered period	2006-2011

Table 4.1: Distribution of recalls according to their cause in components of an embedded system

As a reference case study, field data about recalls in two application fields have been used for consideration. The first case, i.e., the Use Case 1, is targeting the automotive sector, and provides a collection of analysed failures that were caused by different system components, and that contributed to product recalls [Qi 14]. Another case, i.e., the Use Case 2, provides the analogue data collection but for the biomedical engineering sector [AIKR13]. Figure 4.1 shows the architecture of a considered system model, i.e., an embedded system and controlled process, and the scope of analysed failures for the Use Case 1. Generally, failures are collected and classified according to their cause in the signal chain from sensors, controllers, to actuators, including software, hardware and other components such as mechanics, battery, and external influences. This classification is for both use cases summarized in Table 4.1, including a time period used for the analysis. From these tables it can observed that for example 64% of medical devices had to be recalled due to failures caused by software faults. For automotive embedded systems, this rate is about 26%[1]. Finally, the considered studies have also outlined which parts of software contributed to failures, like for example control algorithms in Use Case 1, software application, code, and program in Use case 2, among few other.

The introduced recalls and related necessary repair actions have contributed to the overall costs in system lifecycle for the considered period. The goal of the evaluation here is to estimate the potential reduction in those costs for both studies. To this end, the system models applied in studies are compared with the introduced approach, taking into account some quality attributes that enable the comparison. In the following, the applied metrics for the comparison is introduced, and a report on possible reductions in costs is given.

[1]It should be noted that there is no standard classification scheme for such failures or recalls, and therefore the two studies cannot be objectively compared.

Used Metrics to Compare System Models

To date, a number of different methods and metrics on how to characterize the quality of systems and to compare systems based on quality attributes have been developed. Many of the metrics require details about system components and specific information about process performance (e.g., the effort for developing and changing particular components). The field data introduced previously, in contrast, provides just a rough description of failures, recalls, and mainly lacks standard classification scheme for details about causing components, for example on the level of software architectures. To characterize a system with respect to this field data, the metrics proposed by Bengtsson and colleagues [BLBvV04], i.e., the Architecture-level Modifiability Analysis (ALMA), has been applied. This metrics allows to compare system architectures by considering their capabilities with respect to the modifiability quality attribute. According to this metrics, characterizing the system modifiability is performed in the following steps:

- Develop change scenarios
- Define architecture
- Evaluate scenarios
- Overall evaluation

Change scenarios correspond to concrete activities in the engineering to perform specific types of changes, like adding new functions for example. Evaluating scenarios means characterising their effort or costs with respect to supported system architecture. The ultimate goal of the ALMA method is to collect all supported change scenarios, to evaluate them by estimating their costs, and to estimate the overall costs for the targeted architecture. The final value can be used to predict the change costs for example. Formally, this cost value is defined as follows:

$$C_{next} := \left[\frac{P}{N} \cdot \sum_{j=1}^{N} (size_j \cdot weight_j) \right] \cdot M_{num} \tag{4.1}$$

Parameters $size_j$ and $weight_j$ characterize a single change scenario j by its effort or cost required for the realisation and its frequency of occurrence in given period of time, respectively. The parameter N is used here to get the average value for costs, that is then used to predict the future costs, i.e., the C_{next}. This is done by simply multiplying the average change costs with the expected maintenance interval M_{num}. Finally, the parameter P is a constant, which is used to adjust the final value to specific project performance.

To estimate the reduction in costs, the ALMA method is applied as follows: for the Use Case 1 and 2 the change costs are estimated under assumption that the system model applied is the commonly used one, i.e., without the support to perform changes dynamically. Then, the cost values are computed for both studies again, but now under assumption that the applied system model incorporates the proposed approach to manage changes. The change costs are given as follows (from Equation 4.1):

$$C_{eff} := \sum_{j=1}^{N-1} (size_j \cdot weight_j) + (size_f \cdot weight_f) \tag{4.2}$$

Change scenarios and their volumes			
SNr.	**Change scenario (repair of)**	**size**	**weight**
1	Software	$c_{dev_1} + c_{cert_1}$	0.643
2	Hardware	$c_{dev_2} + c_{cert_2}$	0.147
...	...	...	...
Sum			1.0

Table 4.2: Describing change scenarios according to ALMA method for Use Case 2 (excerpt)

The part of the equation ($size_f \cdot weight_f$) corresponds to costs required to repair the application software, whereas the another part corresponds to costs for repairing the remaining system components from Table 4.1. This separation is used just to easily understand how the cost reduction is estimated later.

In order to get C_{eff}, some assumptions on parameters of the ALMA method are given In the following.

Assumption 1: The ALMA method requires to first define a collection of scenarios in order to evaluate system architectures. A change scenario in this case corresponds to a repair action that needs to be conducted on the system in order to remove faults from a defective system part. According to this scheme, there are 6 different change scenarios for the Use Case 1 and 5 scenarios for the Use Case 2 (see Table 4.1).

Assumption 2: The parameter *size* defines a volume of a change scenario in terms of required effort/costs, and can depend on many factors. Usually, this value is observed from the history, i.e., the experience about the same or similar scenarios, or it can be in some cases rated by the experts. Because the *size* parameter is not known in provided field data, it will be considered as a variable, and defined as follows:

$$size := (c_{dev} + c_{cert}) \tag{4.3}$$

The parameters c_{dev} and c_{cert} correspond to costs of change scenarios related to development and certification. These are normalized values, and show to which extent a particular change scenario impacts the development and certification (or re-certification).

Assumption 3: As mentioned previously, field data are not sufficiently precise when considering details (e.g., software parts and layers). It is, for example, not completely clear how much the applications, device drivers or libraries are contributing to recalls with respect to the overall software. The use cases provide some classification scheme, which is mainly based on keywords collected from recall reports. Because the approach proposed in this thesis targets the application software, it needs to be known how much the application software contributes to the recalls. In contrast to the Use Case 1, the Use Case 2 provides more detailed information, in particular, the *application*-caused recalls are 1/12 of the overall software-related recalls. It should be noted that also the *program*, *function* or *code*, may belong to this category, but, in order to have more precise results with regard the cost reduction, it will be therefore assumed that software applications are contributing to max. 12% of the overall software-related recalls.

UC	# of recalls	# of sw related	Change effort/costs C_{eff}	Cost reduction $(\cdot 100)$ [%]
1	315	82	$0.229 \cdot (c_{dev_1} + c_{cert_1})$ $+...$ $+0.317 \cdot (c_{dev_6} + c_{cert_6})]$ $+0.031 \cdot (c_{dev_f} + c_{cert_f})$	$\frac{0.031 \cdot (c_{dev_f} + c_{cert_f})}{0.031 \cdot (c_{dev_f} + c_{cert_f}) + [0.229 \cdot (c_{dev_1} + c_{cert_1}) + ...]}$
2	1210	778	$[0.565 \cdot (c_{dev_1} + c_{cert_1})$ $+...$ $+0.117 \cdot (c_{dev_5} + c_{cert_5})]$ $+0.077 \cdot (c_{dev_f} + c_{cert_f})$	$\frac{0.077 \cdot (c_{dev_f} + c_{cert_f})}{0.077 \cdot (c_{dev_f} + c_{cert_f}) + [0.565 \cdot (c_{dev_1} + c_{cert_1}) + ...]}$

Table 4.3: Estimated change effort/costs C_{eff} for use cases 1 and 2 and possible cost reductions

4.2 Results

Identifying and Evaluating Change Scenarios

As one of the first steps towards applying ALMA method, change scenarios have to be defined based on recall data and corresponding causing components from Table 4.1. An excerpt of these scenarios is given in Table 4.2. For every scenario, *weight* and *size* parameters are defined. As mentioned, *size* is a cost related to a change scenario, and is consiederd as a variable here, whereas *weight* corresponds to a frequency of occurrence of a given scenario, normalized to the overall number of recalls. For example, 64,3% software-related recalls correspond to 778 recall actions and thus 778 occurrences of a single change scenario.

Estimating Change Costs and Possible Cost Reductions

Change scenarios from Table 4.2 are used to estimate the costs of changes, according to Equation 4.2. Table 4.3 summarizes costs for both use cases, i.e., their C_{eff}. This value is estimated for both use cases, once under an assumption of having a system model without a support to perform changes dynamically, and once with the system model that incorporates the proposed approach to manage changes. Generally, the difference between assumed system models is that the application software in the later case can be repaired without impacting the *size* parameter, i.e., in other words, without impacting the certification costs.

The cost reduction is therefore estimated relative to the C_{eff} of the first system model. Table 4.3 shows the possible cost reduction, in the last column. The concrete distribution of this cost reduction is illustrated in Figure 4.2, with concrete values for the *size* parameter. This distribution in fact shows the cost reduction when considering the ratio between costs required to repair the system from application-caused failures and costs required to repair the system from failures caused by remaining system parts. For example, in the case of an uniform distribution, i.e., when the same costs are required for all kind of repairs (which is rather unlikely in the practice), the cost reduction lies at about 7% (point P1 in the figure), and 3% for the Use Case 1. At the other extreme point P2, there is no reduction at all. This can for example be the case when the first system model can manage the changes on the application software, without having an impact on the *size* parameter. In contrast to P2, the point P3 shows the cost reduction for the case when

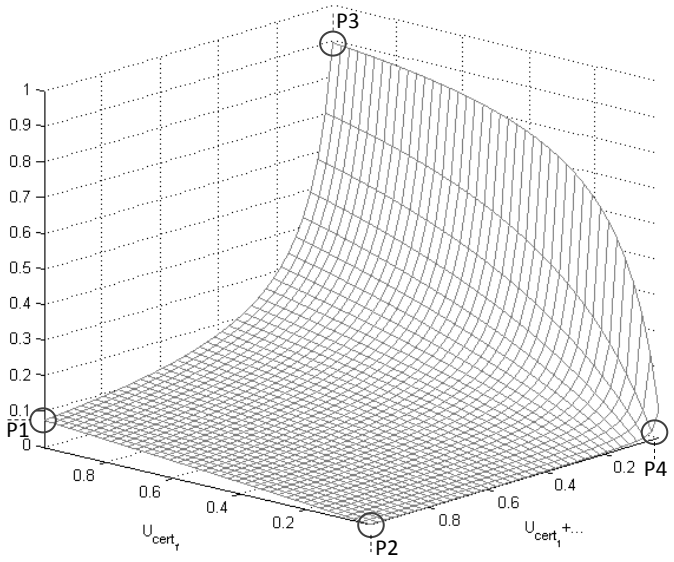


Figure 4.2: Possible reductions in costs for change scenarios related of the Use Case 2

change scenarios related to the application software are the only scenarios that require costs (which is in fact never the case). Finally, at the point P4, there are no costs for any change scenario at all (which is, again, never the case).

Generally, the real concrete scenarios would be expected to be around the point P2, with more focus in direction towards P1, because, obviously change scenarios related to all other components, including hardware, sensors, actuators etc., have more impact on costs than just the application software. This would mean that the cost reduction in some concrete scenarios would likely be below 7%. This, of course, holds for the assumption that 12% of software-related recalls are caused by faults in applications.

4.3 Discussion

In this brief evaluation, it was shown to which extent the proposed approach may help in reducing costs for repairs of safety-critical systems, by considering two application fields, both focused on series production. From the statistics, it could be observed that repairing the application software is just a small excerpt of the overall repair actions, and possible reductions range just from 0% to $< 7\%$, under given assumptions. However, for application fields which focus on mass customization, this reduction is considerable. For example, according to Hommes [Qi 14], 2.6 millions cars were recalled due to failures caused by software, in period of time given in Table 4.1. By applying the proposed approach to manage changes, up to 312000 repair actions or car recalls could be managed in a cost-effective way. For example, a dedicated maintenance channel would be conceivable, that would allow to transmit repaired software components and to upgrade the system, possibly without a need for visiting the service stations and a need for cost-intensive scheduling and planing of time slots with the customers.

One of the most delicate parts of this evaluation is that field data does not provide a

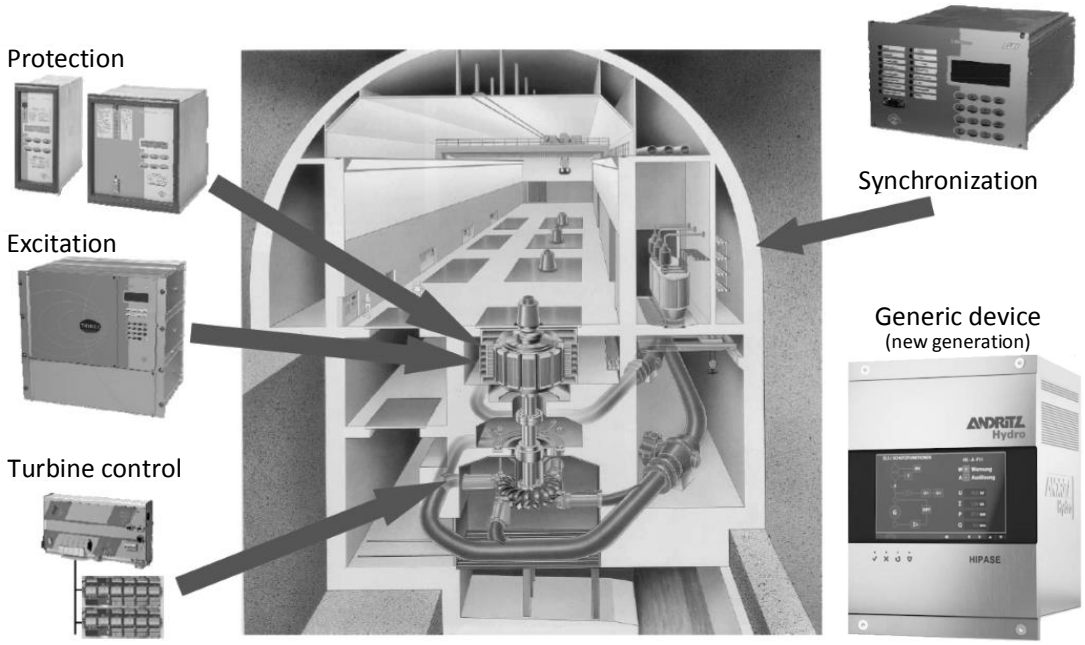


Figure 4.3: Standard functions used for controlling processes in hydro power plants (source: Andritz Hydro)

classification scheme that particularly considers details for different system elements, so that the cause of recalls can be more precisely addressed. For example, the Use Case 1 just maps software caused failures to control algorithms, and in general to software caused failures due to changes or upgrades. These recall distributions are imprecise mainly because of different systems being analysed, their different architectures, functions, and different involved organizations (i.e., car manufacturers) that were considered in the analysis. Further refinement and standardization of such classification schema would bring much benefits in conducting more precise statements about possible cost reductions, even for cases when the concrete costs for individual change scenarios are not publicly known.

4.4 Implementation Status and Applications

This section summarizes the implementation status of individual prototypes developed in the context of the thesis and their concrete application. Note that in the following the focus is given only on contributions of the thesis and not on evaluation and case studies introduced previously.

Application: Controllers for Hydro Power Plants

As introduced in the acknowledgements part, the work in this thesis has been carried out in a cooperation with the Andritz Hydro GmbH company (Vienna), in the scope of the HIPASE project (High Integrity Protection Automation Synchronization and Excitation).

The goal of the project was to develop next generation controllers with generic architecture and abilities to customize applications of the product line to different functions and purposes. Besides, the project goals are put towards providing high quality products with strong emphasis on safety and dependability requirements.

Basically, hydro power plants are used in different configurations to utilize the water energy for producing the electricity. During the last decades, computer-based controllers have been intensively deployed to automatically control, monitor and protect various processes in the plants. Figure 4.3 shows the plant internal structure and the installation including the control functions that are performed by the embedded systems. In summary, these functions are the following:

Turbine Control. Controlling the produced energy by regulating the volume of water flowing into the turbine blades.

Protection. The plant is monitored according to some operational requirements and characteristics, including the behaviour of the current produced and frequency. In cases of deviation from the desired behaviour, the protecting devices are capable to turn the plant into the safe state before any damage can be incorporated.

Excitation. The energy produced by generators is originating from the magnetic field that is produced by rotating electromagnets, which are usually supplied with the current from external excitation machines. This configuration is realized in order to be able to dynamically control the strength of the magnetic field, with the aim to balance the produced and consumed energy.

Synchronization. In many cases, multiple generators are used in a composition to produce the energy and are connected to the distribution network. To realise this scenario, it is very important that they are synchronized with respect to current characteristics. Otherwise, disastrous damages are likely to occur.

Due to different nature and configurations of plants, many possible ways exist on how to realize and to deploy these four functions. The HIPASE new generation devices, shown in Figure 4.3 on the right, provide a common platform for the application software, by utilizing the principles of CBSE. That is, the functions are realized as compositions of software components, which correspond to small functions, such as ones that provide basic logic, arithmetic operations but also some complex functions such as controllers and filters that can be found in the IEC 61131 standard for example.

An important feature of these new generation devices is that they are highly reconfigurable and allow to deploy applications dynamically. To enable this deployment, the operating system support to perform changes proposed in this thesis has been applied. Thus, software components and corresponding interfaces to the container are defined as described in Section 3.3.

Runtime Support to Perform Changes

The proposed runtime support to perform changes has been developed to fit characteristics of the ARM processor family, in particular ARM9, and has been evaluated on the Freescale

i.mx287 Board (ARM926EJ-S)[2]. The applied operating system that runs the application software is SafeRTOS, which is based on functional models of OpenRTOS and FreeRTOS, and is certified according to the IEC 61508 standard for applications in safety-critical embedded systems. The operating system software is certified for the use up to SIL 3.

The i.mx287 has been also used to evaluate the runtime performance, scalability and to identify eventual performance penalties, by comparing the proposed support with the functionally same, but statically linked, software configuration. In response, some performance penalties could be observed, mainly because of the level of indirection introduced between software components and their container, but these penalties are predictable. This can be seen as one of the possible topics for the future work.

System Modeling and Analysis

The modeling and analysis support, as introduced in Section 3.2, are developed following the principles of Contract-based Design (CBD) and Constraint Programming (CP) paradigm, respectively[3]. For the analysis, the Java-based Choco Solver has been applied [YVNC08], to demonstrate the feasibility of using CP. Thus, the system defined in form of contracts is translated into a constraint network using notations of the Choco Solver. Roughly, the implementation can be characterized as follows:

Data Model. Used solver allows to define various kinds of expressions in constraints, including logical, arithmetic, and some functions like computing a sum over real numbers for example (cf., Table 2.3). Besides, it supports few different domains for used variables. In the current implementation, some basic expressions can be defined so that fixed values or intervals can be observed for a single data variable inside contracts.

Contract Operators. Composition, refinement, platform mapping and viewpoint fusion are the contract operators required to build robust designs. The current implementation supports the composition and platform mapping, using the expressions in contract properties mentioned previously. Although the viewpoint fusion is not supported directly (as modeling part), it can be realized by analysing each of the system views independently. The refinement, on the other hand, is to some extent more difficult to analyse automatically, and is a topic for the future work.

One of the most delicate characteristics of tools for formal analysis is their runtime performance and scalability. In CP, the scalability can be influenced by many factors, including the variable domain and used concrete configuration of variables (e.g., ranges, fixed values), the problem statement in terms of variables, constraints and complexity of their relationships, and solver algorithm, among few other factors. The current implementation used integers as domain of variables, to optimize performance. The approach has been evaluated with respect to scalability for different complexities of system configurations, i.e., number of contracts and used properties. The results can be observed from **Publication [6]**.

[2]For details about these ARM-specific characteristics, please refer to **Publication [3]**

[3]System modeling and analysis are not integrated into the development process of HIPASE controllers.

Chapter 5

Conclusion

5.1 Approach Overview

This thesis proposed a systematic approach to manage software changes in the engineering of safety-critical systems, with the objective to reduce the effort/costs associated with repairs that such systems have to undergo in the case of failures in their operation. The problem statement behind this work was motivated by the fact that safety-critical systems currently face a number of challenges in their engineering. In particular, the growing system complexity is one of the major reasons why the development fails to effectively eliminate systematic design faults, which consequently often lead to system failures and in some application fields, such failures are occurring at an increased rate. As showed in some recent reports from the field data, most of those failures are caused by software. The work in this thesis addressed this problem from a different perspective, as it is handled in related studies introduced here. In particular, it is based on the assumption that introducing faults in the development due to complexity issues and a lack of an adequate engineering support becomes inevitable, and that costs associated with such faults can be alternatively targeted by employing systematic approaches for managing system repairs, or concretely, for managing changes required for such repairs. For this class of systems, repairs are usually cost-intensive engineering activities, and often require the intervention of the external safety certification authorities (assessors) to approve the required changes. To this end, the proposed approach provided a solution to target specific repairs, in terms of supported types of changes and necessary procedures to manage those changes. Changes are performed on software, by replacing or exchanging defective software components. The distinct feature here is that procedures required for such replacements/exchanges can be planned in the safety lifecycle, which allows to perform certain repairs without involving external assessors, and thus saving related costs, in cases when system integrity is not compromised.

From the technical perspective, the approach utilized the synergies between emerging paradigms in safety engineering, i.e., Component-based Software Engineering (CBSE) and Contract-based design (CBD), to support managing changes as introduced. CBD was applied to provide a modeling means for designing a system with necessary information about its integrity, in order to be able to analyse whether intended changes can be performed or not, in terms of their impact on system integrity. One the other side,

the architectural and runtime operating system support have been provided to enable performing changes dynamically, i.e., focusing only on defective software components without a need for re-building and re-deploying the complete software. Finally, the procedures on managing changes were aligned with regulations of the IEC 61508 safety standard. This alignment comprises on the one side, the choice on the type of supported changes which resulted from requirements, procedures, measures and techniques defined in the standard, and the system integrity information on the other side, that needs to be modelled in order to be able to analyse the impact of changes. With CBD, such an integrity information can defined on a design level, and can be tracked to the top level system requirements.

The approach was evaluated using two studies which deal with the analysis of failures and related recall actions caused by malfunctioning of embedded systems. The studies provide a collection of recalls from the field data taken from some defect reports in the automotive domain and biomedical engineering. Recalls are classified according to source of their cause, i.e., the system components, including software, hardware, and interfaces, among few other. They have contributed to costs required to repair targeted systems, and the goal of the evaluation was to identify how much of the repair costs can be reduced, when applying the proposed approach to manage changes. Although the studies did not provide a detailed classification schema for failures and recalls, it could be observed from the evaluation results that application fields which mainly rely on a mass customization of products, with predominating software-implemented functions, can benefit from applying the approach presented here. On the other side, repairs on the level of software, in particular on the level of applications as supported here, are just an excerpt of the overall possible defects embedded systems are facing, and managing only this kind of repairs would not bring much benefits in costs for the application fields that release products (devices) in small numbers.

In the end, the application of the proposed approach was introduced. Some parts of the contributions, in particular the runtime operating system support for performing changes, have been deployed in an industrial setup to drive component-based applications for controlling physical processes in hydro power plants (in the scope of the HIPASE project). This runtime support is used to deploy and to reconfigure control software for different functions and different purposes on a common platform, that should be safety certified. Although this domain cannot take much benefits from managing changes as proposed here, the application of the runtime support shows that different use cases can be utilized from the contributions of this thesis and that potential advantages in maintaining software for safety-critical systems can be achieved.

5.2 Future Work

This section summarizes some potential improvement suggestions for the future work. The improvements are on the one side targeting limitations of the implementation features introduced in Section 4.4, and, on the other side, they focus on possible extensions regarding the theoretical aspects of the proposed approach to manage changes.

Systematic way for gathering the system integrity information. The information about system integrity is crucial to enable the analysis of change impacts. This information cannot be completely derived from individual software components or their

contracts, but instead, when integrating or composing those components, i.e., in the system development process. The system developers or integrators usually have to know to which failures the invalid composition may contribute, and then, based on those failures, they may adjust contracts or add some new ones if needed. However, there is no general and systematic way on conducting related cause of failures from contracts, for such a low level design details. In this thesis, the non-functional requirements or attributes defined for the DO-178B standard have been proposed as guidelines for building contracts (see **Publication [1]**), but they are not the guarantee that all possible failures or failure modes for all possible system models have been considered.

A very important support in modeling systems out of contracts would be a systematic method or workflow on how to gather faults that might be introduced when composing contracts, when refining the system design and when mapping the system design onto its platform. The main outcome of this support is that information about possible faults would be more complete and the standard procedure can help to easily conduct the system integrity information that has to be documented in the planning phase of the safety lifecycle.

Analysis Support for Different Types of Contracts. The system analysis applied in this thesis is based on Constraint Programming (CP) paradigm and related problem (constraint) solvers. CP provides a lot of features, with respect to the representation of domains (integer, real, set), their expressions (SAT, SMT, functions, etc.), but it is not always an adequate tool for certain use cases. In particular, the platform mapping in some cases requires more complicated expressions, to solve resource allocation problems for example. Therefore, as also proposed in the COMPASS project [Rid12], a landscape of tools for the (formal) analysis are required to address different system properties or requirements. New methods and tools have to be evaluated with regard to modeling requirements using contracts and runtime performance or scalability.

Extension of Supported Changes. The introduced types of changes provide a support to target rather simple repairs, but in response, they allow to define the necessary procedures in advance in the safety lifecycle, and to perform repairs in a cost-effective way. Addressing particular types of repairs as presented in this thesis could be a promising way to manage costs in systems lifecycle after production. The existing support for changes should be extended, to allow for targeting more complex repairs. In particular, the following two essential issues have to be addressed: (**i**) to define a model that allows for analysis of change impacts, and (**ii**) to define the responsibility of roles, i.e., which parts can be performed without involving assessors and which are necessary to be approved by the assessors on-site.

The above outlined improvement suggestions particularly target the modeling and analysis support, and managing changes in general. The following points are the improvements related to the runtime operating system support for performing changes.

Portability to Other Processor Architectures. The introduced architecture for system and software components is tailored to specific processor architectures. In particular, the interfaces between software components and their container (part of

the platform) are realized by exploiting mechanisms for relative addressing modes[1]. Software components in response can access their required interfaces by using relative locations, for example based on the current location of the program counter. In addition, these mechanisms allow to link some required services directly with the object code of software components, thereby allowing to utilize the optimal runtime performance. However, some processors do not have such a support, and new methods have to be analysed on how this interfacing can be realized.

Runtime Performance Analysis and Improvement. Interfaces between software components and their container introduce an additional level of indirection, and thus, some runtime performance penalties. As shown in **Publication [3]**, these penalties are predictable, but might for some systems be considerable. They mainly depend on the interaction intensity between the container and software components. To this end, adequate design strategies have to be employed to optimize the runtime performance, at least for cases of intensive interactions.

[1]The way on how a reference to some symbol (data, function) is realized in instructions.

Chapter 6

Publications

This chapter lists publications made during the work in this thesis. A collection of publications and their mapping to thesis objectives is illustrated in Figure 6.1.

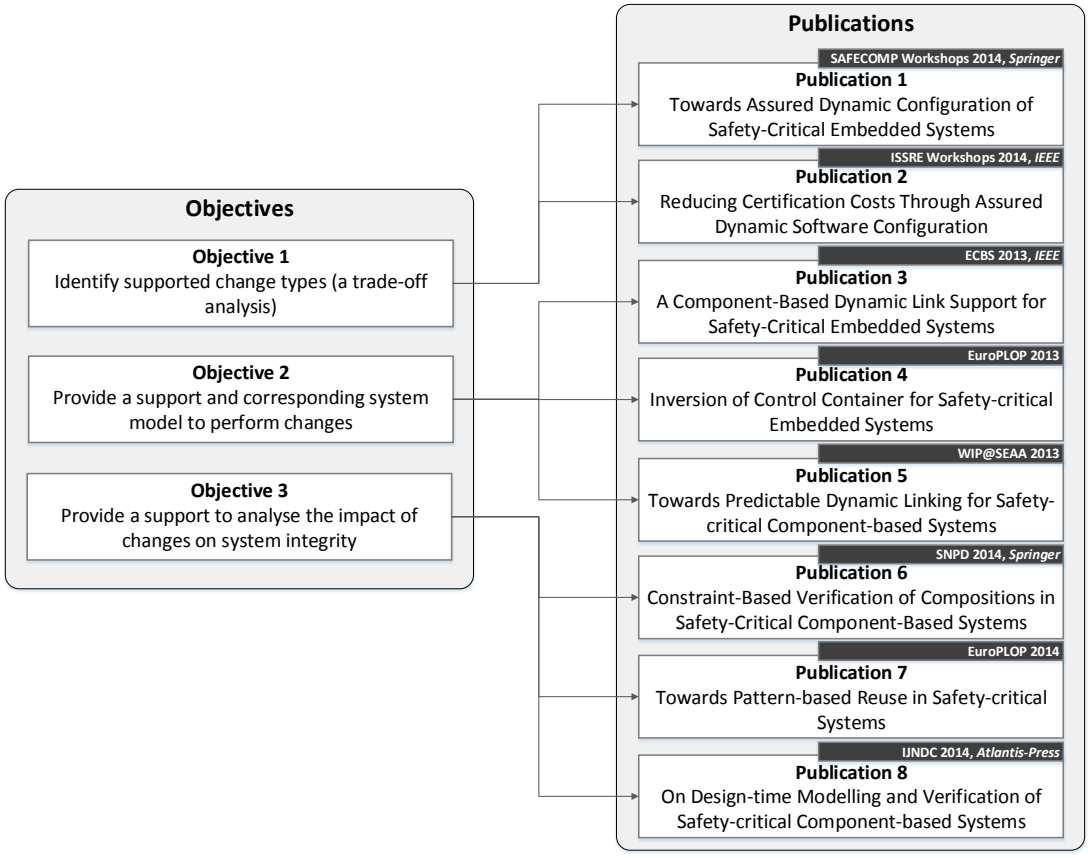


Figure 6.1: Overview of the publications and their mapping to thesis objectives

Towards Assured Dynamic Configuration of Safety-Critical Embedded Systems

Nermin Kajtazovic, Christopher Preschern,
Andrea Höller, and Christian Kreiner

Institute for Technical Informatics,
Graz University of Technology,
Infeldgasse 16, Graz, Austria
{nermin.kajtazovic,christopher.preschern,
andrea.hoeller,christian.kreiner}@tugraz.at

Abstract. Assuring systems quality is an inherent part of developing safety-critical embedded systems. Currently, continuous increase of systems complexity, in particular that of software, makes this development challenging. In response, more and more software faults are remaining unidentified at design-time so that changes and maintenance need to be performed at an increased rate. Unfortunately, today's safety-critical systems are not designed to be upgraded or maintained in a seamless way, so that the overhead of performing changes may be considerable, especially when such changes require to re-verify and re-validate the whole system.

In this paper, we present an approach to perform software changes in the operation and maintenance phase of the systems lifecycle. Changes are performed dynamically, by replacing parts of software (i.e., software components) with their functionally equal out-of-the-box instances. In order to prevent the impact of changes on systems integrity, we provide a support to model and to analyze the system. The main outcome here is that specific kind of changes can be maintained without adding any development costs.

Keywords: safety-critical embedded systems, component-based systems, dynamic configuration.

1 Introduction

Maintaining a correct function even in presence of faults is an important characteristic of safety-critical embedded systems. In order to reduce the risk of failures, and thus to avoid the potential environmental damages or harm on humans, their hardware/software development has to be rigorous and quality assured.

Currently, rapid and continuous increase of systems complexity, in particular that of software, makes the development of these systems challenging [4] [12]. In response, more and more software faults are remaining unidentified at design-time so that changes and maintenance need to be performed at an increased rate. Concrete examples of such change and maintenance demands are quite often

A. Bondavalli et al. (Eds.): SAFECOMP 2014 Workshops, LNCS 8696, pp. 167–179, 2014.

recalls of vehicles, medical devices, and other products. Some of these recalls are related to faults located in the software functions, such as the control algorithms, libraries, flaws in modification or adaptation, and other. According to recent studies related to defect analysis in recalls, those faults are getting more frequent, as more and more functions are being implemented in software [2]. Eliminating those faults in most current safety-critical systems is quite difficult, in particular because it has to be evidenced that the changed system still maintains certain level of quality – a so called safety integrity in the notation of safety standards. To provide such an evidence, many steps in the development lifecycle have to be repeated. In addition, depending on the impact of changes and regulations of the considered safety standard, new certification might be required.

In this paper, we present an approach to perform software changes in the operation and maintenance phase of the systems lifecycle. Changes are performed dynamically, by replacing parts of software (software components [5]) with their functionally equal out-of-the-box instances. Before any change can be performed, a new system configuration is analyzed against the violation of the safety integrity. Thus, only the configurations that pass this analysis step can be installed into the system dynamically. To enable such assured dynamic configurations, we have provided the following basis in our previous work: (a) a runtime mechanism that allows to load the out-of-the-box software components into a real-time operating system dynamically – the dynamic linker [10], and (b) a design-time mechanism to ensure the consistency of new system configurations [11]. This consistency mechanism performs the analysis of a changed system based on modelled properties which describe certain system attributes, such as memory and timing budgets for example[1]. In order to determine whether changes caused by replacing software components have an impact on the safety integrity, there is a need to identify which attributes may be relevant here. For this purpose, we analyze in this paper how the change management is regulated in some safety standards, and under which conditions the replacements of components are allowed.

The main outcome here is that for specific kind of changes, in which software components can be replaced, the system does not need to be turned back into the development phase. Furthermore, if the re-certification of the system is required, the original certification data can be reused, since they are not impacted by those changes. In response, replacements of software components can be maintained without any development costs.

The remainder of this paper is organized as follows: Section 2 provides a brief overview of relevant related work. Section 3 describes how changes are handled in safety standards, and which system attributes have to be considered when analyzing changes. In Section 4 the proposed approach is described, and a short discussion is given in Section 5. Finally, concluding remarks are given in Section 6.

[1] We use the notation *system attributes* to identify various functional and non-functional system aspects, such as performance requirements, constraints, etc.

2 Related Work

Now we turn to a brief overview of related studies. We summarize here some relevant articles that handle the analysis of changes in safety-critical embedded systems.

To date, much research has been done on analyzing planned changes in software architectures for safety-critical systems [1] [15] [13] . In the work by Adler et al. [1], an adaptive architecture for safety-critical automotive systems is proposed. The main goal here is to increase the systems availability by allowing software components to implement diverse behaviours, so that in the event of failures or degradation of quality, the automotive system can continue operating by switching between correct implementations. Since different implementations of components may have different quality, the authors provide a design-time analysis to prevent mixing not allowed combinations of component implementations. For this purpose, they define a quality system, with a set of fixed quality types. A more advanced framework for dynamic adaptation of avionics systems was developed by Montano [15]. The goal is to adapt the system to new, correct configurations, in case of failures. To perform this, a common quality system defines the contracts between functions and available static resources (e.g. memory consumption, CPU utilization, etc.) and in this way it restricts the possible set of correct configurations. An important aspect of this work is that it demonstrates the CP approach to solving the composition problem. However, the quality type system only considers static resources, and does not consider contracts between functions. Ultimately, the approach is strongly focused on dynamic adaptation with human-assisted decision making. Similar reconfiguration strategy is used in [13], but the consistency of the reconfiguration here is ensured by the runtime mechanisms (partitioning).

There are also some works which focus on upgrading safety-critical systems [20] [16] [19]. One of the most notable is work done in the scope of the project PINCETTE, which has as a goal to perform live upgrades of software systems that control the safety-critical processes [20]. Although the topic is beyond the scope of available validation methods in the practice, the aim is to evaluate the feasibility of formal methods to such use cases. In contrast to our data flow-oriented analysis, the focus here is on validating the interaction between upgraded behaviours. Another work [16], done in the scope of the RECOMP project, addresses also live upgrades as one of the goals to reduce the costs for certifying systems. However, only dynamic linker has been realized here, without considering the analysis of changes. Finally, the work in [19] shows how to validate changes of upgraded safety-critical system. Here, model checker is used to verify changed behaviour.

In summary, various analysis methods have been developed to validate changes. However, none of the approaches discussed here consider regulations of standards, to identify whether changes they support are allowed and, if so, to which extent.

3 Addressing Changes in Engineering of Safety-Critical Embedded Systems

Identifying system requirements affected by changes is a crucial step in the change management process. To determine which requirements and which related system attributes influence the systems safety integrity, we analyze in the following how changes are regulated in safety standards. Based on this analysis, we build a list of system attributes that we further use to construct our software architecture, and to build properties for our software components.

3.1 Change Management in Safety Standards

In general, standards for functional safety provide the guidelines on how to align the system development with the safety lifecycle in each phase. One aspect of these guidelines are activities related to maintenance and operation phase of the systems lifecycle. Changes in the operation phase are usually handled in the context of the supporting processes defined in standards, such as the maintenance, the configuration management, and the change management [18]. In the following, we describe the change management defined in the IEC 61508, which is a generic safety standard applied in the industry. We align our approach to this standard, because many guidelines it provides can also be found in other standards applied in specific industrial sectors, since they represent derivatives of the IEC 61508 (e.g., the ISO26262 standard provides similar guidelines for maintaining changes in automotive systems).

The lifecycle of the IEC 61508 standard comprises the engineering activities for software and systems scope. Changes in the operation and maintenance phase of systems are described in parts 1, 2 and 3 of the standard, in the context of the supporting processes: maintenance, configuration management, and change management. Each of these processes has defined steps, the inputs and the work products it shall produce. To ensure the safety integrity after implementing changes, the standard prescribes requirements that have to be fulfilled and a list of possible techniques and measures to apply within these processes. The requirements are mainly related to activities that need to be performed if safety integrity is affected by changes. In Table 1, we have filtered out the most relevant requirements. Basically, if safety integrity is affected by changes the standard recommends to (i) perform the hazard and risk analysis in order to identify additional faults that might be introduced by such changes and (ii) to return to the appropriate phase in the software lifecycle to implement changes. On the system level (part IEC61508-2), it is recommended to use the same development equipment and expertise (e.g., tools, previous system configuration, project artifacts, etc.), in order to just focus on changed parts only. In addition to requirements, developers have the option to choice which techniques and measures to perform, based on the level of safety integrity they want to achieve after implementing changes (bottom part of the table). Among them, the most influential measure here from the aspect of costs is a need for the verification and validation. For the highest levels of safety integrity, the standard recommends to

Table 1. IEC 61508 requirements, measures and techniques related to change management (an excerpt)

Requirements on software change management, IEC 61508-3	
7.8.2.3	An analysis shall be carried out on the impact of the proposed software modification on the functional safety of the E/E/PE safety-related system: a) to determine whether or not a hazard and risk analysis is required; b) to determine which software safety lifecycle phases will need to be repeated.
7.8.2.5	All modifications which have an impact on the functional safety of the E/E/PE safety-related system shall initiate a return to an appropriate phase of the software safety lifecycle. All subsequent phases shall then be carried out in accordance with the procedures specified for the specific phases in accordance with the requirements in this standard. Safety planning (see Clause 6) shall detail all subsequent activities.
Requirements on system change management, IEC 61508-2	
7.8.2.3	Modifications shall be performed with at least the same level of expertise, automated tools (see 7.4.4.2 of IEC 61508-3), and planning and management as the initial development of the E/E/PE safety-related systems.
7.8.2.4	After modification, the E/E/PE safety-related systems shall be reverified and revalidated.
Recommended techniques and measures, IEC 61508-3 A.8	
2	Reverify changed software module
3	Reverify affected software modules
4a/4b	Revalidate complete system or Regression validation

perform the re-verification and re-validation of the complete system (measures 2, 3, 4a in the Table 1). Alternatively, regression validation would also suffice (measure 4b). Nevertheless, changed artifacts (from the work products of the hazard and risk analysis down to the test reports) have to be newly certified.

In summary, the change impact on safety integrity implies to update many work products throughout the systems lifecycle, to repeat particular steps of that lifecycle and to re-verify and re-validate the system. However, according to requirements 7.8.2.3 and 7.8.2.5, those activities have to be performed only if there is an impact on the functional safety (i.e., the systems safety integrity is changed). Our goal in this context is to allow changes to an extent to which they have no impact on the systems safety integrity. For this purpose, we need to evaluate the requirement 7.8.2.3-a, for every change request. If there is no need for the hazard and risk analysis, changes are allowed, otherwise not. To realize this, we first need to identify the system attributes that have an impact on systems safety integrity. Based on these attributes, we can set constraints on the architectural level (e.g., software components, layers, operating system configuration, etc.) that would allow us to evaluate the requirement 7.8.2.3-a. In the following, we introduce these attributes.

3.2 Impact of Changes on System Requirements

Safety standards set requirements to achieve the functional safety, while leaving the space for the developers on details on how they should implement those requirements. The same holds for the change management, i.e., the IEC 61508 does not specify which system attributes have to be considered when analysing the impact of changes. More concrete guidelines about this can be found in the avionics domain, concretely in the concept Reusable Software Component (RSC) from the Federal Aviation Administration (FAA) that was developed for the standard

DO-178B, to enable reuse of software components and their late integration into a certified safety-critical system [7]. Similar to change management, the aim is to maintain the functional safety after integrating components. Although RSC provides concrete information about reusing pre-fabricated components, no focus has been given on how to design such components for reuse – for example, how to describe the context in which components have to operate (embedded system, environment, etc.) and which system attributes contribute to that context. Similar to RSC, the concept Safety Element out of Context (SEooC) as part of the automotive standard ISO26262 defines reuse for the sub-systems, but on the abstraction level of requirements.

To our knowledge, the only available official publication that handles change management in detail and is related to safety standards are the FAA guidelines on analyzing the impact of changes in software [6] [17]. Here, a collection of the concrete system attributes that might be affected by changes is presented. This collection is made to help developers in the post-certification process of the DO-178B standard to ensure the safety integrity of the changed system by determining the impact of changes on the system, and by estimating the overhead to re-verify, re-validate and re-certify the system. Although avionics domain is addressed here, most of those attributes are common to embedded systems in general. In Table 2, we summarize the common system attributes.

Table 2. Considered system attributes to analyze impact of changes, according to Federal Administration Aviation (FAA) [6]

System attribute	Description
traceability	requirements, design, tests, procedures
memory margin	memory allocation requirements (volatile, non-volatile memory)
timing margin	timing requirements (task scheduling, interface timing, ...)
data flow	coupling between software components (data syntax, semantics)
control flow	coupling between software components (events, calls, ...)
input/output	interfaces with the external world (bus, hardware, memory, ...)
development environment and process	compilers, linkers, loaders, tools
operational characteristics	runtime mechanisms (changes on limits, i.e. contracts, exception handling, ...)
partitioning	change on protective safety mechanisms

We use some of the FAA attributes as the first class entities to maintain the consistency of the system, and to estimate the impact of changes. We discuss the selection of attributes in the following section more in detail.

4 Ensuring Consistency of System Configurations

In this section, we introduce our approach to ensuring the consistency of system configurations. To this end, we show how we define a system using attributes

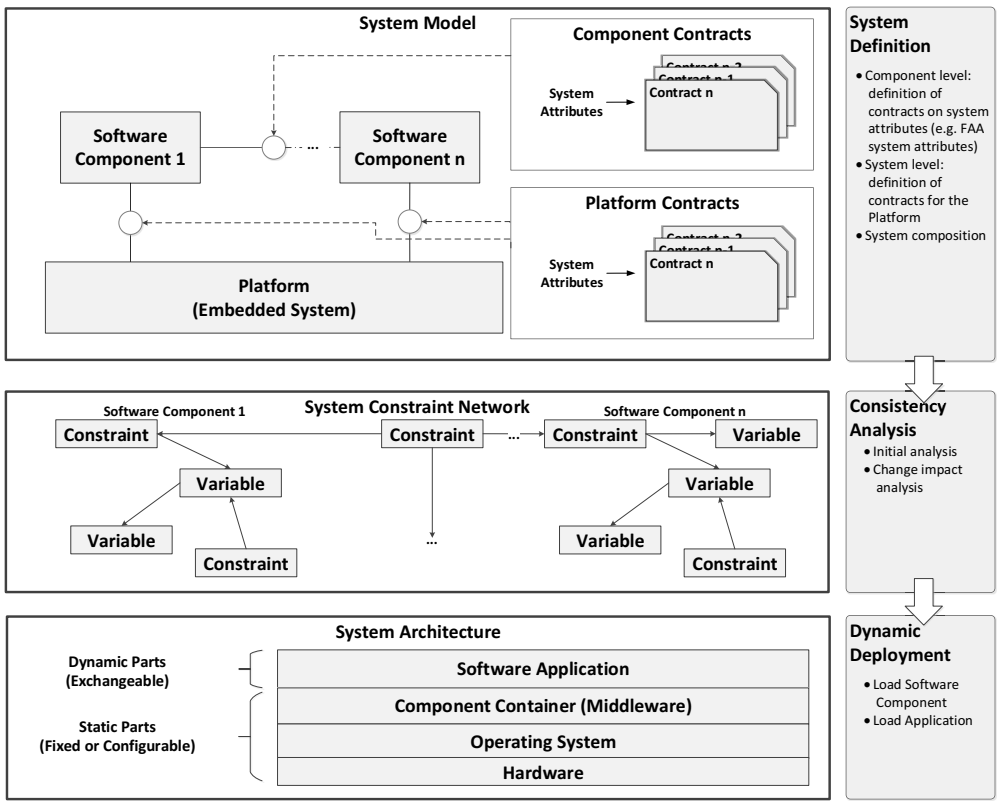


Fig. 1. Proposed workflow for ensuring systems consistency: system modelling using contracts to describe attributes (top), consistency analysis (middle) and dynamic deployment of software components (bottom)

described in the previous section, and how we analyze the impact of changes. All information about systems consistency is contained in those attributes.

The proposed approach in the workflow form is depicted in Figure 1. On the top, a model of the system is defined. This model consists of the two elements: software components which implement certain application-level functions, and the platform, which is a model of an embedded system. Both software components and the platform implement certain contracts, in order to express relations to other dependent components or platform. These contracts are the fundamental elements of the system model that allow us to maintain the consistency of the system. They contain the information about system attributes discussed in the previous section, and provide means to build relationships to other contracts. Based on those relationships, impact of changes in one particular contract can be tracked throughout the complete system. We introduce contracts later in Section 4.2. In the next step of the workflow, the system in terms of contracts is translated into a so called constraint network, i.e., a set of inter-connected variables and constraints. This constraint network represent contracts and their relationships in another problem domain, which allows us to automatically analyze the consistency of the system by evaluating constraints.

In the last step of the workflow, components can be dynamically loaded into the platform, depending on results of the analysis. If all constraints in the analysis step are satisfied, the system modelled in the first step is consistent, i.e., we say the system configuration is assured. Thus, any change in the modelling step can be captured and analyzed in the constraint network.

In the following, we describe parts of this workflow more in detail.

4.1 Software Architecture

To perform changes in the system by replacing some parts of it, there is a need for an adequate architectural support, i.e. a design for upgrades [9]. Another important aspect here is that a degree of flexibility shall be balanced to an extent to which an the impact of changes on system attributes listed in Table 2 can be managed. For example, if changes in behaviour of certain software functions cannot be analyzed (e.g., in the constraint network from Figure 1), replacing those functions with different behaviours shall not be approved. Therefore, certain limitations are necessary to set on the design.

For our system, we use a Component-based Software Engineering (CBSE) [5], which is currently a key paradigm applied for building safety-critical systems. Automotive AUTOSAR, standards such as IEC61131/499 and IEC61850, are some of the reference component-based architectures. In those architectures, software components implement parts of systems functions, such as the controllers, software sensor and actuators. Due to well-defined interfaces, component may implement functions on different granularity levels, e.g., like Matlab Simulink function blocks and sub-systems, thus allowing for compositional (hierarchical) design. Moreover, well-defined interfaces allow their reuse, customization for the use in different contexts, and so forth.

Our software architecture is depicted in Figure 1 (bottom part). Here, software components implement certain software functions composed into an application, whereas their lifecycle, their coordination and resources from the operating system are managed by the underlying middleware, i.e., a component container. Changes related to software may impact any of these layers, and therefore any of the introduced system attributes. In order to be able to analyze such an impact, we set limitation on the design so that replacements of software components are allowed only. That means, some of the system attributes are fixed at design-time so that changes have no influence on those attributes. For example, connections between software components have to be static, since they may affect the functional requirements if changed (e.g., adding/removing software components, or changing connectors may affect systems behaviour), and this can only be analyzed manually.

With our limitations, the impact of changes is related to software components and their interaction with the platform only. However, such cases are also not trivial, since changes may still have an impact on systems consistency. For example, the consistency may be compromised if the replaced software component implements interfaces with different semantics, e.g. different value intervals provided to dependent components, and new intervals were not considered during

the system verification. Similarly, mixing components with different quality levels may cause the same effects, e.g. deploying components qualified for the lower level of the safety integrity than the integrity of the platform.

The main impact of changes here is on (i) resource management, in particular on task and memory management for components, and on (ii) interfaces between components and their interfaces to the platform. From the perspective of the software architecture, the configuration of tasks (i.e., number of tasks, their scheduling policy, etc.) and the organisation of memory (i.e., memory layout, size of the heap, and allocation to tasks) are static features. However, they have to be included in the analysis since exchanged components may have different demands with regard to resources (timing, memory).

Similarly, the connections and interfaces between components are static, but many details have to be considered in order to ensure that the integration or the composition is correct (i.e., the syntax and the semantics, such as units, valid intervals of certain data values, etc.). In addition, some components may implement many alternative behaviours so that different configurations of interfaces are possible. Therefore, we consider interfaces in our analysis. Finally, details with regard to the development process and the operational profile are also parts of the analysis. In Table 3, we summarize system attributes that we include in the analysis, and possible types of changes (right column). The systems consistency is therefore analysed based on these changes only. The remaining systems attributes from Table 2 are fixed at design time, and cannot be influenced, i.e., the control flow of components, their interaction semantics and behaviour are implemented as a static part of the architecture.

Table 3. System attributes considered in the consistency analysis, selected from FAA attribute collection [6]

System attributes	Allowed changes
memory margin	components: volatile and non-volatile memory platform: volatile memory (allocated to tasks), non-volatile memory
timing margin	components: execution time platform: task execution time
data flow	components/platform: syntax (datatype, interface), semantics (intervals, values, specific constraints (units, standard compliance, configuration and calibration data, ...))
development environment and process	components/platform: tools (compiler, linker, specific constraints (build options, ...)), version
execution platform	components/platform: architecture (cpu, floating point support, ...), safety integrity level

4.2 System Modelling

To integrate the information about selected attributes from Table 3 into the systems structure, we use Contract-based Design paradigm (CBD) [3]. According to CBD, software components and the platform implement certain contracts, which capture part of that information (i.e., a quality stamps or properties). In addition to capturing information, contracts provide means to integrate components and platform.

Among few types of different contract available, such as state transition-based contracts like interface automata, probabilistic contracts, etc., we use a form that is based on data semantics, i.e., data flow contracts [3]. According to this type, every contract consists of data parameters, and expressions (or properties) on those data parameters in form of assumption and guarantees. The guarantees describe a valid behaviour in form of expressions, that can evaluate to true/-false, depending on the evaluation of expressions in assumptions. For specifying contracts, various formalisations can be used, for example logic languages such as propositional logic, first order logic, their extensions, and other.

In Figure 2, we show how the structure of our software components is defined to match with used contracts, and how different types of system attributes are modelled using contracts. A trivial example is shown here just to simplify the demonstration. Similar to the structure of contracts, every software component and the platform are defined as a set of data parameters, input and output data variables. In addition to this information, they contain a list of implemented contracts. Thus, data parameters in contracts relate to data parameters of their corresponding components/platform.

Another essential aspect of CBD are the relationships between contracts, which allow to verify the composition between two contracts, if their assumptions and guarantees are defined in a formal way. In our example, the contracts of components M_{IS} and M_{IIAS} are related with each other using a composition relation, which is valid only if these contracts are compatible and can interact. Concretely, this means the relation is valid if the contract of a component which accepts data, the $C1$ implemented by component M_{IIAS}, can be satisfied by

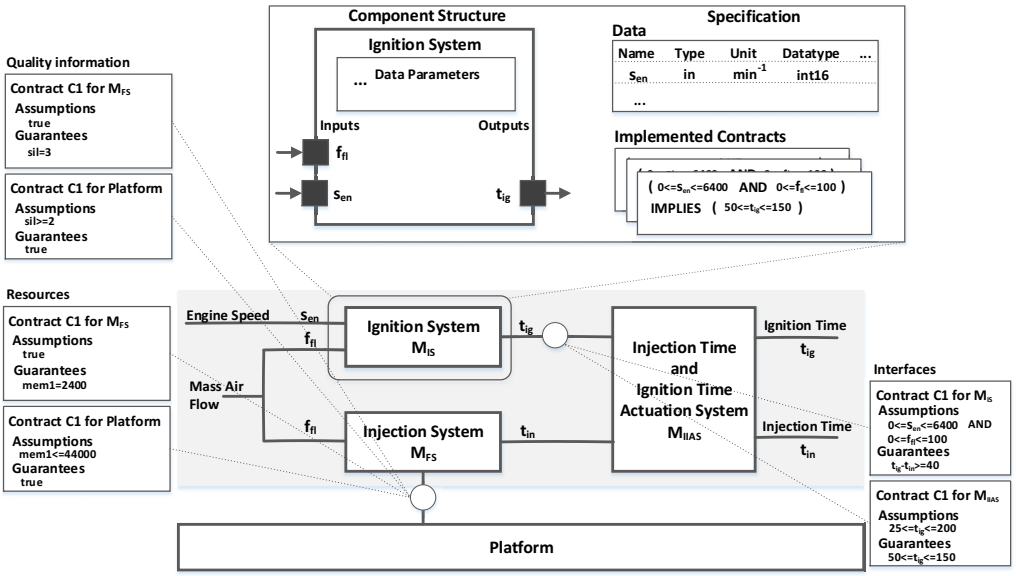


Fig. 2. Structure definition of software components and the platform, and supported types of contracts shown on an example of the engine controller, adopted from [8]

the guarantees of the providing component. In the example, this can evaluate to true only if the assumption ($25 \leq t_{ig} \leq 200$) can also evaluate to true[2]. This can only be satisfied, if the guarantee of the contract $C1$ of M_{IS}, ($50 \leq t_{ig} \leq 150$), matches with the assumption ($25 \leq t_{ig} \leq 200$), which is the case in the example, since M_{IIAS} can accept more values of t_{ig} (for more details, please refer to [11]). Therefore, guarantees and assumptions of dependent components are interrelated by their expressions. In a similar way, we define contracts for resources and quality information, as shown in the figure.

Based on relationships between contracts, information about the system attributes on a system level is maintained. Changes in any contract (or exchanges of contracts) can be captured by evaluating assumptions/guarantees of other dependent contracts.

4.3 Consistency Analysis

The consistency analysis is based on verifying relations between contracts, in particular, by evaluating their assumptions and guarantees. As a background technology, we use Constraint Programming paradigm (CP) [14], which is a widely applied method to solve decision and optimization problems. The essential aspect of CP is a problem definition, which is represented as a network of variables of various types and constraints. Here, constraints represent various kinds expressions on variables (logical, arithmenic, etc.), and can be related to other constraints. Solving that problem means evaluating all constraints in the network. Thus, there is a solution if all constraints in the network are satisfied.

In our approach, we translate the system modelled in form of contracts into such a constraint network. For this purpose, we have defined a model of a contract, its variables, assumptions and guarantees, and relations between contracts as network elements, i.e., variables and constraints. The systems consistency is therefore analyzed by evaluating constraints that are derived from contracts (for more details, please refer to [11]).

5 Discussion

We showed in this paper that simple replacements of software components are not trivial. Many details about functional and non-functional aspects of software components have to be considered to ensure that replacements have no impact on systems integrity. One of the major challenges here is to determine how much information should be considered in the analysis, to have a confidence in its results. With the list of systems attributes we introduced in this work, some fundamental aspects are covered, but much more details might be required, depending on the specific domain. This collection of attributes can be extended according to types of contracts introduced.

[2] This data is related to the ignition time of an engine controller t_{ig}. The components modelled here implement contracts in order to satisfy the timing requirement on allowed difference between injection and ignition time, i.e. ($t_{ig} - t_{in}$).

The analysis method presented here can also be applied to some existing component-based systems, but in some cases with certain limitations. In AUTOSAR for example, changes would be possible on a level of Runnables, which are units of the execution inside of AUTOSAR software components, and have a generalized standard behaviour (read, execute, write) [12]. In contrast, changes of complete software components could not be supported, because events for the execution of Runnables are user-defined, and other techniques are required here to analyze the interaction between those events. Generally, for synchronous data flow systems, such as IEC61131-based systems, or Matlab Simulink function blocks, it is more easily to apply the analysis, since software components used here have a standard behaviour and standard execution semantics.

6 Conclusion

In this paper, we presented an approach to perform changes on safety-critical embedded systems in the operation and maintenance phase. Changes are limited to replacements of software components. To prevent the impact of such type of changes on systems integrity, we have analyzed which related system attributes might be affected when replacing software components. Based on those attributes, we provided a modelling means to build a system including attributes on the level of software components and their platform (embedded system), and we provided a consistency analysis of such a modelled system. The main outcome of this work is that for replacements of software components the system does not need to be turned back into the development phase.

The collection of attributes described here provides a foundation for the future work. One of the major challenges here is to determine how much information is required to describe software components and their platform, in order to have a confidence on results of the consistency analysis. This may depend on many factors, such as used software architecture, domain-specific details, and so forth.

As part of our ongoing work, we will analyse different component-based architectures with regard to the use case of replacing software components, and derive specific system attribute out of them. The aim is to extend the proposed modelling and analysis support to system attributes, which can be commonly used in safety domains.

References

1. Adler, R., Schaefer, I., Trapp, M., Poetzsch-Heffter, A.: Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. ACM Trans. Embed. Comput. Syst. 10(2), 20:1–20:39 (2011)
2. Alemzadeh, H., Iyer, R., Kalbarczyk, Z., Raman, J.: Analysis of safety-critical computer failures in medical devices. IEEE Security Privacy 11(4), 14–26 (2013)
3. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.: Contracts for Systems Design. Tech. rep., Research Report, Nr. 8147, 2012, Inria (2012)

4. Butz, H.: Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. Department of Avionic Systems at Airbus Deutschland GmbH Kreetslag 10, D-21129 Hamburg, Germany (-)
5. Crnkovic, I.: Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood (2002)
6. FAA: Guidelines for the Oversight of Software Change Impact Analyses used to Classify Software Changes as Major or Minor. Notice 8110.85, FAA (2000)
7. FAA: AC20-148 Reusable Software Components. Tr, FAA (2004)
8. Frey, P.: Case Study: Engine Control Application. Tech. rep., Ulmer Informatik-Berichte, Nr. 2010-03 (2010)
9. Gluch, D., Weinstock, C.: Workshop on the State of the Practice in Dependably Upgrading Critical Systems: April 16-17, 1997. Special report, Carnegie Mellon University, Software Engineering Institute (1997)
10. Kajtazovic, N., Preschern, C., Kreiner, C.: A component-based dynamic link support for safety-critical embedded systems. In: 20th IEEE ECBS (2013)
11. Kajtazovic, N., Preschern, A., Hoeller, C., Kreiner, C.: Constraint-based verification of compositions in safety-critical component-based systems. In: IEEE/ACIS SNDP (Juni 2014)
12. Kindel, O., Friedrich, M.: Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis. dpunkt Verlag, Auflage: 1 (2009)
13. Lopez-Jaquero, V., Montero, F., Navarro, E., Esparcia, A., Catal'n, J.: Supporting arinc 653-based dynamic reconfiguration. In: 2012 Joint WICSA and ECSA (2012)
14. Marriott, K., Stuckey, P.J.: Programming with Constraints: An Introduction. The MIT Press (March 1998)
15. Montano, G.: Dynamic reconfiguration of safety-critical systems: Automation and human involvement. PhD Thesis (2011)
16. Pop, P., Tsiopoulos, L., Voss, S., Slotosch, O., Ficek, C., Nyman, U., Ruiz, A.: Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the recomp approach. In: WICERT (2013)
17. Rierson, L.: A systematic process for changing safety-critical software. In: Proceedings of the 19th Digital Avionics Systems Conference, DASC, vol. 1, pp. 1B1/1–1B1/7 (2000)
18. Smith, D., Simpson, K.: A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849. Elsevier Science (2010)
19. Soliman, D., Thramboulidis, K., Frey, G.: A methodology to upgrade legacy industrial systems to meet safety regulations. In: 2011 3rd International Workshop on Dependable Control of Discrete Systems (DCDS), pp. 141–147 (June 2011)
20. Zhang, M., Ogata, K., Futatsugi, K.: Formalization and verification of behavioral correctness of dynamic software updates. Electronic Notes in Theoretical Computer Science 294, 12–23 (2013); Proceedings of the 2013 VSSE Workshop

2014 IEEE International Symposium on Software Reliability Engineering Workshops

Reducing Certification Costs Through Assured Dynamic Software Configuration

Nermin Kajtazovic, Andrea Höller, Tobias Rauter, and Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
{nermin.kajtazovic, andrea.hoeller, tobias.rauter, christian.kreiner}@tugraz.at

***Abstract*—Engineering activities in the operation and maintenance phase of safety-critical systems are becoming increasingly important. The ever more rising software complexity in terms of an amount of implemented functions led to a proportional increase of various change demands. Most of these demands are initiated to repair the system from defects, i.e., due to design faults not identified in the development for example. Maintaining changes in the operation phase can be very cost-intensive, because regulations of safety standards require to re-verify and re-validate the system in most cases, in order to ensure that the systems integrity is not compromised by the incorporated changes.**

In this paper, we describe an approach to perform changes on software in the operation and maintenance phase of systems lifecycle. To prevent the impact of changes on systems integrity, certain design limitations are set, so that controlled types of changes are permitted only. Furthermore, since also in cases of strong design limitations the systems integrity can be compromised, a support for systems modelling and analysis has been provided. The modelling captures certain functional and non-functional aspects of the system, which are then analyzed to decide whether changes can be performed or not. The main outcome here is that specific types of changes can be maintained without having an impact on systems integrity and therefore without requiring an extensive re-verification and re-validation. We report on possible improvements in costs of changes, by considering several industrial use cases and their typical change scenarios in the maintenance phase.

***Keywords*—dynamic configuration; component-based systems; safety-critical systems**

I. Introduction

The complexity of today's safety-critical systems in terms of provided software functions, their distribution, and reuse, represents one of the major issues in safety engineering [1][2]. To cope with the market demands on cost reduction, flexibility, and performance, the industry has made a significant technological progress in the last decades by adopting new paradigms and methods in their engineering landscapes [1]. Currently, the synergies between modular architectures, (international) standards, and model-based engineering are the main supporting drivers in developing safety-critical systems.

On the other side, though significant improvements in systems engineering, managing software complexity still remains a very tedious task. According to some recent studies that deal with the analysis of defects in the operation phase of safety-critical systems, software faults introduced in the development are one of the main causes for computer-related recalls of products[1]. Taking in account that annual increase of software complexity for embedded systems in general lies between 10 and 30 percent (depending on domain [1]), it is expected that maintenance activities will need to be performed at an increased rate, proportionally. However, managing changes to repair the introduced defects is a challenge. Recommendations on change management defined in safety standards, in particular in a generic industrial standard IEC 61508, mainly require to perform a complete system re-verification and re-validation, in order to ensure that the systems integrity cannot be compromised due to introduced changes [5]. In the worst-case, i.e., when the system integrity is compromised and has to be re-established, a new certification has to be conducted. For many change scenarios this can be a very cost-intensive activity, especially for minor changes which usually require only to exchange control algorithms, software components, libraries or to calibrate and configure certain system parameters for example. Providing a systematic way on handling such changes would bring benefits in reduction of costs in the change management process. However, an additional problem is that safety-critical systems in general are not designed to support changes, i.e., new software configurations can only be built and installed at design-time. This also requires to rebuild the complete software system even for minor changes and to re-map it onto the target embedded platform.

In our previous work, we proposed an approach to perform controlled changes on software in the operation and maintenance phase of systems lifecycle [6]. Changes are related to dynamic replacements of software components, and can address defect repairs on a level of those components (e.g., exchange of control algorithms, certain libraries). The essential part here is that every new software configuration is assured, i.e., verified against compromising systems integrity. Another important contribution is that assured software configurations can be dynamically deployed onto an embedded system.

In this paper, we analyze possible improvements in certification costs when applying the approach. We compare the approach with some industrial use cases by taking into account typical change scenarios collected from defect analysis.

Section II summarizes the main contributions of our assured dynamic software configuration. In Section III, possible improvements by applying the approach are outlined. A brief overview of relevant related work is given in Section IV. Finally, concluding remarks are given in Section V .

[1]Studies are related to medical products and automotive [3][4]

978-1-4799-7377-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ISSREW.2014.90

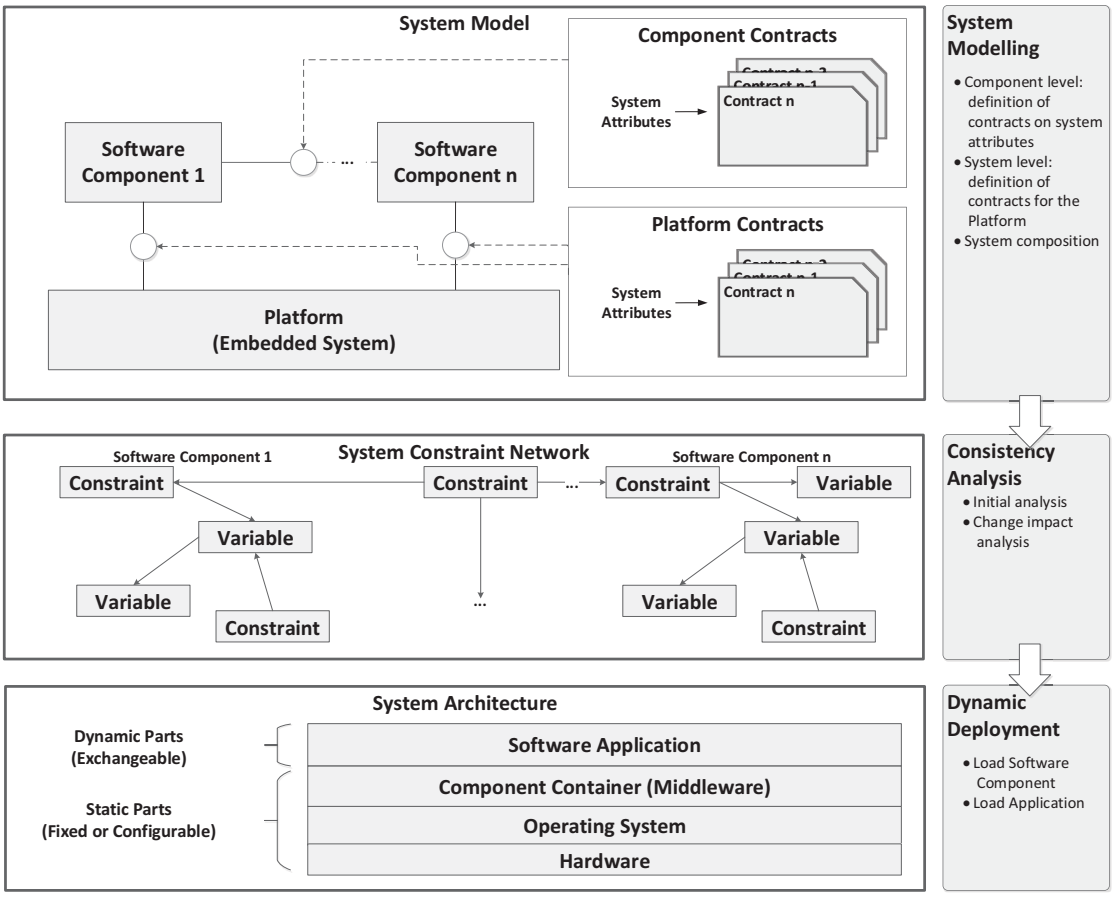


Fig. 1. Workflow for managing changes: system modelling using contracts to describe attributes (top), consistency analysis (middle) and dynamic deployment of software components (bottom)

II. Assured Dynamic Software Configuration

In this section, we introduce our approach to performing changes in the operation and maintenance phase of safety-critical systems. The following three distinct contributions can be identified here: (a) a **system modelling** – it allows to capture certain functional and non-functional system attributes such as memory and timing budgets for example[2], so that changes in a certain part of a system can be identified and their impact can be estimated (e.g., change in timing of the replaced component can influence the overall timing of a real-time task that executes that particular component), (b) **consistency analysis** – the systems integrity or consistency is automatically analysed based on modelled functional and non-functional system attributes, and (c) **dynamic deployment** – a runtime support is provided that allows to deploy software components into a Real-Time Operating System (RTOS) dynamically.

The introduced modeling, analysis and dynamic deployment are summarized in the workflow in Figure 1. First, the system attributes required to estimate the impact of the aforementioned changes are captured within a system model. This model consists of (i) software components, which realize some application-level functions, and (ii) the platform, which describes an embedded system (hardware). Both software components and the platform implement certain contracts, in order to structure the system attributes and to inter-relate those attributes thorough the complete systems hierarchy. To this end, contracts provide means to inter-relate those attributes by incorporating different types of relationships. In this way, the impact of changes in any of the contracts can be estimated, i.e., all affected, dependent contracts in the system hierarchy can be identified. We describe contracts in Section II-A in more detail. In the analysis step of the workflow, the component-based system is translated into a so called *constraint network* – a set of inter-connected variables and constraints. This network allows us to automatically analyze the consistency of the system by evaluating constraints (see Section II-B).

In the last step of the workflow, an architectural support is provided to perform replacements of software components dynamically. Those replacements are first modelled, and then analyzed against violating systems integrity (consistency). Thus, they are only permitted if they do not compromise systems integrity, i.e., in case when no contracts are violated.

The workflow described above can be attached to an existing change management process, for example the one defined in the IEC 61508 standard (in part 2 and 3). If defects identified in the operation are related to individual software components, this workflow can be used to manage the corresponding changes and to upgrade the system. In the following, we describe parts of this workflow more in detail.

[2]We use the notation *system attributes* to identify various functional and non-functional system aspects, such as performance req. and constraints [7]

A. System Modelling

For our system model illustrated in Figure 1 (bottom), we use a typical modular architecture that relies on a component-based paradigm [8], in order to distribute the functions of software applications to individual software components. The information about the functional and non-functional system attributes is therefore defined on a level of software components and the platform. To technologically enable this definition, we use a Contract-based Design paradigm (CBD) [9]. According to CBD, software components and the platform implement certain contracts, which capture part of that information (i.e., a quality stamps or properties). In Figure 2, we show the structure of used software components, and the structure of contracts including supported types of system attributes (i.e., quality, resource, and data-flow attributes). Here, a trivial example is shown just to simplify the demonstration (for more details, please refer to [10]). Every contract captures the information about system attributes in a form of so-called *assumptions* and *guarantees*. The later are the constructs which are very similar to requirements but on a component-level, whereas the assumptions describe a context that must be provided to components in order to satisfy such requirements. For example, a software component Ignition System M_{IS} from figure guarantees certain values on its outputs only if assumptions it defines on inputs are respected by the other, dependent components, i.e., the inputs s_{en} and f_{fl} in this case. Based on the structure of contracts, software components and the platform are defined as a set of data parameters, input and output data variables. In addition to this information, software components contain a collection of implemented contracts. Assumptions and guarantees of these contracts are therefore related to data variables of those software components.

Another important aspect of CBD are the relationships between contracts, which allow to make contracts dependent on each other within the complete system hierarchy. For example, the top-level requirement for the system in figure, which states that the time difference between the fuel injection t_{in} and ignition t_{ig} shall be above $40ms$ (for more details, please refer to [11]), depends on contracts of the containing components M_{IS}, M_{FS}, and M_{IIAS}. In this way, the information about the system attributes on a system level is maintained. Thus, changes in any of the contracts, or even exchanges of contracts, can be captured by evaluating relationships between contracts.

To have a confidence in results of the analysis in the next step, i.e., to be sure that all impacts of changes can be captured, the essential part of systems modelling is to first define the required attributes, based on used system model. In our previous work [6], we provided a collection of the most general system attributes that should be considered during the modelling. This collection is based on FAA (Federal Aviation Administration) guidelines for software change management in the avionics domain [12].

B. Consistency Analysis

In the analysis part of the workflow, the systems integrity (consistency) is verified by evaluating contracts and their relationships. As a supporting technology, we use Constraint Programming (CP) [13], which is a paradigm applied in many application fields to solve decision and optimization problems.

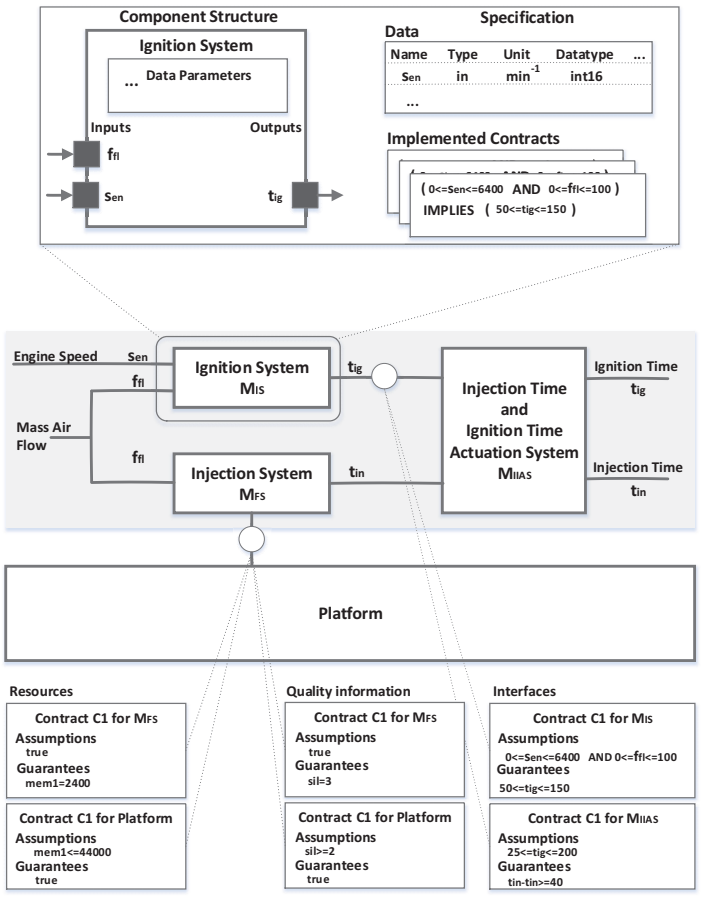


Fig. 2. Structure definition of software components and the platform, and supported types of contracts shown on an example of the engine controller, adopted from [10]

According to CP the system is represented as a network of variables and constraints – a problem definition in CP. Constraints are used here to put variables in expressions such as logical, and arithmetic expressions, and can be also related to other constraints. Solving CP problem means evaluating all constraints in the network.

In our approach, we translate the component-based system modelled using contracts into such a constraint network. To this end, we have defined a model of contracts, its variables, assumptions and guarantees, and relations between contracts as CP network elements, i.e., variables and constraints. The systems consistency is therefore analyzed by evaluating constraints derived from contracts (for more details, see [11]).

C. Dynamic Deployment

To deploy software components at load-time or runtime, we have realized a dynamic linker for Unix-like RTOSs [14]. It allows to load binaries of software components, in a similar way to shared library concept in Linux. Using this mechanism in context of operating systems for safety applications is a challenge, because the process of linking is quite complex and faults it may introduce may have considerable consequences at runtime. In our work [14], we report on several problems by applying the existing, standard dynamic linker to RTOSs for safety applications and how those problems can be solved.

Recalls data for Use Case 1	
Faulty system components	**# of recalls**
Software (control algorithm, flaws in creation, change)	82
Sensor (inadequate operation, change)	52
Actuator (inadequate operation, change)	30
External disturbance	26
Controller hardware (faults in hardware, change)	25
Other	100
Considered period	2002-2013

TABLE I. DISTRIBUTION OF RECALLS ACCORDING TO THEIR CAUSE IN COMPONENTS OF AN EMBEDDED SYSTEM - USE CASE 1

Recalls data for Use Case 2	
Faulty system components	**# of recalls**
Software (software, application, function, code, version, backup, database, program, bug, java, run, upgrade)	778
Hardware (board, chip, hardware, processor, memory, ...)	179
Battery (battery, power, power-up, discharge, charger, ...)	70
I/O (sensor, alarm, screen, interface, monitor, connect, wireless, ...)	41
Other	142
Considered period	2006-2011

TABLE II. DISTRIBUTION OF RECALLS ACCORDING TO THEIR CAUSE IN COMPONENTS OF AN EMBEDDED SYSTEM - USE CASE 2

III. EVALUATION

In this section, we analyze the possible reductions in costs when applying the approach to maintain changes. To this end, we first outline considered use cases and metrics used to conducts the evaluation. In the end, we present the results.

A. Use Cases

We use in this evaluation the reports on an analysis of recalls carried out in two studies, in particular in fields of medical engineering [3] and automotive [4]. Both studies provide a comprehensive analysis of repair demands, and provide a classification frameworks which show a distribution of such recalls according to several aspects, including systems components that caused systems failures. Tables I and II show an excerpt of the analysis focusing on this distribution, for use cases 1 and 2 respectively. The right column shows a number of recalls caused by failures of system components. For instance, 64% of the overall computer-related recalls in medical domain are caused by software failures.

B. Metrics

In order to estimate the overall costs of changes required to repair all components from Table I or II, the volume in terms of development costs and certification of every change (or a recall) has to be considered.

We use here a simple cost estimation metrics proposed by Bengtsson [15], the Architecture-Level Maintainability Analysis (ALMA), which is basically made to predict the maintenance costs or to quantitatively compare system architectures. The benefit of this method is that only a volume of changes of individual system components has to be provided – more precisely, their costs and probability of occurrence. According to ALMA, the predicted change costs C_{next} for the next maintenance period are defined as follows:

$$C_{next} := \left[\frac{P}{N} \cdot \sum_{j=1}^{N} (size_j \cdot weight_j) \right] \cdot M_{num} \qquad (1)$$

, where $size_j$ is an impact of a change scenario j (e.g., repair of a component), in terms of function points for example; $weight$ is a frequency of occurrence of a particular change scenario in a given period; N is a number of change scenarios; M_{num} is a number of maintenance tasks in a given period; P is a productivity constant (concrete values depend on a type of a change – modifying, adding, etc.).

Basically, this estimation method requires (i) to identify change scenarios and (ii) to estimate their volume in terms of size and weights, in order to predict the costs. Change scenarios correspond to actions to perform in the change management process, such as modifying and adding new components for example. The concrete definition of scenarios depends on a context. Similarly, the factors $size$ and P are also depending on the context. For example, for change scenarios on a source level, $size$ can be defined in terms of changes in lines of code [15]. To use parameters from Eq. 1, we set the following assumptions:

Assumption 1: A change scenario in our case corresponds to an action required to repair certain type of system components. Considering the distributions in Tables I and II, we have 6 scenarios for the use case 1 and 5 scenarios for the use case 2 respectively.

Assumption 2: The impact of a change scenario on development and certification costs is expressed as follows:

$$size := (c_{dev} + c_{cert}) \qquad (2)$$

We have to assume the values for c_{dev} and c_{cert} here, since statistics about cost per change scenario are not known (i.e., they are usually observed from the history of previous maintenance activities, or by interviews with experts [15]). Note that we use here normalized values for parameters c_{dev} and c_{cert} to express the impact of changes on development and certification costs respectively.

Assumption 3: With the introduced approach, the application-level functions such as control algorithms and certain libraries can be exchanged without having an impact on certification. The amount of this part of software has to be known. However, in the classification framework for the Use Case 1 there is no distinction between application functions, middleware, OS, etc., in contrast to User Case 2, in which 12% of software faults are located in applications. We assume therefore that automotive systems have the same distribution, i.e., that an amount of functions to repair is max. 12% of the overall software.

Finally, to estimate the reduction in costs, we use part of the Eq. 1 to just consider the costs required to repair components from Tables I and II in given periods, i.e.:

$$C_{eff} := \sum_{j=1}^{N-1} (size_j \cdot weight_j) + (size_f \cdot weight_f) \qquad (3)$$

Change scenarios and their volumes			
SNr.	**Change scenario (repair of)**	**size**	**weight**
1	Software	$c_{dev_1} + c_{cert_1}$	0.643
2	Sensor	$c_{dev_2} + c_{cert_2}$	0.147
...	...	...	...
Sum			1.0

TABLE III. APPLYING ALMA TO USE CASE 1 AND 2 - CHANGE SCENARIOS AND THEIR VOLUME IN TERMS OF SIZE AND WEIGHTS

The right part of the equation are costs required to repair the application software, whereas the left part corresponds to costs for repairing the remaining system components.

C. Results

The identified change scenarios and their corresponding weights are summarized in Table III. The occurrence of each of the scenarios is estimated from a number of related recalls for the time period from Tables I and II. Based on this data, we get the change costs C_{eff} for each use case, and a relative cost reduction when applying the introduced approach. The final results are summarized in Table IV.

The last column in the table shows the cost reduction relative to the C_{eff} of the corresponding use case. Generally, this reduction is related to repairs that have to be made on the application software. For such kind of repairs, there are no certification costs using the proposed approach, so that C_{eff} for both use cases is reduced by $(weight \cdot c_{cert_f})$. The overall reduction may therefore vary depending on the distribution of cost of individual system components, i.e., concretely, on a ratio between the c_{cert_f} for the application software and the $c_{cert_1} + ... + c_{cert_n}$ for other components. For example, in a case of an uniform distribution, in which all recalls require the same costs, the overall cost reduction for the use case 1 would be 7,7%. On the other hand, if the application-related recalls for that use case have no impact on certification, there would be no reduction at all. We summarize the possible cost reductions with Figure 3 (c_{dev} is not considered here). The horizontal axes represent normalized change impact on certification costs, in particular for software functions c_{cert_f}, and for remaining components $c_{cert_1} + ... + c_{cert_n}$. The peak values are marked with points $P1$ (uniform cost distribution, i.e., $c_{cert_f} = c_{cert_1} + ... + c_{cert_n}$), $P2$ (no reduction at all), and $P3$ (changing remaining system parts at no costs, which is unlikely in practice).

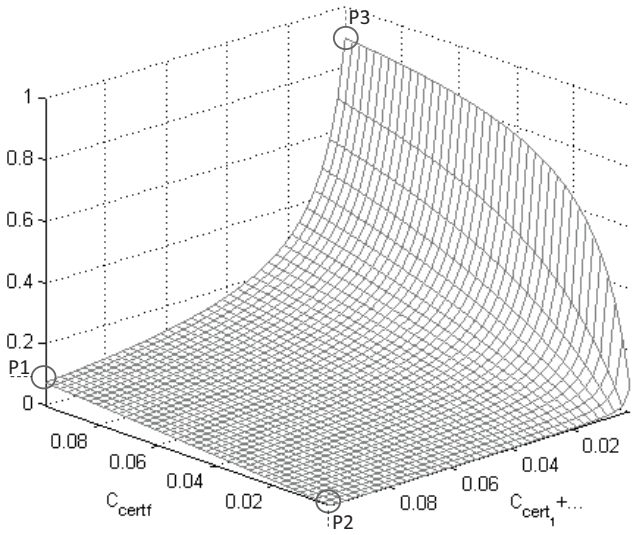


Fig. 3. Reduction in costs (relative to C_{eff} for the Use Case 2)

IV. RELATED WORK

Now we turn to a brief overview of related studies. We summarize here some relevant articles that handle the analysis of changes in safety-critical embedded systems[3].

To date, much research has been done on analyzing planned changes in software architectures for safety-critical systems [16] [17] [18]. In the work by Adler et al. [16], an adaptive architecture for safety-critical automotive systems is proposed. The main goal here is to increase the systems availability by allowing software components to implement diverse behaviours, so that in the event of failures or degradation of quality, the automotive system can continue operating by switching between correct implementations. Since different implementations of components may have different quality, the authors provide a design-time analysis to prevent mixing not allowed combinations of component implementations. For this purpose, they define a quality system, with a set of fixed quality types. A more advanced framework for dynamic adaptation of avionics systems was developed by Montano [17]. The goal is to adapt the system to new, correct configurations, in case of failures. To perform this, a common quality system defines the contracts between functions and available static resources (e.g. memory consumption, CPU utilization, etc.) and in this way it restricts the possible set of correct configurations. An important aspect of this work is that it demonstrates the CP approach to solving the composition problem. However, the quality type system only considers static resources, and does not consider contracts between functions. Ultimately, the approach is strongly focused on dynamic adaptation with human-assisted decision making. Recently, the approach proposed by Oertel et al. [18], [19] describes a comprehensive configuration management strategy for safety-critical systems. Similar to our work, it captures various non-functional system attributes to ensure the consistency of changes. On the other side, the approach is more general in terms of provided types of changes. More flexibility is given to make changes on a system, however, more effort is required for modelling to represent many types of requirements. Furthermore, the approach basically performs change management offline, i.e., a support for the dynamic software deployment is not considered.

There are also some works which focus on upgrading safety-critical systems [20] [21] [22]. One of the most notable is work done in the scope of the project PINCETTE, which has as a goal to perform live upgrades of software systems that control the safety-critical processes [20]. Although the topic is beyond the scope of available validation methods in the practice, the aim is to evaluate the feasibility of formal methods to such use cases. In contrast to our data flow-oriented analysis, the focus here is on validating the interaction between upgraded behaviours. Another work [21], done in the scope of the RECOMP project, addresses also live upgrades as one of the goals to reduce the costs for certifying systems.

[3]This collection is partially taken from our previous work in [6]

UC	# of recalls	# of software related	Change effort C_{eff}	Cost reduction $(\cdot 100)$ [%]
1	315	82	$0.229 \cdot (c_{dev_1} + c_{cert_1})$ $+...$ $+0.317 \cdot (c_{dev_6} + c_{cert_6})]$ $+0.031 \cdot (c_{dev_f} + c_{cert_f})$	$\frac{0.031 \cdot (c_{dev_f} + c_{cert_f})}{0.031 \cdot (c_{dev_f} + c_{cert_f}) + [0.229 \cdot (c_{dev_1} + c_{cert_1}) + ...]}$
2	1210	778	$[0.565 \cdot (c_{dev_1} + c_{cert_1})$ $+...$ $+0.117 \cdot (c_{dev_5} + c_{cert_5})]$ $+0.077 \cdot (c_{dev_f} + c_{cert_f})$	$\frac{0.077 \cdot (c_{dev_f} + c_{cert_f})}{0.077 \cdot (c_{dev_f} + c_{cert_f}) + [0.565 \cdot (c_{dev_1} + c_{cert_1}) + ...]}$

TABLE IV. COST REDUCTIONS COMPARED TO CHANGE EFFORT C_{eff} FOR USE CASES 1 AND 2

However, only dynamic linker has been realized here, without considering the analysis of changes. Finally, the work in [22] shows how to validate changes of upgraded safety-critical system. Here, a model checker is used for the verification

V. CONCLUSION

In this paper, we described an approach to perform changes on safety-critical embedded systems in the operation and maintenance phase. The main contribution here is that controlled changes on the application software can be performed dynamically, without having an impact on systems integrity, and therefore without requiring to re-verify, re-validate and finally to re-certify the system. Performing such controlled changes can bring benefits in maintaining safety-critical systems, especially if there are high maintenance demands on the application software.

We analyzed here the possible reduction in certification costs when applying the approach. To this end, we considered here two exemplary use cases, which describe recalls caused by failures of embedded systems in domains of automotive and medical engineering. Some of these failures are caused by faults introduced by the application software. For instance, out of 1210 recall actions in medical engineering, 7,7% are caused by faults in the control algorithms. With the proposed approach, repairing software from such faults such can be managed without introducing costs for the certification. Based on a simple cost model, and the distribution of faults to system components, we showed the possible reduction in certification costs when applying this approach.

REFERENCES

[1] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," *Computer*, vol. 42, no. 4, pp. 42–52, April 2009.

[2] O. Kindel and M. Friedrich, *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt Verlag; Auflage: 1, 2009.

[3] H. Alemzadeh, R. Iyer, Z. Kalbarczyk, and J. Raman, "Analysis of safety-critical computer failures in medical devices," *IEEE Security Privacy*, vol. 11, no. 4, pp. 14–26, 2013.

[4] Qi Van Eikema Hommes, "Applying STAMP Framework to Analyze Automotive Recalls," The National Transportation Systems Center, Research report, 2014.

[5] D. Smith and K. Simpson, *A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849*. Elsevier Science, 2010.

[6] N. Kajtazovic, C. Preschern, A. Hoeller, and C. Kreiner, "Towards assured dynamic configuration of safety-critical embedded systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8696, pp. 167–179.

[7] M. Glinz, "On non-functional requirements," in *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, Oct 2007, pp. 21–26.

[8] I. Crnkovic, *Building Reliable Component-Based Software Systems*. Norwood, MA, USA: Artech House, Inc., 2002.

[9] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for Systems Design," Research Report, Nr. 8147, 2012, Inria, Tech. Rep., 2012.

[10] P. Frey, "Case Study: Engine Control Application," Ulmer Informatik-Berichte, Nr. 2010-03, Tech. Rep., 2010.

[11] N. Kajtazovic, C. Preschern, A. Hoeller, and C. Kreiner, "Constraint-based verification of compositions in safety-critical component-based systems," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ser. Studies in Computational Intelligence. Springer International Publishing, 2015, vol. 569, pp. 113–130.

[12] FAA, "Guidelines for the Oversight of Software Change Impact Analyses used to Classify Software Changes as Major or Minor," FAA, Notice 8110.85, 2000.

[13] K. Marriott and P. J. Stuckey, *Programming with Constraints: An Introduction*. The MIT Press, Mar. 1998.

[14] N. Kajtazovic, C. Preschern, and C. Kreiner, "A component-based dynamic link support for safety-critical embedded systems," in *20th IEEE ECBS*, 2013.

[15] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (alma)," *J. Syst. Softw.*, vol. 69, no. 1-2, pp. 129–147, Jan. 2004.

[16] R. Adler, I. Schaefer, M. Trapp, and A. Poetzsch-Heffter, "Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 20:1–20:39, Jan. 2011.

[17] G. Montano, "Dynamic reconfiguration of safety-critical systems: Automation and human involvement," *PhD Thesis*, 2011.

[18] M. Oertel, O. Kacimi, and E. Bde, "Proving compliance of implementation models to safety specifications," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8696, pp. 97–107.

[19] M. Oertel and A. Rettberg, "Reducing re-verification effort by requirement-based change management," in *Embedded Systems: Design, Analysis and Verification*, ser. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2013, vol. 403, pp. 104–115.

[20] M. Zhang, K. Ogata, and K. Futatsugi, "Formalization and verification of behavioral correctness of dynamic software updates," *Electronic Notes in Theoretical Computer Science*, vol. 294, no. 0, pp. 12 – 23, 2013, proceedings of the 2013 VSSE Workshop.

[21] P. Pop, L. Tsiopoulos, S. Voss, O. Slotosch, C. Ficek, U. Nyman, and A. Ruiz, "Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the recomp approach," *WICERT*, 2013.

[22] D. Soliman, K. Thramboulidis, and G. Frey, "A methodology to upgrade legacy industrial systems to meet safety regulations," in *Dependable Control of Discrete Systems (DCDS), 2011 3rd International Workshop on*, June 2011, pp. 141–147.

20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)

A Component-based Dynamic Link Support for Safety-critical Embedded Systems

Nermin Kajtazovic
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
nermin.kajtazovic@tugraz.at

Christopher Preschern
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
christopher.preschern@tugraz.at

Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
christian.kreiner@tugraz.at

***Abstract*—Safety-critical embedded systems have to undergo rigorous development process in order to ensure that their function will not compromise humans or environment where they operate. Therefore, they rely on simple and proven-in-use design. However, with growing software complexity, maintenance becomes very important aspect in safety domain. Recent approaches for managing maintenance allow to perform changes on software at design-time, which implies that the whole system has to be rebuilt when the application software changes. In this paper, we describe more flexible solution for updating the application software. We apply the component-based paradigm to construct the application software, i.e. we define a model of a software function that can be dynamically linked with the entire operating system (OS). In order to avoid the usage of the OS-provided support for dynamic linking, we design software functions as position-independent and relocation-free binaries with well-defined interfaces. With the help of component-based paradigm we show how to simplify the link support and make it suitable for safety domain.**

***Keywords*-component-based engineering; safety-critical embedded systems, dynamic linking**

I. Introduction

Maintenance in software engineering is important aspect to cope with changes in software environment. Software as a product continuously undergoes changes which may originate from request to fix errors, to adapt the product to new requirements or to improve the product's performance [7]. Built-in operating system (OS) mechanisms such as dynamic linking and loading are the main driver for updating software products, but they do not ensure that the products are designed for maintenance. Thus, the crucial factor here is the software architecture. In the last decade, the expansion of component-based paradigm improved the maintenance considerably, since the software architecture has moved from monolithic structure to modular design. Because of loose coupling between parts that constitute the component-based software products, maintenance can be managed at lower costs.

Component-based paradigm has also been identified as a key tool for developing software for safety-critical embedded systems. The examples are AUTOSAR Standard for automotive domain and IEC61499 Standard for industrial automation. Systems in this domain have to undergo rigorous development process, because their correct function is crucial for humans and environment where they operate. Therefore, they are conservatively designed and lack a lot of features present in general purpose systems. Software maintenance is also very limited here. In AUTOSAR [2] for example, each change request to the application software requires to compile and to rebuild the whole system (i.e. OS and application). The component models for real-time systems such as SaveCCM [8] and SOFA-HI [9] have also the same strategy for managing software updates. The ability to update the application software without compiling or linking it statically with the OS would be very practical, in particular for domains where the application and OS development are in responsibility of separated organizations. The application developer would therefore be only confronted with its own application logic. However, the existing OS support for dynamic linking is relatively complex and therefore it is difficult to assure its quality. Further, it has higher runtime overhead than static linking model. Some alternatives for dynamic linking in domain of wireless sensor networks (WSN) have been proposed, but they use static indirection tables (fixed memory addresses) as interfaces between the application software and OS services [4], [5]. Another domain of alternatives for dynamic linking are DSU systems (Dynamic Software Updating), which perform online patching of software [13], [11]. Although DSU systems are far away of safety domain, they use more flexible models for dynamic linking.

In this paper, we propose a component-based dynamic link support for safety-critical embedded systems. We show how the component-based paradigm can be utilized to make the link support simple and easy to verify. Compared to existing solutions that use static linking, the application developers can benefit from improved flexibility.

The remainder of this paper is organized as follows: Section II summarizes the requirements for the linker related to safety domain. Section III introduces the component-based paradigm and outlines the existing linking models. In Section IV, the proposed component-based dynamic link support is described. Section V describes how the safety is addressed in the linking process. Section VI covers the evaluation and concluding remarks are given in Section VII.

978-0-7695-4991-0/13 $26.00 © 2013 IEEE
DOI 10.1109/ECBS.2013.19

II. Requirements for Dynamic Linker

In this section we briefly summarize some of the most relevant requirements that the target system (including the dynamic linker) has to satisfy in safety domain. The system in our context is a software comprising the OS and the application software on top of that OS. We further use the linker requirements to evaluate our contribution.

The requirements are:

- Flexibility: the system shall not be re-built when the application software changes. That is, the OS shall be able to exchange the application at least at load-time (i.e. system initialization phase). The granularity of exchange shall be the application binary image at least.
- Safety: many safety-critical systems strictly follow the principle of simplicity, thereby reducing the system complexity and risk of failures. As a consequence, they also reduce the overhead of getting certified (i.e. evidence that they will operate correctly with certain failure rate). Concretely for the linker support, the correctness of the dynamic linking shall be predictable. That is, the correctness of the interfaces between the application and the OS shall be ensured before the system can compromise safety. Therefore, the linker support shall be as simple as possible.
- Low runtime overhead: compared to statically linked image, the system shall not introduce significant runtime overhead between the application and the OS.

III. Background

In this section, we introduce the component-based paradigm. The intent is to show which aspects of component-based paradigm are relevant for this work. Further, we review linking models for Unix-like systems and show their implications on safety-critical embedded systems.

A. Component-based Engineering

Component-based Software Engineering (CBSE) is a paradigm to systematically design software for reuse and maintenance [1]. The literature offers different definitions for a software component, but in general, a software component represents a reusable piece of software that is used for modular construction of software applications. This means that a binary object file with some functions can be defined as a component for example, but actually, a component is more than that. To ensure reuse, the system which follows the component-based paradigm has to provide features to independently deploy components. Further, it has to care about dependencies introduced between components and to manage their lifecycle. In fact, the change of software (due to reconfiguration for example) shall only be a subject of deployment, rather than an implementation issue. Therefore, systematic reuse has the implications on design of components and their execution environments as well. Such a component-based system is constituted by following parts:

- Components: A component consists of an implementation (application logic) and interfaces. The interfaces act as an access point to the component implementation from the outside. They are usually the only visible part of components and describe the implementation syntactically (e.g. signature of operations) and semantically (e.g. contracts set on operations). The interfaces are the main driver for ensuring reuse, because components access and provide the functionality only through their local interfaces and are therefore not dependent on other components.
- Framework (Container): A container is an execution environment for components. It takes the responsibility for the part of the component lifecycle, in which a component serves as a part of the software application.

Another important aspect of the CBSE is a development process, which, in contrast to the process used for monolithic design, distinguishes between system development and component development [3]. Within the later process, a component undergoes several phases until it is ready for deployment. Thereafter, a component is used in system development process to assemble the software application.

The specification of the framework, its features and components is often referred as a component model.

B. Linking Binary Code

Object linking is a process which combines object files in order to produce application image (executable) or library. Two aspects of this process are important when changing software: granularity level (which is an object file) and time when linking is performed. In following, we introduce these two aspects.

1) Executable and Linkable Format (ELF): Object files that participate in the linking process are, for Unix-like systems, represented in the ELF binary file format. This format describes, besides a raw binary, the memory layout of a binary. This information is essential when linking independently created object files or when loading them at runtime. Depending on binary type, ELF provides two views on how to express memory layouts: linking view - code and data are temporarily stored in sections for further linking, and executable view - final memory layout that is used at runtime.

This organization of the binary ensures that code, data and the entire references are flexible regarding their memory location and can be aligned to the address space of other object files. The information for dynamic linking, in particular dynamic relocations and linker reference, persist in special sections (see Sec. III-B2).

	Linker Requirements		
Linking model	Flexibility	Safety	Runtime overhead
Static link	- Re-linking is required.	+ All symbol references are resolved at design-time and binary (i.e. symbol references) can be verified at design-time.	+ No runtime/load-time overhead. All symbols are resolved at design-time.
Shared library	+ Update of the application image at granularity of an object file.	- Symbol references cannot be verified at design-time. - Binary is modified at runtime.	- Load-time overhead due to symbol resolution (O(n) for n references).
PIC	+ Same as shared library.	- Same as shared library.	- Load-time and runtime overhead due to symbol resolution and indirection table respectively.
Binary rewriting	+ Update of the application image at granularity of a symbol (patching symbols).	+ Symbols can be statically resolved. - Binary is modified at runtime.	+ If no indirection used, same as static link model.
Indirection tables	+ Update of the application image at granularity of a symbol (redirecting symbols).	+ Symbols can be statically resolved. + Binary is not modified at runtime.	- Potential performance penalty (depends on implementation).

Table I
COMPARISON OF LINKING MODELS WITH RESPECT TO REQUIREMENTS INTRODUCED IN SEC. II

2) Linking Models: Basically, ELF object files can be statically or dynamically linked. Further, dynamic linking can be performed either at load time or at runtime. Both linking models have different impacts on performance: statically linked executables have no runtime overhead since all external symbols are already resolved (i.e. an object file has no relocations), but they require much more space than dynamically linked executables [6]. Though their memory demands, they are still very important for embedded systems. In following, we describe object file types that are used to facilitate static and dynamic linking:

Relocatable Code: In static linking model, several object files are combined to a single application image, i.e. an executable object file. A relocatable object file typically consists of the unresolved symbol references. For instance, if an access to a symbol in an external library has been made, a relocation entry for this reference will be created in the compile phase. This indicates that the linker has to fix this relocation when it is aware of the location of the imported library. Fig. 1 illustrates this scenario. Function `mainFct` uses a function that is available in another object file. As a result of such an unresolved reference, a single relocation entry is created (at offset c in Fig. 1b). At link-time, this relocation entry is removed (see Sec. III-B3).

Shared Libraries: The problem of static linking is that required object code is imported in every executable. This problem is solved by sharing that code among other applications. Correcting the relocations in shared library model is performed earliest at application load-time. To trigger that correction, the location of the dynamic linker or loader is set in the `.interp` section of the ELF file. When the application starts, the loader is first started to fix all relocations.

Position-independent Code: The idea behind position-independent code (PIC) is to have only relative symbol references within an ELF file. The independence between symbol references and their real locations is made using the indirection table, the Global Offset Table (GOT) [10]. Thus, each reference to an external symbol is an offset to a specific entry in this table. When fixing relocations, only the address of the first GOT entry has to be added to this offset. However, this address is absolute one and has to be previously known. In most processor architectures, the GOT entry is stored in a separated register. The PIC model is also practical for MMU-less (Memory Management Unit) systems, because the relative addressing allows to place the code anywhere in the address space. The uCLinux[1] OS, for instance, uses this model to allow dynamic linking for MMU-less embedded systems.

3) Symbol Resolution: Resolving symbol references and correcting relocations is not a trivial process. For functions, the intent is to change the operand of a branch instruction that points to an external symbol. In Fig. 1b, the relocation entry of the type `R_ARM_CALL`, which is created for the symbol `externFct`, is used to identify the right oper-

[1]ucLinux Homepage - http://www.uclinux.org/

```
extern void externalFct();

void mainFct(){
   externalFct();
}
```

```
void externalFct(){
   ...
}
```

(a)

```
00000000 <mainFct>:
  0:    e1a0c00d mov     ip, sp
  4:    e92dd800         stmdb   sp!, {fp, ip, lr, pc}
  8:    e24cb004 sub     fp, ip, #4  ; 0x4
  c:    ebfffffe bl      0 <externalFct>
                         c: R_ARM_CALL     externalFct
 10:    e89da800         ldmia   sp, {fp, sp, pc}
```

(b)

Figure 1. Symbol relocation: (a) sample code and (b) its assembly code

ation that shall be applied to get the correct address of `externFct`. In architectures that support relative addressing (i.e. relative to program counter (PC)), this value will be an offset that has to be added to the current value of the PC to resolve the `externFct`. Shared libraries or PIC have different resolution schemes, and therefore, different relocation types are applied there.

4) Alternatives for Dynamic Linking: Introduced linking models have the granularity of object files. In some domains, however, there is a need for more fine granular linking where functions, their signatures, data, etc. can be changed at runtime. The examples are distributed systems with high availability and maintenance demands. Linking is here based either on binary rewriting technique or indirection tables [13]. Former technique modifies the original binary at runtime, in order to establish the symbol references between loaded binary and application or to upgrade a binary. The indirection tables, in contrast, just redirect the symbol references between dependent binaries. This technique is very similar to PIC.

In Tab. I we summarize the introduced linking models and show their characteristics with respect to requirements from Sec. II.

IV. Proposed Component-based System

Both standard models for dynamic linking are not suitable for safety-critical embedded systems, since results of the symbol resolution cannot be predicted at design-time. For binaries which are safety certified this is an issue, because they have to be modified in the symbol resolution process. The linker, on the other side, has to undergo more extensive validation and verification process in order to ensure that symbol resolution will not fail. More effective way would be to avoid the load-time and runtime symbol resolution, but relocations within a binary make this impossible.

Our work is based on indirection tables, which act as interfaces between the OS and applications. The OS maintains a set of indirection tables that consist of the OS symbols required by the applications. To avoid the symbol resolution, the applications follow some design and implementation constraints. In this section, we describe our linker as a part of the component-based system.

A. Component-based System Architecture

According to Crnkovic and Larson [1] a component-based system can be separated in concepts forming the framework layer (i.e. a component container), the interface model (interface between components and container) and component implementations. Based on this model, we separate our component-based system into static and dynamic parts. The former is the OS, which is statically linked with the container and required libraries. The dynamic part corresponds to the applications which are subject to dynamic linking. Fig. 2

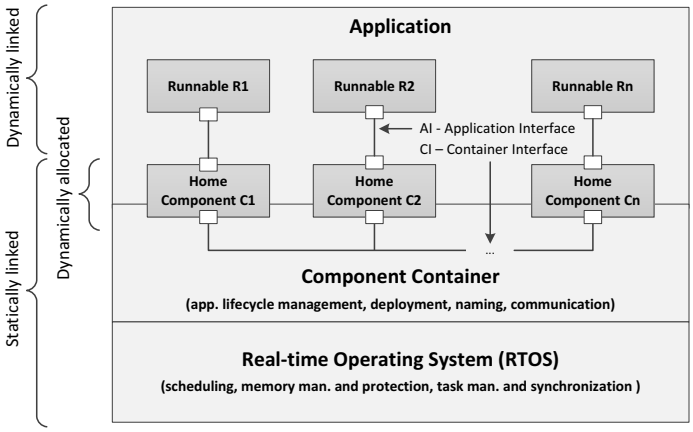


Figure 2. Layered architecture of the proposed component-based system

shows the layered architecture of that system. To ensure dynamic linking for the application layer, following constraints have to be satisfied:

- the application shall not have relocations,
- the application shall be position-independent (for single-address space systems),
- the container shall support dynamic memory management (for varying number of components).

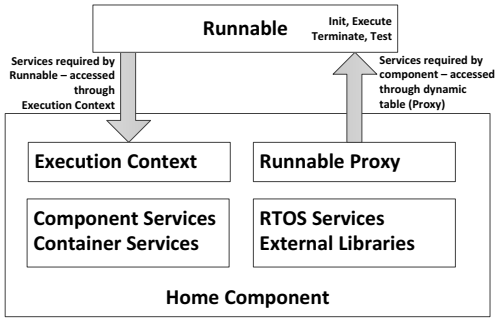
By applying the CBSE the application is divided into components, which can be independently deployed. Therefore, it is sufficient to satisfy the first two constraints for components only. In the architecture above, the component implementations, called *runnables*, constitute the application logic. Their lifecycle and inter-connections between *runnables* are maintained by the container in the OS. For each of the runnables the container allocates and instantiates corresponding *home component*. Both parts, i.e. *runnable* and *home component*, constitute a software component in the proposed component-based system. In following, we describe the component architecture more in details.

B. Software Component Architecture

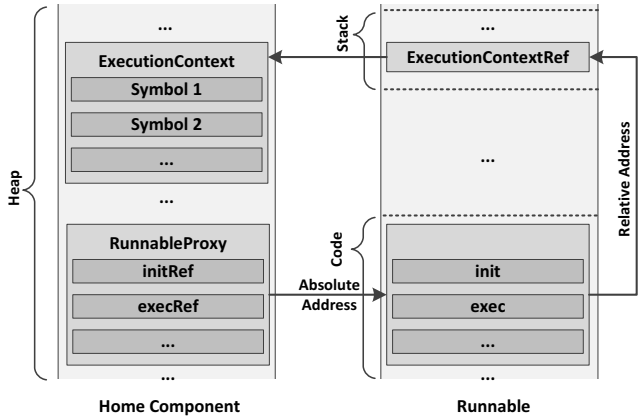
The *runnable* introduced in previous section represents a piece of code that implements a part of the application logic (PID controller for example). The other part of the component, i.e. *home component*, consist of non-functional component features such as interfaces for inter-component communication, lifecycle methods, etc (see Fig. 3a). The reason for this separation is to avoid relocations in the application logic. The relocations can be avoided if a binary has no explicit dependencies to external symbols. The *runnables* are, therefore, fully isolated binaries and make no references to external symbols. Some of the runnables are stateful and have to maintain their local storage. Referencing other segments from code would also introduce relocations. Therefore, *runnables* maintain their local storage within corresponding *home components*. For this purpose, *home*

components manage a generic storage pool that can be provided to any *runnable*.

For systems with single-address space the *runnables* have to use relative symbol addresses only. This ensures the position-independence of a binary and consequently multiple applications can be hosted by the container. Therefore, the indirection table is placed on the stack of the container. All symbols which are required by *runnables* are included in this table. The indirection table acts as required interface from *runnable* viewpoint. The *runnable* functions, on the other side, are provided interfaces used by the container. In following, we describe the interface model for dynamic linking.



(a)



(b)

Figure 3. Software component: (a) architecture and (b) runtime memory layout

C. Interface Model for Dynamic Linking

In Fig. 2 we depicted two types of interfaces used in the proposed component-based system: application interfaces (AI) for interaction between components and runnables (i.e. interfaces for dynamic linking), and (2) container interfaces (CI) for interaction between components and container.

1) Application Interfaces: Fig. 3a shows the architecture of the software component and interfaces between *runnable* and its *home component*. In this interaction model it is required that *runnables* can access their *home components* and vice versa. The interfaces *ExecutionContext* and

```
bool execFct( IExecutionContext* context )
{
  ...
  double setP = context->getInterface( DIN1, ... );
  double currV = context->getInterface( DIN2, ... );

  ...
  context->setEvent(EOUT1, val);
  ...
}
```

Figure 4. Application interfaces between *runnable* and *home component*: communication from *runnable* to container (example)

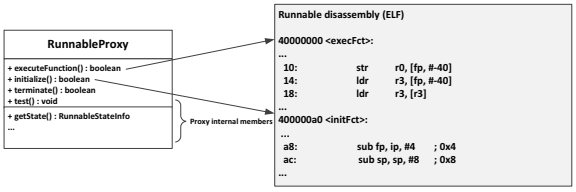


Figure 5. Application interfaces between *RunnableProxy* and *runnable*: communication from container to *runnable* (example)

RunnableProxy ensure this. They form the indirection tables between dynamically linked *runnables* and the container. The interface *ExecutionContext* exposes the services (i.e. functions) that *runnables* need in order to perform their function. From the *runnables* viewpoint, this interface is required interface and allows *runnables* to access to: external libraries, container services for inter-component communication, OS services for resource management and component services for local resource management (i.e. state allocation, interface access, etc.). An excerpt of this type of interfacing is depicted in Fig. 4, where the *runnable*, which implements a PID controller in `C`, requires a data interface from its *home component*. The access to the *ExecutionContext* is made over the stack of the container (see Fig. 3b), since stack variables are accessed relative to some position (e.g. relative to frame pointer in ARM). All references to mentioned OS services and libraries are provided by the *home component* (see *home component* in Fig. 3a).

Another direction is realized by the interface *RunnableProxy*. It references four standard methods of runnables (i.e. provided interface): *Init*, *Execute*, *Terminate* and *Test*. Fig. 5 shows the structure of the *RunnableProxy* and an exemplary runnable assembly code. The references to *runnable* methods are function pointers that are initialized by the container. For establishing these references the container adds the load address of the runnable to the offset of each method. This is most critical operation in dynamic linking, because the incorrect addresses could lead to a failure and compromise safety. Therefore, runtime verification shall ensure that this cannot occur (see Sec. V-B).

2) Container Interfaces: This type of interfaces is used (1) to interrelate software components and (2) to provide container, OS services and libraries to software components.

The *runnable*, for instance, can access another *runnable* by setting the specific event that is handled by the *home component*. The *home component* in turn triggers the execution of the target *runnable*. On a similar way, the runnables access the external libraries and other services[2].

V. Ensuring Safety in Dynamic Linking

In this section, we show how the correctness of the proposed dynamic link support is ensured.

Generally, two types of faults can occur in the linking process: (1) interface faults due to incorrectly estimated addresses of *runnable* interfaces, and (2) component faults due to incorrectly produced *runnable* binary. The former are faults caused by the container, and occur when the container makes an access to *runnable* interfaces using the incorrect absolute addresses. Component faults, on the other side, are concerned with *runnables* only. They occur when the *runnables* do not follow the specification of the component model (constraints in Sec. IV-A are just a part of that specification). Examples are incompatible versions of *runnables*. Note that any of these faults can cause the failures with major consequences [12]. To ensure the absence of these faults, we perform the verification in two phases: at design-time and at load-time.

A. Design-time Verification

Faults in *runnables* are typically caused in component development process (e.g. systematic faults by developer). In our context, a *runnable* is faulty if it consists of relocations, or does not have the interface that is conform to *RunnableProxy*, or is eventually not *runnable* binary, etc. In fact, if the *runnable* binary does not conform to the component model, it can cause the address failure when loaded by the container. Therefore, as a part of the component development process, we verify each of the *runnables* for conformance with the component model. This verification is automated process and has following phases:

- Architecture verification: inspects the binary according to the target processor architecture (e.g. CPU type, address width, endianness, etc.).
- Binary verification: inspection according to the binary (e.g. relocatable binary)
- Symbol verification: inspection according to the structure of binary (e.g. relocations, segments and sections, aligning, provided symbols, etc.).

B. Load-time Verification

Verification performed at design-time does not ensure that the container will correctly host each of the *runnables*. As described in Sec. IV-C it is obvious that the estimation of the resulting absolute address of each of the *runnable* methods is very simple, i.e. each of the methods follows the same scheme: load address of a *runnable* added to the offset of the method in a binary. However, to guarantee that all interfaces are inspected before the system starts operating, they have to be tested. For this purpose, the *runnables* provide a test method. This method takes the estimated addresses of *runnable* methods as a parameter. When the container starts the execution, the test method inspects all other runnable methods by calling them. Negative test would mean addressing failure, which in the worst case cannot be handled. Therefore, test execution is performed only once in system initialization phase, where the system cannot compromise safety.

[2]The execution semantic of components and their domain-specific characteristics are out of scope in this work.

VI. Evaluation

In this section, we evaluate the introduced dynamic linking using requirements from Sec. II. For observing quantitative results, we implemented a prototype of the proposed component-based system in C language. The prototype has been launched on an ARM processor with 454MHz, 128MB of RAM.

A. Qualitative Results

1) Flexibility: As in usual systems with dynamic link support it is possible to update the software application without rebuilding the system. The granularity of the update is a *runnable* binary. Because of safety requirement in Sec. II, the link can only happen at load-time, i.e. at system initialization time. Another advantage of applying the CBSE is that the application developers do not have to consider the container and OS in their application (component) development process. They just use the OS-provided services to deploy their software components.

2) Safety: One of the most important outcomes of applying the CBSE for dynamic linking is that the results of the linking can be predicted at design-time. Compared to the standard dynamic link model, there are no symbol references which are unknown until load-time or runtime. The *runnable* methods (provided interfaces) are simply estimated by adding the load address of a runnable to the offsets of the methods. Further, the required symbols (required interfaces) are placed by the container on the stack. As shown in Sec. V, design-time verification process ensures that the container always gets the correct *runnable* binary. Therefore, the only changing part in dynamic linking are the addresses of the *runnable* methods, but they always have the same estimation scheme.

To sum it up, dynamic linking of the *runnables* is safe, since the *runnables* do not contain any symbols which have to be estimated at runtime by using a relocation scheme, which is present in standard linking models. Further, the symbols required by the *runnables* are part of a data structure called *ExecutionContext*, which is statically linked with the OS and the container. Therefore, each symbol which

	#Iterations / Runtime [μs]		
Linking model	50	100	150
Static link	1,67	3,31	4,98
Proposed (with context)	2,40	4,77	7,33
Proposed (without context)	1,96	3,92	5,82

Table II
BENCHMARK 1: RUNTIME OVERHEAD FOR PROVIDED INTERFACES

	#Iterations / Runtime [μs]		
Linking model	50	100	150
Static link	<1	1,48	2,19
Proposed (with references)	1,20	2,35	3,47
Proposed (with data)	<1	1,49	2,18

Table III
BENCHMARK 2: RUNTIME OVERHEAD FOR REQUIRED INTERFACES

is intended to be used by the *runnables* can be verified at design-time, i.e. before delivery of the OS and container.

Another advantage of using the CBSE is that no type safety checks have to be performed, as it is done by some linkers [14]. Because *runnables* implement standard interfaces, type casting to *RunnableProxy* will always work.

B. Quantitative Results

1) Runtime Overhead: Dynamic indirection tables *ExecutionContext* and *RunnableProxy* consist of function and data pointers for function and data symbols respectively. Generally, it is expected that indirection tables introduce an additional runtime overhead compared to statically linked binaries [13]. In order to estimate this overhead, we have constructed two benchmarks:

- benchmark for provided interfaces - runtime overhead for calling the *runnable* from the container using the *RunnableProxy* interface,
- benchmark for required interfaces - runtime overhead for calling the container from the *runnable* using the *ExecutionContext*.

For both benchmarks, no implementations are provided within method bodies, because only the overhead of indirection tables is relevant. We performed tests in three iterations, each of them consisting of 50, 100 and 150 interface calls. The average value for runtime is computed using 1000000 samples for each iteration.

For the first benchmark, we first estimate the runtime overhead without the *ExecutionContext*, i.e. nothing on the stack of the container, in order to just consider the provided interfaces. Thereafter, we include the *ExecutionContext*, since it stays always on the stack for all *runnable* methods, regardless of if it used or not.

For the second benchmark, we estimate the runtime overhead once for the *ExecutionContext* containing the references to required data (i.e. data pointers) and once containing the direct data. These are possible configurations for the *ExecutionContext*. When referencing functions, only the variant with references can be used.

Observations: Tab. II shows the results of the first benchmark. For the variant without the *ExecutionContext* on the stack (Proposed (without context)), there is a negligible runtime overhead, which originates from de-referencing function pointers. Direct calls, which are used in statically linked variant, do not need to perform de-referencing. More runtime overhead is required for the variant with the *ExecutionContext* on stack. The reason is that the *ExecutionContext* has to be copied on the stack before calling the provided interfaces. Thus, compared to static link model, the overall additional runtime overhead for provided interfaces r_{PI} is:

$$r_{PI} = r_{copy\_on\_stack} + r_{dereference\_pointers}$$

Runtime overhead for required interfaces depends only on number of pointers that have to be de-referenced. When requiring data symbols for example, it is possible to use direct data instead of data pointers. In this case, there is no additional runtime overhead, compared to static link model (see Proposed (with data) in Tab. III). In the worst case, the additional runtime overhead for required interfaces r_{RI} is:

$$r_{RI} = r_{dereference\_pointers}$$

Note that the results above represent the runtime overhead of interfaces between the *home component* and *runnable*. That is, the values are estimated on *runnables* and container with empty method bodies. For real-life examples, the implementation if often the main contributor to the runtime overhead, and the values above are in this case often negligible. To demonstrate the variant with the implementation, we constructed an exemplary case study that implements *runnable* provided interfaces (its function is not relevant here). Tab. IV shows the results. The first two columns show the number of pointers that need to be de-referenced within a single iteration, and the third column shows the runtime. As expected, slight deviations from results for static link model are observed.

Linking model	# pointers in Ex.Ctx.	# pointers in R.Proxy	Runtime [μs]
Static link	0	0	4,96
Proposed (with references)	65	3	5,20
Proposed (with data)	49	3	4,93

Table IV
CASE STUDY: RUNTIME OVERHEAD FOR IMPLEMENTATION AND INTERFACES

C. Implications on Target System

Implementing *runnables* as relocation-free binaries is an implementation issue, because the developer is constrained to a very small set of the features provided by the programming language. All symbols required by a *runnable* have to be accessed using the *ExecutionContext* interface (see Fig. 4 for example). Therefore, any eventual relocation has to be manually fixed by the developer.

VII. Conclusion

In this paper we presented a component-based dynamic link support for safety-critical embedded systems. Compared to approaches that rely on static linking or alternatives for dynamic linking that use static indirection tables, better flexibility can be achieved when applying the CBSE. More importantly, the results of the link process can be predicted at design-time, which makes the link support suitable for safety domain.

Following the CBSE, we defined a model of a software component that consists of dynamic and static parts and ensures loose coupling between them. The dynamic part is represented as a so-called *runnable*, a position-independent and relocation-free binary that forms a part of the application software. Only *runnables* in the proposed component-based system can be dynamically linked. Static part of a component, which is represented by a so-called *home component*, is managed by the container and is statically linked with the OS. The essential part of the container that contributes to dynamic linking is an indirection table, which consists of the symbols required by the *runnables* to perform their function. This indirection table is statically linked with the container and therefore the locations of the symbols required by the *runnables* are always predictable.

Compared to pure static link model, an expected increase of runtime overhead has been observed. Because of usage of the indirection tables as interfaces between *home components* and *runnables* it is often required to make the indirections to get desired data.

One of the major limits in our component-based system is that runnables have to access their resources using the standard interfaces, which obligates the developer to follow the certain guidelines when implementing *runnables*.

References

[1] Crnkovic I. and Larsson M., *Building Reliable Component-Based Software Systems*. Artech House Publishers, ISBN 1-58053-327-2, 2002.

[2] AUTOSAR: *Software Component Template*, AUTOSAR Homepage: http://www.autosar.org, 2010.

[3] Crnkovic I. and Chaudron M. and Larsson S.: *Component-Based Development Process and Component Lifecycle*, Proceedings of the International Conference on Software Engineering Advances (ICSEA '06), 2006, IEEE Computer Society, Washington, DC, USA

[4] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava: *A Dynamic Operating System for Sensor Nodes*. In Proceedings of the 3rd international conference on Mobile systems, applications, and services (MobiSys '05). pp. 163-176, ACM, New York, NY, USA

[5] Wei Dong; Chun Chen; Xue Liu; Jiajun Bu; Yunhao Liu: *Dynamic Linking and Loading in Networked Embedded Systems*, IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, MASS 2009, 12-15 October 2009, Macau (S.A.R.), China

[6] Franz, M.; *Dynamic Linking of Software Components*, Computer, vol.30, no.3, pp.74-81, Mar 1997

[7] April A. and Abran A., *Software Maintenance Management: Evaluation and Continuous Improvement*. Wiley-IEEE Computer Society Press, May 2008.

[8] Hansson H., Akerholm M., Crnkovic I., and Torngren M.: *SaveCCM - A Component Model for Safety-Critical Real-Time Systems*. In Proceedings of the 30th EUROMICRO Conference (EUROMICRO '04). IEEE Computer Society, Washington, DC, USA, 627-635, 2004.

[9] Hosek P., Pop T., Malohlava M., Hnetynka P., Bures T.: *Supporting Real-Time Features in a Hierarchical Component System*, Tech. Report No. 2010/5, Dep. of Distributed and Dependable Systems, Charles University in Prague, December 2010

[10] Levine J. R.; *Linkers and Loaders*. Morgan Kaufmann, 256 pages, ISBN 1558604960, 9781558604964, 2000.

[11] Hayden, C.M. and Smith, E.K. and Hardisty, E.A. and Hicks, M. and Foster, J.S.; *Evaluating Dynamic Software Update Safety Using Systematic Testing*. IEEE Transactions on Software Engineering, 2012.

[12] Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C.; *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Transactions on Dependable and Secure Computing, vol.1, no.1, pp. 11- 33, Jan.-March 2004

[13] Hicks M. and Nettles S. *Dynamic Software Updating*. ACM Trans. Program. Lang. Syst. 27, 6 (November 2005), p. 1049-1096.

[14] Michael W. Hicks, Stephanie Weirich, and Karl Crary: *Safe and Flexible Dynamic Linking of Native Code*, In Selected papers from the Third International Workshop on Types in Compilation (TIC '00), 2000, Robert Harper (Ed.). Springer-Verlag, London, UK, 147-176.

Inversion of Control Container for Safety-critical Embedded Systems

Nermin Kajtazovic, Institute for Technical Informatics, Graz University of Technology
Christopher Preschern, Institute for Technical Informatics, Graz University of Technology
Christian Kreiner, Institute for Technical Informatics, Graz University of Technology

It is common for safety-critical embedded systems that strategies for software reuse and maintenance are mainly planned and managed at design-time. Currently, most frameworks for managing application software in the safety domain use a component-based paradigm to allow applications to be constructed from reusable parts. This construction is mainly conducted at design-time in order to comply with stringent safety requirements. In contrast, the runtime support for construction of the application software would be very practical, particularly for domains where the framework and application development are in the responsibility of different organizations. Further, the overhead of maintaining these systems would be considerably reduced. However, standard mechanisms for handling such dynamic construction are not applicable to the safety domain and therefore frameworks in this domain are not designed for such use cases.
In this paper, we present a component-based framework with the architectural support for runtime application construction and maintenance. To allow this, we apply the inversion of control principle, which is a technique used in many domains to ensure loose coupling between interrelated components and consequently to make them reusable. Since there are many ways on how to implement the inversion of control, we show its realization for resource-constrained systems by applying design patterns which are primarily used for object interaction and resource management in general purpose systems.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*

1. INTRODUCTION

Inversion of control (IoC) is a widely applied technique to systematically design software for reuse and maintenance. It offers a pragmatic way to ensure loose coupling between dependent parts that constitute a software application (in further text components). In most common scenarios, the control over explicit dependencies between components is shifted to the infrastructure, thereby ensuring flexible construction of the application software[1]. Because of the cross-domain support, the IoC relies on different strategies and idioms [Prasanna 2009]. For the most flexible variant of the IoC, it is not only necessary to provide a good architectural support, but also to have a system that would allow to achieve the desired degree of flexibility. For instance, the operating system (OS) support for dynamic linking may be required when reconfiguring the application at runtime. In this context, the existing IoC containers for enterprise applications such as Spring or PicoContainer use Java class loaders to dynamically load and link dependent components [SpringSource 2013;PicoContainer 2013].

In the safety domain, the application construction is mainly fixed at design-time. That is, there is no support for maintaining dependencies between components at runtime. The reason for such a limited flexibility is meeting a trade-off between safety and maintainability on one hand, and resource limitations on the other hand. To address

[1] Considering that a software system is composed of the infrastructure and applications on top of that infrastructure.

the former concern, safety-critical systems mostly follow the principle of simplicity, thereby ensuring that they expose predictable behavior and low risk of failures. As a consequence, they lack a lot of features present in general-purpose systems like Enterprise Resource Planing (ERP) for example. On the other hand, resource limitations such as the absence of the virtual memory, file system, and others prevent these systems to have a maintainable runtime model, i.e. a model that would allow for changing only relevant dependencies between components, instead of requiring to re-build the whole system for each change request. However, because of the exponential grow of software-implemented functions in safety-related domains, new models for software maintenance are desirable [Butz 2007].

Current approaches mainly address the problems of maintaining software by employing standardized, modular architectures and automated support for software development process. In the automotive domain, for instance, the reuse issue has been addressed by the introduction of a new development methodology and component-based architecture called AUTOSAR [Kindel and Friedrich 2009]. This approach enabled to have a clear separation between infrastructure software and applications. Further, it has created a baseline for involved development organizations that independently have to maintain their parts of software (e.g. infrastructure software is typically delivered as a product to the vehicle company). Unfortunately, the application software, although component-based, is strongly coupled and has no runtime support for maintaining dependencies. Similar concepts for software architecture are present in the industrial automation sector, and they mostly rely on component frameworks based on IEC61131 and IEC61499 standards. Some reference implementations that follow these standards allow for changing dependencies at runtime (i.e. some Programmable Logic Controllers, PLCs), but in this paper we address a more general class of embedded systems which have the Unix-like system model and software implemented in C/C++ programming languages.

To narrow our problem statement, let us consider the component-based system depicted in Fig. 1. It shows the simplified form of the vehicles' anti-lock braking system, realized in AUTOSAR. As mentioned, the software system is separated into the infrastructure (i.e. runtime environment, the RTE) and the application. In the example above, three software components form the execution chain from the brake sensor to the brake actuator. From the viewpoint of a vehicle company (i.e. application developer), it is important to be able to deploy and to maintain different kinds of applications on top of the RTE. Based on a high-level application description, the application developer automatically generates a part of the RTE responsible for inter-component communication, thus conforming to AUTOSAR methodology. The problem is here that explicit dependencies between components are introduced. Namely, the component `BrakePedalSensor` routes its values to `BrakePedalController` using the RTE API `Rte_Write_PPosition_PData` and becomes dependent on it, since this part of RTE is generated based on naming conventions used to model the application [AUTOSAR 2011]. The last part of the API corresponds to interface and data names of the sending component. Because of the explicit dependencies between software components and the infrastructure software, the following limitations can be observed:

—Design-time software maintenance: the software application can be assembled not later than at design-time. For instance, in order to extend the `BrakePedalController` to get values for the wheel speed, the RTE has to be newly generated. This requires a high maintenance effort for devices in operation, since the whole software stack has to be re-built.

—Dependent development: the application development is not really independent from the infrastructure software. In fact, the application developer has to include the RTE in its development process. Further, in order to track different versions of the application software consistently, there is a need for the RTE configuration and version management on the application developer side.

—Dependent safety certification: modifications of the application software directly imply modifications of the RTE. Although safety is a system property, with clearly separated infrastructure and application software, various combinations to ensure safety would be possible. One of the variants would be, for instance, certified hardware

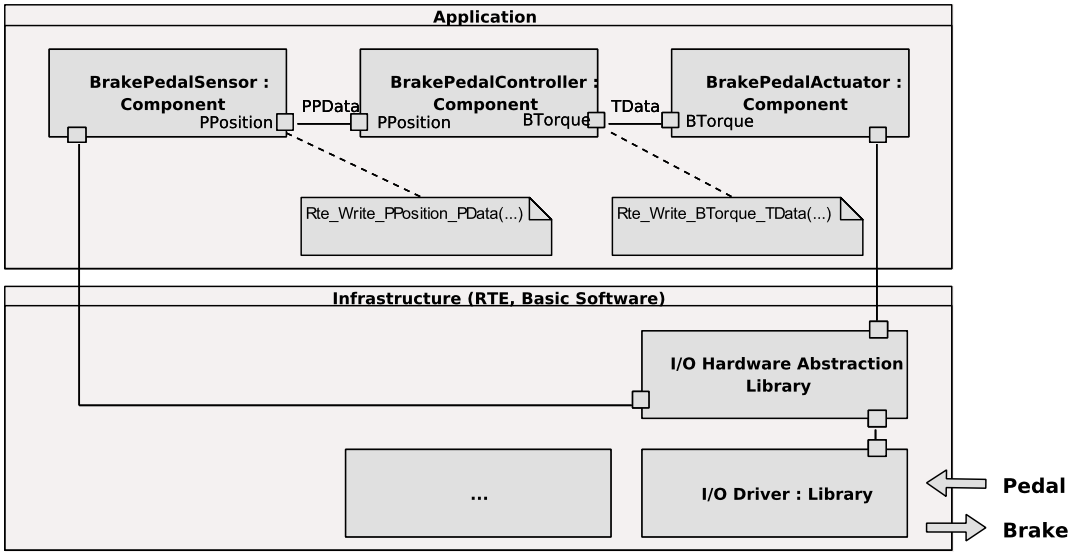


Fig. 1. Motivation example: application software for anti-lock braking system, adopted from [Piper et al. 2012]

and RTE provided to the application developer. In the current framework, however, it is only possible to certify the whole software system (i.e. to evidence that it will operate correctly with certain failure rate).

The issues listed above are common also for other customized component frameworks for (safety-critical) embedded systems such as SOFA-HI or SaveCCM [Hansson et al. 2004;Hosek et al. 2010]. To overcome these issues, the software architecture has to be systematically designed to manage dependencies between software components at runtime. Further, there is a need for a runtime mechanism which would allow to load software components when needed (i.e. dynamic linker). These two features might look completely independent from each other, but our experience in realizing and adopting the dynamic linker for component-based and safety-critical systems showed that this is not the case. Thus, the dynamic linker influences the architecture and the use of available mechanisms to realize dependencies between software components. Therefore, we handle it as part of the overall contribution, which we summarize as follows: we present the IoC container for component-based and safety-critical embedded systems. In this context, we show how to cope with stringent safety requirements while having dynamic link support and, on the other side, we also show how to realize the IoC using reduced set of features available in these domains (e.g. single address space, function pointers as flexible interaction mechanism, etc.). We further map these features to some relevant design patterns for object-oriented systems and show how the composition of these patterns forms the IoC container. The IoC structured in such a way can be practical for software developers who have to consider safety requirements when implementing their runtime mechanism for software maintenance.

The remainder of this paper is structured as follows: Sec. 2 briefly introduces the problem of using dynamic linking mechanism in the safety domain. Further, it outlines the IoC technique conceptually. In Sec. 3 the proposed IoC container is described, including the patterns that constitute it. Sec. 4 describes a case study which demonstrates the application of the IoC container (in further text, the framework). Sec. 5 describes related work and Sec. 6 gives concluding remarks.

2. BACKGROUND

As previously mentioned, dynamic linking is required when integrating out-of-the-box components in the application. In this section, we outline existing models for linking binary code and briefly document their conformance to the safety domain. At the end of the section, we introduce the IoC principle.

2.1 Dynamic Linking vs. Safety

The linking process combines object files in order to produce an application binary (executable) or library. Object files, in our context, are software components (e.g. compiled source code of the `BrakePedalController`). For Unix-like systems, these object files are represented in the Executable and Linkable Format (ELF), which describes their memory layout [Levine 1999]. The information behind the ELF is essential when linking independently created object files or when loading them at runtime, since it contains references to required libraries in form of symbolic names (i.e. symbols). For instance, if the `BrakePedalController` requires the sinus function from the `cmath` library, the symbol entry for this request will be placed into the ELF at compile-time. These entries are known as relocations and they instruct the linker how to find the appropriate symbols. The linking process, therefore, has to fix these relocations. Depending on the time when the linking is performed, there is a difference between static and dynamic linking. In the earlier linking model, the application is monolithic, and cannot be modified anymore. This is the most commonly used variant for resource-constrained systems.

Let us now consider the scenario where the RTE has to load the `BrakePedalController` dynamically. In this case, the object file of the `BrakePedalController` has the symbol references to the RTE libraries, which are unresolved until the begin of the linking process. Unfortunately, this is a problem from the safety aspect, because the symbol references between dependent binaries are not known until load-time or runtime, and therefore are not predictable (i.e. the absolute address of the required library symbol cannot be reproduced at design-time). Because it is difficult to ensure the correctness of the symbols resolved on such a way, the dynamic link model is often omitted in the safety domain [Kindel and Friedrich 2009].

To overcome the mentioned problem of dynamic linking, relocations have to be eliminated early at design-time. One way to do this is to access the required symbols using a so called indirection table. This is a common technique for patching software in Dynamic Software Updating (DSU) systems [Hicks and Nettles 2005]. Concretely for the example in Fig. 1, the `BrakePedalController` has to get the reference to that table in order to consume the required symbol located on RTE. How this table is realized and how components access it, is an architectural issue and is handled in Sec. 3. An additional prerequisite to avoid relocations is to have components implemented in plain `C` language (refer to [Kajtazovic et al. 2013] for more details).

2.2 Inversion of Control

The first concepts of the IoC principle were introduced in the late eighties when the software reuse has become an important research topic [Johnson and Foote 1988]. Many object-oriented constructs like polymorphism, inheritance and frameworks were used in that time as a main driver to reuse recurring functionality. Because of benefits gained from such a reuse, the IoC is nowadays a widely applied principle in many object-oriented and component-based containers.

Following this principle, loose coupling is ensured by inverting the responsibilities that would introduce dependencies between dependent components or introduce poor control over their lifecycle and configurations. According to the IoC principle of the PicoContainer, not only wirings between components are managed by some central entity (i.e. this instance of the IoC is known as dependency injection [Fowler 2004]), but also their creation and configuration [PicoContainer 2013]. Fig. 2 illustrates the IoC principle. The component `Dependent` as a client has to consume services of the component it depends on, i.e. the `Dependency`. In the worst case, the `Dependent` would take control over the lifecycle and the configuration of `Dependency` and thus increase the coupling between them. Instead, the responsibility of the creation and the configuration of the `Dependency`, and injection of its reference into the `Dependent` as well, are moved to the object `Injector`, thereby fully decoupling

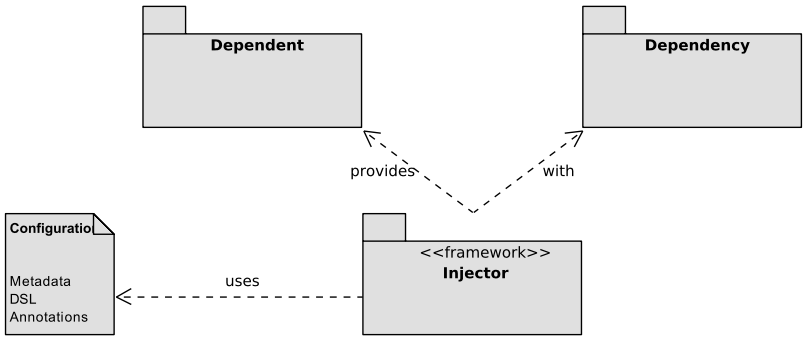


Fig. 2. Inversion of control principle, adopted from [Prasanna 2009]

both inter-related components. How the `Injector` realizes this, depends on implementation, features provided by the system and the used programming language. Therefore, IoC relies on different idioms. In most Java-based IoC containers, for instance, the reference to `Dependency` is provided to the component `Dependent` using its class constructor or setter methods (i.e. known as constructor and setter injection idioms [Prasanna 2009]). For systems implemented in C++, these types of injection are more difficult to realize, because of lacking relevant language features such as reflection mechanisms and annotations. Therefore, just experimental but no mature C++ IoC containers are currently present.

Related to the example shown in Fig. 1, the `Injector` is the infrastructure software (i.e. RTE), and the injected parts, i.e. `Dependent` and `Dependency` are software components that build the application software. However, the AUTOSAR framework does not follow the IoC principle completely, and Fig. 2 is just an idealized form of it.

In our framework, we have to deal with components realized in the `C` language and an application constructed following the component-based paradigm [Crnkovic 2002]. Considering components as black-box entities with only visible interfaces, other injection strategies for assembling applications are required than those used for object-oriented IoC containers. In this case, the application developer is not aware of the component internals such as the source code or objects while establishing dependencies between components. Thus, an IoC container, which is feasible to handle injections based on only visible aspects of components, is required here.

3. PROPOSED INVERSION OF CONTROL CONTAINER

In this section, we describe the architecture of our framework and show the patterns applied to realize the inversion of control.

3.1 Overview and Scope

Prior to the framework description, we first briefly outline some relevant characteristics of the target software system:

—Component-based architecture: the component-based paradigm has been identified as the best architectural solution to support black-box software reuse. As in introduced frameworks for (safety-critical) embedded systems, we apply it to separate the infrastructure from applications.

—Event-driven execution semantic: it is common for embedded control that applications are driven by occurrence of events (e.g. brake pedal event caught by the `BrakePedalSensor` component). Therefore, component activations are managed based on events produced by the infrastructure or neighboring components.

—Dynamic memory management: to allow load-time or runtime reconfiguration of the applications, the framework uses the OS-support for dynamic allocation of components.

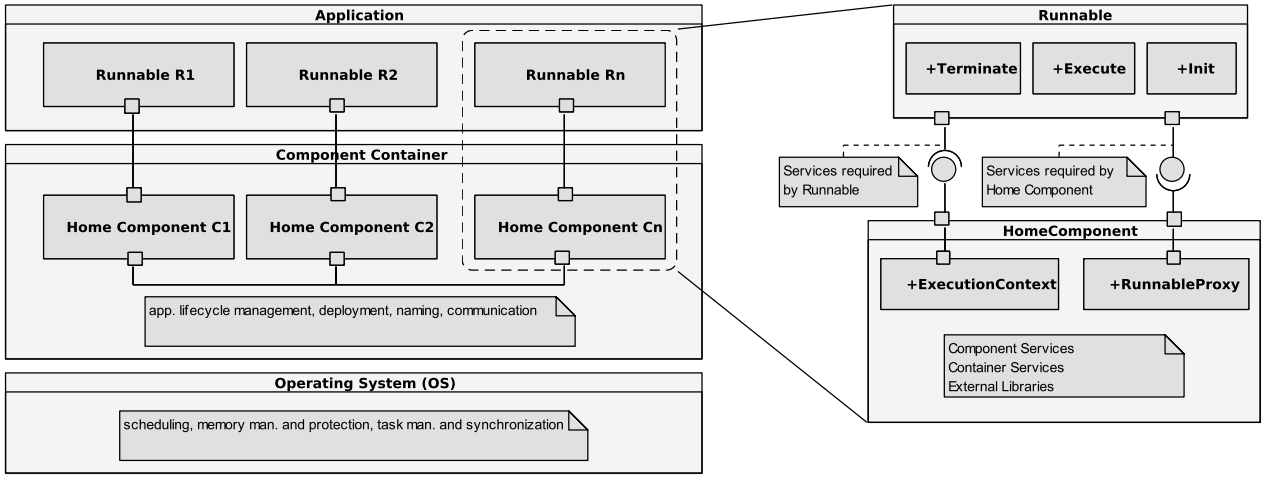


Fig. 3. Software architecture of the proposed component framework

—Dynamic linker: in order to load out-of-the-box components, the framework uses a linker mechanism as described in Sec. 2, i.e. it only loads component object files without having to relocate any symbols.

The main goal of the framework is to allow to dynamically load and reconfigure the application software. As mentioned, the application on top of the infrastructure software is constructed by wiring components, which form parts of the application logic. Therefore, loading the application implies loading and configuring individual components, and finally wiring them together. These are well-known issues in the domain of distributed object-oriented and component-based applications, and they have been soundly handled in the last decade. For instance, in the fourth part of the POSA, a pattern language which addresses the holistic view of cross-domain distributed computing has been presented [Buschmann et al. 2007]. We realize the inversion of control based on some of these patterns, in particular patterns for resource management and object interaction, and show how they complement the applied component-based paradigm.

3.2 Architecture

Based on the linking models from Sec. 2.1, we separate the architecture of the framework into static and dynamic parts. The former is the infrastructure software, statically linked with the OS. It consists of all libraries and services needed by applications, which represent another part of the framework. In order to allow to configure applications dynamically, each of the out-of-the-box binaries that constitute the application logic owns its local container within the infrastructure, the so called *home component*. Home components reside within the static part of the framework, and are dynamically allocated by it. Fig. 3 shows the framework architecture. This way of interfacing with the framework can be compared with the concept of *home interface* in the CORBA component model (CCM) [OMG 2006].

The home component together with the out-of-the-box binary, called *runnable*, form a software component in the proposed framework. The reason for such a separation is to shift the configuration, lifecycle and dependency management of runnables to the infrastructure, thus making the scope of components more flexible. In contrast to the AUTOSAR framework described in Sec. 1, a component has only one object, because of easier configuration of component's internals. However, the same effect as in AUTOSAR can be achieved when grouping home components.

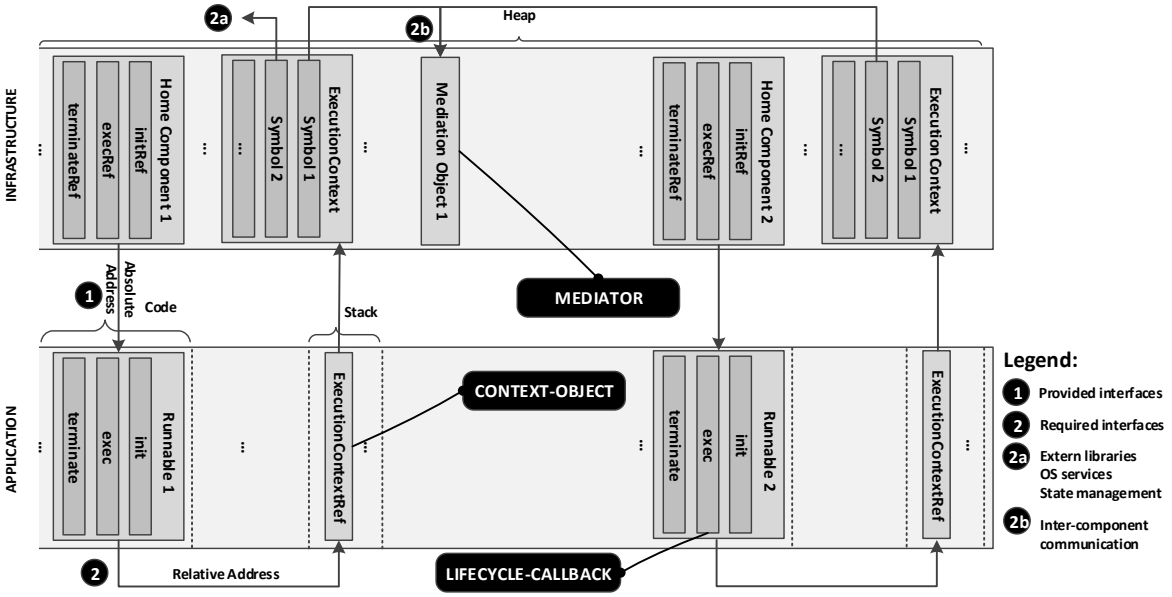


Fig. 4. Runtime memory layout of the framework and applied patterns (COMPONENT-CONFIGURATOR is out of the context here)

The right part of Fig. 3 shows the more detailed architecture of the software component with the focus on interaction between runnables and home components. From the viewpoint of a runnable, there are two types of interfaces:

—Provided interfaces: interfaces that provide access to the application logic from the infrastructure software.

—Required interfaces: interfaces required by a runnable to perform their functions. These include external libraries, infrastructure services for inter-component communication, OS services for resource management and component services for local resource management (e.g. state allocation).

3.3 Interfaces for Dynamic Linking

The interfaces described above are dynamically bound at deployment time of runnables. As discussed in Sec. 2.1, runnables use indirection tables to access required symbols. These tables corresponds to required interfaces, and are realized as structures of function and data pointers, placed on the stack. Accessing the tables from the stack is beneficial for systems lacking virtual memory, because in this way the position-independence of the runnables can be achieved, i.e. runnables can be placed anywhere in the address space, thus allowing to host multiple applications on top of the infrastructure. The reason for position-independence is that communication using the stack is based on relative symbol addressing, i.e. address of the sinus function within the `libc` library for example is estimated at runtime. Fig. 4 shows the runtime memory layout which describes the interaction between runnables and the rest of the framework.

Another direction, i.e. provided interfaces, are absolute symbol references, and are automatically estimated in the deployment process of runnables (see usage of COMPONENT-CONFIGURATOR pattern in Sec. 3.6).

3.4 Application Context

Besides symbols required for dynamic linking, runnables have to be in position to allocate their local state, to communicate with other runnables or with the infrastructure and to take the responsibility of the component's lifecycle on their own. Because all these services are not present in the binary of runnables, but inside of home

components, runnables have to access them using the indirection table from the stack. Thus, part of the indirection table used for these services corresponds to the execution context of a runnable (see Fig. 4). For each execution of the runnables, the infrastructure places that context on the stack. This sort of interaction between the infrastructure and runnables is based on the CONTEXT OBJECT pattern [Buschmann et al. 2007]. The aim is to share the information between both parties, with possibly low coupling. In the original form of the pattern described in POSA, the intent is to allow clients to propagate their execution context such as the session state to the service. In the proposed framework, not only services of the infrastructure are contained in the execution context, but also the state of related runnable, since runnables cannot carry any data.

3.5 Loose Coupling Through Mediation

Using the execution context to propagate the state to runnables is just one step towards loose coupling. It allows runnables to interface with the infrastructure, but not to use dependencies to other components.

The interaction between components is based on events, which may contain data or just signals to trigger the execution. Considering the example in Fig. 1, the component `BrakePedalController` routes the computed values for the actuator to the `BrakePedalActuator` component by signaling those values with the `Rte_Write_PTorque_TData` call. From the viewpoint of the `BrakePedalActuator` component, this is an event indicating that data are available and execution can start. Similar to this principle, the interaction in the proposed framework is realized, but with the exception that components are fully decoupled.

In order to trigger the execution of another component, a runnable has to use an API call provided within the execution context, described in the previous section. Further, to route that call to the target component, the framework instantiates an intermediate object, i.e. mediation object, that holds the event values (see Mediation Object 1 in Fig. 4). In a similar way, the receiving component consumes the event values by scanning all its mediation objects using the execution context. This principle is based on the MEDIATOR design pattern, which allows to decouple dependent components by moving their dependencies to the external mediation object [Buschmann et al. 2007]. For each dependency created between two components, one such mediation object is created.

3.6 Application Configuration

By applying the CONTEXT OBJECT and MEDIATOR patterns, the desired inversion of control with minimal coupling was achieved. Now, it is the task of the framework to load and configure components according to the design described above. This includes loading runnable binaries (see Sec. 2.1), allocating their home components, initializing their execution context and allocating mediation objects. The latter corresponds to the wiring between components, and is declaratively specified in an application assembly file (such as XML, see Fig. 2). Similar to this description, information about a single component such as required interfaces, state members, etc. are also declaratively specified and used to configure a component. Each of the created instances has a flexible lifetime, so that the application can be reconfigured at runtime, if required. Here, we apply the COMPONENT-CONFIGURATOR pattern in order to allow for flexible configuration of applications [Buschmann et al. 2007].

3.7 Application Lifecycle and Scope

Since there are no standard ways for managing component's creation and destruction as it is done with objects, components have to implement some standard interfaces which the infrastructure can use to manage their lifecycle. These are particularly initialization and finalization routines that have to be explicitly handled by each of the runnables. However, the creation and destruction of mediation objects is not in the responsibility of runnables, but the initialization of the state and values within the mediator objects. Therefore, runnables provide interfaces as documented in the LIFECYCLE-CALLBACK pattern [Buschmann et al. 2007]. Concretely, they only implement initialization, finalization and activation methods, i.e. `init`, `terminate` and `execute` respectively.

4. EXAMPLE: AUTOMOTIVE BRAKE SYSTEM USING PROPOSED IOC

In this section, we revise the example from Sec. 1 and show how dependencies between components are now handled by the introduced IoC container. It is obvious that we are now able to interconnect software components without having strong coupling links like it is the case with the initial example from Sec. 1. As a consequence, software components can be seamlessly exchanged.

4.1 Overview and Scope

In this example, we just consider the communication between `BrakePedalSensor` and `BrakePedalController` from the viewpoint of the `BrakePedalController`. This component has two interfaces, i.e. `PPosition` and `BTorque`, each of them of the type `float`. To compute the value of the `BTorque`, the component maintains a single state member `prevPosition` of the same type.

4.2 Realization

The configuration of each of the components including the syntax and semantics of their interfaces is specified in an application assembly file. In a similar way, the composition of the components (i.e. the application) is provided in another description file. For each component description, the component developer provides the lifecycle routines and implements the application logic. The following code listing shows the exemplary initialization routine of the `BrakePedalController` runnable.

```
#define PREVIOUS_BRAKE_POSITION 1
...
int init( ExecutionContext* ctx )
{
        ctx->setSFValue( PREVIOUS_BRAKE_POSITION, 0.0F );
        return SUCCESS;
}
```

In this example, the runnable `BrakePedalController` just initializes its state member to `0`. Using the context, the runnable gets the reference to the state member `prevPosition`, located in the home component. Considering the runtime memory layout from Fig. 4, this action takes the paths marked as 2 and 2a. In order to find the requested members, the infrastructure software uses the identifiers specified in the application assembly file. Therefore, both the code and the specification within the assembly file have to be consistent.

In a similar way, the runnable interacts with the execution context when accessing interfaces. The following code listing shows an excerpt of the application logic of the `BrakePedalController` runnable.

```
#define BRAKE_POSITION          2
#define BRAKE_TORQUE            3
...
int execute( ExecutionContext* ctx )
{
        float prevPos = ctx->getSFValue( PREVIOUS_BRAKE_POSITION );
        float currentPos = ctx->getIFValue( BRAKE_POSITION );

        float torqueOut = 0.0F;

        ... // estimate torque value

        ctx->setIFValue( BRAKE_TORQUE, torqueOut );

        return SUCCESS;
}
```

In the listing above, the `prevPos` state and the `currentPos` interface are used to compute the torque value and to route it to the output interface `BTorque`. All these values are accessed using the standard API of the execution context.

4.3 Observations

For the interaction between `BrakePedalSensor` and `BrakePedalController`, one mediation object was created. However, none of the runnables are aware of it. They make the reference to their interfaces and state members using identifiers defined in a component description, while assembly description is only known to the infrastructure software. The components, therefore, just know their local structure and configuration, whereby the infrastructure is aware of the standard interfaces (API) of the execution context, instantiated mediator objects and home components.

In contrast to the example from Fig. 1, it is possible to maintain dependencies, configurations and the lifecycle of components at runtime. However, the main limitation in this framework is that also the symbols used for dynamic linking have to be accessed using the execution context. For instance, to use any function from `cmath` library, it is necessary to first define it in the execution context. For large number of symbols, this may introduce a significant overhead when implementing the execution context.

5. RELATED WORK

Most component frameworks in safety the domain have no support for dynamic linking and therefore are not designed for maintaining software at runtime. On the other hand, existing C++ IoC containers such as Qt IOC Container[2] or autumnframework[3], which rely on macros and templates, are pure object-oriented IOC containers and are therefore not compatible with the introduced concept for dynamic linking. In this section, we outline some similar works which indirectly address the inversion of control.

Fractal THINK is a component-based framework implemented in the `C` language [Polakovic et al. 2006]. It offers a built-in reflection mechanism and a flexible inversion of control concept. Each of the components gets so-called controllers which manage the component's lifecycle and bindings. These controllers are infrastructural elements and they ensure the late binding of components by managing function and data pointers used for binding at runtime. This framework, unfortunately, uses the standard dynamic linking mechanism for deploying the components, i.e. it performs relocation of the symbols when loading the component binaries.

An approach for dynamic linking based on indirection tables was presented by Han et. al [Han et al. 2005]. It extends the TinyOS operating system for wireless sensor networks (WSN) to load binaries dynamically. In contrast to FRACTAL , the dynamic linking mechanism is based on relocation-free loading of binaries. These binaries are, similar to our concept of linking, position-independent modules and require no relocations. The framework is, however, not component-based and dependencies between modules are managed by the central indirection table within the OS kernel. The inversion of control is based on the indirection table, which is placed on a fixed address within the kernel and therefore is known to all binaries. Since the framework is not component-based, binaries that are dynamically loaded have no standard interfaces and the identification of the methods of binaries is based on names or profiles of these methods. Therefore, safety-relevant features such as type safety checks and testing of the interfaces for dynamic linking are difficult to perform.

6. CONCLUSION

It is common practice for safety-critical embedded systems that strategies for software reuse and maintenance are mainly planned and managed at design-time. Because systems in this domain have to undergo a rigorous software development process, any behavior which is difficult to predict is usually omitted. This is also the case

[2]Qt IOC Container Homepage - http://qtioccontainer.sourceforge.net/
[3]autumnframework Homepage - http://code.google.com/p/autumnframework/

with dynamic linking support, which is the main driver for reconfiguring software at runtime. Because of a lack of such a support, current frameworks in this domain are not designed for runtime software maintenance.

In this paper, we presented a lightweight variant of the framework with a simple dynamic link support and the architectural means to allow for maintaining software at runtime. We applied here the inversion of control principle with the aim to ensure loose coupling between components that constitute the software application. Since the framework is a component-based system (i.e. as many other frameworks in this domain), we have shown that inversion of control can be effectively realized using patterns for object interaction and resource management, which are primarily used in general purpose systems. From the viewpoint of dependency injection, it is important to have the infrastructure decoupled from components and components decoupled from each other. This is ensured by the CONTEXT-OBJECT and MEDIATOR pattern respectively. On the other hand, the runtime support for lifecycle and configuration management is realized according to LIFECYCLE-CALLBACK and COMPONENT-CONFIGURATOR patterns. Applied patterns provide a holistic view of the inversion of control in the proposed component-based framework.

7. ACKNOWLEDGMENTS

We would like to thank our shepherd Eduardo Fernandez for being ready to analyze this paper and to improve it by providing very constructive and helpful feedback.

REFERENCES

AUTOSAR. 2011. Specification RTE. Tech. rep., AUTOSAR Consortium, Homepage: `http://www.autosar.org`.

BUSCHMANN, F., HENNEY, K., AND SCHMIDT, D. C. 2007. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley.

BUTZ, H. 2007. Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. White paper, Department of Avionic Systems at Airbus Deutschland GmbH Kreetslag 10, D-21129 Hamburg, Germany.

CRNKOVIC, I. 2002. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA.

FOWLER, M. 2004. Inversion of Control Containers and the Dependency Injection pattern. Tech. rep., Homepage: `http://martinfowler.com`.

HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. 2005. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*. MobiSys '05. ACM, New York, NY, USA, 163–176.

HANSSON, H., AKERHOLM, M., CRNKOVIC, I., AND TORNGREN, M. 2004. Saveccm - a component model for safety-critical real-time systems. In *Proceedings of the 30th EUROMICRO Conference*. EUROMICRO '04. IEEE Computer Society, Washington, DC, USA, 627–635.

HICKS, M. AND NETTLES, S. 2005. Dynamic software updating. *ACM Trans. Program. Lang. Syst. 27,* 6, 1049–1096.

HOSEK, P., TOMAS, P., MICHAL, M., PETR, H., AND TOMAS, B. 2010. Supporting real-time Features in a Hierarchical Component System. Tech. rep., Dep. of Distributed and Dependable Systems, Charles University in Prague.

JOHNSON, R. E. AND FOOTE, B. 1988. Designing Reusable Classes. *Object-Oriented Programming 1,* 2.

KAJTAZOVIC, N., PRESCHERN, C., AND KREINER, C. 2013. A Component-based Dynamic Link Support for Safety-critical Embedded Systems. In *Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. ECBS 2013. IEEE.

KINDEL, O. AND FRIEDRICH, M. 2009. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt Verlag; Auflage: 1 (8. Juni 2009).

LEVINE, J. R. 1999. *Linkers and Loaders*. Morgan Kaufmann; 1 edition (October 25, 1999).

OMG. 2006. Corba components - version 4.0, specification. Technical report, Object Management Group Inc., formal/06-04-01. April.

PICOCONTAINER. 2013. Picocontainer - ioc container. Homepage: `http://picocontainer.com/`.

PIPER, T., WINTER, S., MANNS, P., AND SURI, N. 2012. Instrumenting AUTOSAR for dependability assessment: A guidance framework. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DSN '12. IEEE Computer Society, Washington, DC, USA, 1–12.

POLAKOVIC, J., OZCAN, A., AND STEFANI, J.-B. 2006. Building Reconfigurable Component-Based OS with THINK. In *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*. 178 –185.

PRASANNA, D. R. 2009. *Dependency Injection - Design Patterns Using Spring and Guice*. Manning Publications.

SPRINGSOURCE. 2013. Spring framework. Homepage: `http://www.springsource.org`.

APPENDIX

Table I. List of applied patterns

Pattern	Problem	Solution
CONTEXT-OBJECT [Buschmann et al. 2007]	Interrelated components usually share common data by accessing global variables or some central SINGLETON. The coupling between components on this way is very strong, since it is hard-coded and cannot be changed at runtime.	The component, which requires the services (i.e. the client), has to provide its environment to the service component using the context object. By doing so, the service component can access the environment data of the client, without being dependent on it. This pattern is a typical example for the inversion of control.
MEDIATOR [Buschmann et al. 2007]	It is often required that communicating components are fully decoupled. An example is component-based application which can be easily reconfigured at runtime. Currently, using the global variables or SINGLETON, or even CONTEXT-OBJECT, is not practical, since these options make the components explicitly coupled.	For each connection between components, there is an individually instantiated object which maps both components (i.e. the mediator object). By doing so, none of components is responsible for managing dependencies and routing of data is just delegated to the mediator object.
COMPONENT-CONFIGURATOR [Buschmann et al. 2007]	Although many application models used in frameworks for safety-critical systems are component-based, their runtime structure is monolithic. It is therefore difficult to maintain the lifecycle of components designed on such a way. The components have to be replaced at application runtime or load-time, e.g. due to changed environment, bug fixing or because of the improvement in their performance.	The components that constitute the application have to be designed as dynamically linked objects with well-defined interfaces that expose the functionality and configuration of the components. Based on this design, the component configurator can rely on generic mechanism for reconfiguring the application by deploying and re-deploying the components. Maintaining the application at runtime is therefore based on deploying and re-deploying the components.
LIFECYCLE-CALLBACK [Buschmann et al. 2007]	The components have access to system resources and the usage of these resources is typically component-specific. Further, typical creation and disposal of simple objects is for most components not possible (e.g. components may consist of several objects). Managing the lifecycle of such components from a central framework is difficult, particularly because the internals of components are not known (e.g. when and how they acquire which resources).	The components have to expose standard interfaces that would allow the framework to manage their lifecycle transparently. Each component is therefore obligated to implement these interfaces.

Towards Predictable Dynamic Linking for Safety-critical Component-based Systems

Nermin Kajtazovic, Christopher Preschern, Norbert Druml, Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
{nermin.kajtazovic, christopher.preschern, norbert.druml, christian.kreiner}@tugraz.at

I. Introduction and Motivation

Safety-critical systems have stringent requirements regarding systems quality. They drive the processes in which failures in their functions can lead to catastrophic consequences for operating environment and humans. Depending on the application domain, hardware and software quality of these systems is regulated by safety standards (e.g. ISO 26262 for automotive, generic IEC 61508 for industry, etc.). In order to get desired quality for software, a rigorous development process has to be conducted. It is therefore common for these domains to follow the principle of simplicity, thereby removing all unpredictable behaviors and making the system testable at all. This usually leads to typical monolithic system's structure which is difficult to maintain.

In parallel, due to continuous increase of software complexity (i.e. more software-implemented functions, more electronic devices in automobiles), there is a need to shorter development cycles and costs of change and maintenance. This resulted with the paradigm shift from monolithic to modular and standardized component-based architectures. Exemplary specifications are AUTOSAR for automotive and IEC 61131/61499 for industrial automation. Using new approaches enabled a clear separation between common and specific (custom) software functions and corresponding roles (i.e. such as suppliers and manufacturers in automotive). In this way, the system design was utilized to allow for constructing software by reusing existing artifacts, such as pre-fabricated binaries or source code. However, the runtime system created in such a way is still monolithic and maintenance operations (upgrades) cannot be performed without re-building it. To this end, some approaches go one step further and employ the adaptation in software architectures [1] [2]. In case of failures or degradation of quality, they are able to switch between different system configurations at runtime in order to reach again *normal* behavior. Binding static configurations at design-time allows to verify the adaptive behavior prior to deployment of the system, which is promising when it comes to the safety certification (i.e. all possible outcomes of the adaptation can be predicted).

The next step towards more effective way of maintaining safety software would be the ability for late binding of out-of-the-box software artifacts (in further text, components). This would allow to have a repository of reusable components that can be used for software construction and maintenance of the system in mission, like it is a long practice in many enterprise applications. However, the standard binding mechanisms (i.e. dynamic linkers) are not directly applicable to safety domains, since the results of the bindings they produce are not predictable. In this article, we give a brief overview of our approach for dynamic linking and its ongoing work. Similar to mentioned adaptation approaches, we focus on predictability of the underlying mechanism.

II. Approach for Dynamic Linking of Safety Software

The system we describe in this article comprises a component-based architecture, conceptually similar to AUTOSAR [3]. The components are on the top of this architecture and represent the application software. The underlying infrastructure software is a Component Container that manages the components. Both parts are running on a Unix-like Real-time Operating System (RTOS). Regarding software maintenance, the system allows to dynamically link the applications and required libraries.

A. Dynamic Linking and Safety

In order to perform certain application function, the components may require external functionality, such as mathematical libraries, communicate with other components, use resources from RTOS, etc. In binary form, these needs are expressed in terms of *relocations*. The process of dynamic linking in this context has to read this information in order to find desired functions or data (in further text, symbols) within the remaining system and finally to link the both ends. Within this process, two problems arise: (1) final symbol location is known not before runtime and (2) component binary has to be modified after symbols are found. These are the properties of the linking process which are difficult to verify at design-time.

B. Ensuring Predictability of Dynamic Linking

Having binaries without relocations would not introduce the problems mentioned above. However, they always arise as long as the binaries are accessing the external symbols directly (e.g. by calling functions). To still avoid the relocations, we use the *indirection tables* as interfaces between component binaries and the Component Container. The indirection tables are one of the alternatives to realize dynamic linking [4]. Based on these tables, the components can express their needs (i.e. required interfaces) and access the symbols of the Component Container without producing any relocations. On the other side, the Component Container builds the list of symbols which it uses to manage the components' lifecycle (initialization,

activation and finalization, i.e. the provided interfaces). After both these interfaces are instantiated so that components get their indirection tables and the Component Container gets the components' symbols, the process of dynamic linking is completed.

The problem with the concept above is that the components have to know the location of the indirection tables. In order to avoid having the components with hard-coded location, we provide the indirection table on the stack of components' functions each time they are active. In this way the components can access their required interfaces using relative addresses (relative to the frame pointer for example). Using relative addressing mode has the following advantages: (1) components are position-independent so that they can be deployed anywhere in the address space and (2) the infrastructure software can evolve independently from the components. In similar way, we defined a concept for libraries which evolve and have to be integrated into an existing infrastructure software. Those libraries are designed as components, and can be accessed by other components using the mechanisms for inter-component communication.

Another problem is that the usage of the indirection tables can be tedious in some cases. For example, in processor architectures which have no hardware support for floating point operations, the request to indirection tables has to be made for each such an operation. To avoid this, we allow to link the libraries with the component binaries directly, including the inspection of the correctness for that linking (see Sec. II-C). In this way, the symbols which are intensively used by the component code are injected within its binary so that no interaction with the infrastructure software is required. However, this alternative is only possible for processor architectures that allow relative addressing mode in instructions (such as relative branches in ARM for example).

C. Verification of Dynamic Linking

To verify the correctness of the proposed dynamic linking, we have to verify that dynamically established provided and required component interfaces are correct and that a component binary is conforming to the specification of our component model (refer to [5] for more details).

1) Verifying Bindings: Required interfaces are provided to a component using its stack. Further, they are bundled within an indirection table which is statically linked with the Component Container. Since this table contains the symbols to the infrastructure software which are resolved statically, the correctness of these interfaces can therefore be easily verified at design-time. Similar to this, the verification of the provided interfaces can be conducted.

2) Verifying Binaries: It is very easy to produce the component binaries that are not conforming to the target component model, i.e. binaries that are position-dependent and contain absolute relocations. In such cases, the components cannot function, because their static memory layout does not conform to the runtime memory layout. Therefore, we have to identify (1) whether a component binary does access the symbols using the absolute addressing mode and (2) whether the required libraries are conforming to the target component model too. We perform both verification steps by analyzing the binaries in the ELF (Executable and Linkable Format, [6]) format as follows:

- Verifying Components: a component is valid only if it does not use the absolute addressing mode locally (e.g. access to local data, local functions, etc.). This can be easily verified using the ELF-attributes related to relocation information which are stored within a binary.
- Verifying Libraries: to verify required libraries we construct a directed graph, in which nodes represent the binaries and directed edges are symbols connecting the binaries. Starting from the root node, we verify each binary according to the same rules we apply for the components.

III. Conclusion and Future Work

Safety-critical systems have to follow a rigorous development process in order to attain optimal quality. On the other side, due to growing complexity of software there is a push towards cost reduction in development cycles and maintenance. In this article, we have introduced our approach for dynamic linking, as an important step towards more effective maintaining of safety software. The essential in this process is that the results of the linking can be predicted. Instead of modifying the relocations in component binaries, we use the relative addressing mode and indirection tables as a link between the components and remaining system. This allows us to verify all dynamic behaviors of the linking mechanism at design-time.

As stated, we are supporting linking external libraries either with components directly or deploying them as components. Depending on verification of the linking or performance issues (i.e. direct linking is faster than indirection), the developer can choose one of the options. Currently, we are working on automated allocation for used libraries. The intent is to accelerate the component development process by suggesting optimal and feasible configurations. Further, based on required interfaces of the components and of the dependency graph we conducted in the verification process, the aim is to generate the test cases which would allow to easily conduct the performance of the configurations.

References

[1] R. Adler, I. Schaefer, M. Trapp, and A. Poetzsch-Heffter, "Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 20:1–20:39, Jan. 2011.

[2] G. Montano, "Dynamic reconfiguration of safety-critical systems: Automation and human involvement," *PhD Thesis*, 2011.

[3] O. Kindel and M. Friedrich, *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt Verlag, 2009.

[4] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005. [Online]. Available: http://doi.acm.org/10.1145/1108970.1108971

[5] N. Kajtazovic, C. Preschern, and C. Kreiner, "A Component-based Dynamic Link Support for Safety-critical Embedded Systems," in *Proceedings of the IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ser. ECBS 2013. IEEE, 2013.

[6] J. R. Levine, *Linkers and Loaders*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

Constraint-based Verification of Compositions in Safety-critical Component-based Systems

Nermin Kajtazovic, Christopher Preschern, Andrea Höller and Christian Kreiner

Abstract Component-based Software Engineering (CBSE) is currently a key paradigm used for building safety-critical systems. Because these systems have to undergo a rigorous development and qualification process, one of the main challenges of introducing CBSE in this area is to ensure the integrity of the overall system after building it from reusable components. Many (formal) approaches for verification of compositions have been proposed, and they generally focus on behavioural integrity of components and their data semantics. An important aspect of this verification is a trade-off between scalability and completeness.

In this paper, we present a novel approach for verification of compositions for safety-critical systems, based on data semantics of components. We describe the composition and underlying safety-related properties of components as a Constraint Satisfaction Problem (CSP) and perform the verification by solving that problem. We show that CSP can be successfully applied for verification of compositions for many types of properties. In our experimental setup we also show how the proposed verification scales with regard to size of different system configurations.

Key words: component-based systems; safety-critical systems, compositional verification, constraint programming

Nermin Kajtazovic
Institute for Technical Informatics, Graz University of Technology, Austria
e-mail: nermin.kajtazovic@tugraz.at

Christopher Preschern
Institute for Technical Informatics, Graz University of Technology, Austria
e-mail: christopher.preschern@tugraz.at

Andrea Höller
Institute for Technical Informatics, Graz University of Technology, Austria
e-mail: andrea.hoeller@tugraz.at

Christian Kreiner
Institute for Technical Informatics, Graz University of Technology, Austria
e-mail: christian.kreiner@tugraz.at

1 Introduction

Safety-critical systems drive the technical processes in which failures can cause catastrophic consequences for humans and the operating environment. Automotive, railway and avionics are exemplary domains here, just to name few. In order to make these systems acceptably safe, their hardware/software engineering has to be rigorous and quality-assured.

Currently, rapid and continuous increase of system's complexity is one of the major challenges when engineering safety-critical systems. For instance, the avionics domain has seen an exponential growth of software-implemented functions in the last two decades (Butz (-)), and a similar development has also occurred in other domains with a focus on mass production (Kindel and Friedrich (2009)). In response, many domains have shifted towards using component-based paradigm (Crnkovic (2002)). The standards such as the automotive AUTOSAR and IEC 61131/61499 for industrial automation are examples of widely used component systems. This paradigm shift enabled the improvement in reuse and reduction of costs in development cycles. In some fields, the modularity of the system structure is utilized to distribute the development across different roles, in order to perform many engineering tasks in parallel (e.g. the automotive manufacturers are supplied by individually developed middleware and devices which can run their applications).

However, the new paradigm also introduced some new issues. One of the major challenges when applying CBSE is to ensure the integrity of the system after building it from reusable parts (components). The source of the problem is that components are often developed in the isolation, and the context in which they shall function is usually not considered in detail. In response, it is very difficult to localize potential faults when components are wired to form a composition – an integrated system (Gössler and Sifakis (2005)), even when using quality-assured components. The focus of the current research with regard to this problem is to enrich components with properties that characterize their correct behavior for particular context, and in this way to provide a basis for the design-time analysis or verification[1] of compositions (Clara Benac Earle et al (2013)).

This verification is also the subject of consideration in some current safety standards. For instance, the ISO 26262 standard defines the concept Safety Element out of Context (SEooC), which describes a hardware/software component with necessary information for reuse and integration into an existing system. Similarly, the Reusable Software Components concept has been developed for systems that have to follow the DO-178B standard for avionic software. These concepts both share the same kind of strategy for compositional verification: contract-based design. Each component expresses the assumptions under which it can guarantee to behave correctly. However, the definition of the specific contracts, component properties and validity criteria for the composition is left to the domain experts.

[1] In the remainder of this paper, we use the term *verification* for static, design-time verification (cf. static analysis (Tran (1999))).

From the viewpoint of the concrete and automated approaches for compositional verification and reasoning, many investigations have focused on behavioural integrity, i.e. they model the behaviour of the components and verify whether the composed behaviours are correctly synchronized (de Alfaro and Henzinger (2001)), (Basu et al (2011)). On the other side, compositions are often made based on data semantics shared between components (Benveniste et al (2012)). Here, the correct behaviour is characterized by describing valid data profiles on component interfaces. In both cases, many properties can be required to describe a single component and therefore scalability of the verification method is crucial here.

In this paper, we present a novel approach for verification of compositions based on the data semantics shared between components. We transform the modelled composition along with properties into a Constraint Satisfaction Problem (CSP), and perform the verification by solving that problem. To realize this, we provide the following contributions:

- We define a component-based system that allows modelling properties within a complete system hierarchy.
- We define a structural representation of our modelled component-based system as a CSP, which provides us a basis to verify the preservation of properties.
- We realize the process that conducts the transformation of the modelled component-based system into a CSP and its verification automatically.

The CSP is a way to define the decision and optimization problems in the context of Constraint Programming paradigm (CP) (Apt (2003)). Using this paradigm for our component-based system, many types of properties can be supported. Also, various parameters that influence the scalability of the verification can be controlled (used policy to search for solutions for example). In the end of paper, we discuss the feasibility of the approach with regard to its performance.

The remainder of this paper is organized as follows: Section 2 describes the problem statement more in detail and gives some important requirements with regard to modelling a system. In Section 3 the proposed verification method is described. Section 4 describes the experimental results. A brief overview of relevant related work is given in Section 5. Finally, concluding remarks are given in Section 6 .

2 Problem Statement

Properties are an important means to characterize functional and extra-functional aspects of components. Safety, timing and resource budgets are examples here, just to name few (Sentilles et al (2009)). Recently, they get more and more attention in the safety community, since efficient (an practical) reuse methods are crucial in order to reduce costs in development cycles and costs for certification of today's safety-critical systems (i.e. their extensive qualification process). In this section, we give an insight into the main challenges when using properties to verify composi-

tions, and based on these challenges, we outline the main objectives that we handle in this paper.

2.1 Motivating Example

In our work, we address properties that in general describe data semantics. To clarify this, let us consider now the example from Figure 1. The system in this figure

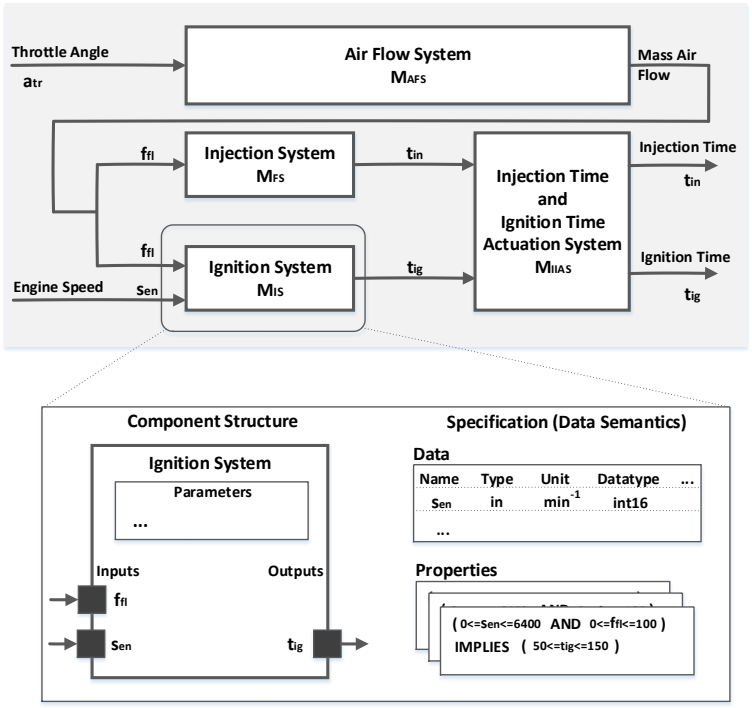


Fig. 1 Motivating example: a component-based system of automotive engine control function, adopted from (Frey (2010)) (top), and detailed view of the component Ignition System (structure and specification, bottom)

shows the composition of four components that form the automotive engine control application on a higher abstraction level. The basic function of this application is to decide when to activate the tasks of the fuel injection and ignition (Frey (2010)). To do this, the application takes the sensed values of the air flow volume, current speed and some parameters computed from the driver's pedal position. In a typical automotive development process[2], the system structure from figure is made based on stepwise decomposition of top-level requirements, having several intermediate

[2] Note that we do not limit our approach to automotive domain.

steps such as the functional and technical system architecture with several levels in the hierarchy. Let us assume now that involved components are already developed, eventually for the complete car product line, and are stored in some repository. Let us further assume that we have a top-level requirement with regard to the engine timing for particular car type, which states the following:

The minimal allowed time delay between the task of the fuel injection and ignition shall be greater than 40 ms.

The main contributors to this requirement are software components M_{AFS}, M_{FS}, M_{IS}, M_{IIAS}, and their execution platform (e.g. concrete mapping of components on real-time tasks, task configurations, and other). In order to satisfy this timing property, the developer has to analyze the specification for each component in order to find the influence of the component behaviour on that property. The example of such a specification is given in Figure 1, bottom. Here, the context for the component Ignition System is defined in terms of the syntax and semantics related to component inputs, outputs and parameters. With the properties shown below, the concrete behavior can be roughly described – in this example, for certain intervals of inputs, the component can guarantee that the output t_{ig} lies within the interval $[50, 150]$ (note that pseudo syntax is used here). When building compositions based on such properties, the developer has to consider their influence on the remaining, dependent components – in this case, it has to be decided whether the M_{IIAS} component can accept such values of the t_{ig} and what should components M_{FS} and M_{AFS} provide so that higher delay than $40ms$ between t_{ig} and t_{in} can be achieved. This can be very tedious and error prone task when doing it manually, because of the following reasons:

- Many components may be required to build a complete system, depending on their granularity. For example, current automotive systems comprise several hundreds of components, and many of them may depend on each other (Kindel and Friedrich (2009)).
- Some components that directly influence the safety-critical process are usually certified, i.e. developed according to rigorous rules from safety standards. Because of costs for such a certification, the practice is to develop components for different context and to certify them just once (e.g. to support different engine types in our example). In response, many properties have to be defined for a single component to capture all context information.

The main problem here is how to define and to inter-relate all properties thorough the complete system hierarchy in a way that the preservation of properties of all components can be verified automatically? Another problem is how to complete with such a verification in a ”reasonable time”?

2.2 Modelling and Verification Aspects

To narrow the problem statement above, very important prerequisite to structure properties within a system hierarchy consistently is to define basic relations among them. For example, properties of the component M_{IS} are related with properties of the component M_{IIAS}, because of direct connections between their output and input variables. On the other hand, properties of all four components influence the semantics of the mentioned top-level property. We summarize different types of these relations as following:

- *Composition*: hierarchical building of composed properties based on their contained properties (e.g. the top-level timing property is composed of properties contained in components M_{AFS}, M_{IS}, M_{FS} and M_{IIAS}). We discuss this later in more detail.
- *Refinement/abstraction*: properties characterize the component behaviour at certain abstraction level. With refined properties, more specialized behaviours can be described. For example, the property in Figure 1 may include some additional parameters to define conditions for the t_{ig} more precisely.
- *Alternatives*: properties may have alternative representations for different context (e.g. the Injection System component M_{IS} can provide different properties for different engine types).

These relations have to be supported when modelling a component-based system and they have to be considered when such a system has to be verified.

3 Constraint-based Verification

In this section, we describe the proposed approach for compositional verification. To get a rough image of our approach, we summarized the basic steps in Figure 2

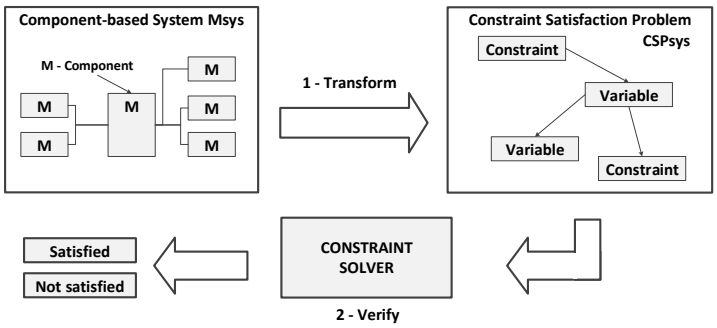


Fig. 2 Overview of the proposed verification method: (1) transformation of the component-based system M_{sys} into the CSP representation CSP_{sys}, (2) verification of the composition CSP_{sys} by solving a CSP

that we perform to conducts the verification process. The input to the verification is a modelled component-based system, enriched with properties – M_{sys} in figure. This model is further transformed into a Constraint Satisfaction Problem (CSP) – CSP_{sys} in figure, which is a network of inter-connected variables and constraints (we discuss this later). The CSP model is processed by the constraint solver, i.e. a tool to solve the CSPs, in order determine the preservation of all properties in the system. As a result, we get a decision about such a preservation. In addition, we get concrete values of data (i.e. inputs, outputs, parameters), for which properties are preserved. All steps in the process are performed automatically.

In the following, we describe how we defined each model described above. We first give some basic assumptions for our system M_{sys}. Then we describe the main elements of that system, including properties. In the end, we describe its representation as a CSP.

3.1 General: Components and Compositions

In our system, we define a component M as follows:

$$M := \langle \Sigma^{in}, \Sigma^{out}, \Sigma^{par}, M_c \rangle \tag{1}$$

, where Σ^{in}, Σ^{out}, and Σ^{par} are inputs, outputs and parameters respectively (i.e. Σ-alphabets define input, output and parameter variables in terms of datatypes, values, and some additional attributes), whereby M_c is an optional set of contained components, and is defined according to relation (1). To clarify this, we distinguish between following two types of components:

- *Atomic components*: components that can not be further divided to form hierarchies, i.e. components for which $M_c = \emptyset$. They perform the concrete computation. The Ignition System for example may contain many atomic components, such as integrators, limiters, simple logical elements and other.
- *Composite components*: hierarchical components that may contain one or more atomic and composite components, i.e. $M_c \neq \emptyset$. Note that we use the term *composition* to indicate composite components, which also may represent a complete component-based system (cf. our system in Figure 1).

The component model introduced above is typical for data-flow systems such as the ones modelled in the Matlab Simulink for example. Similar models are used when considering properties for resource budgets (Benveniste et al (2012)).

3.2 Modelling Compositions Enriched with Properties

As illustrated in Figure 1, properties are defined as expressions over component variables. In order to be able to interpret these expressions during the verification,

we formulate them in a SMT form[3]: each expression can be represented in terms of basic symbols, such as $0, 1, ...s_{en}, ..., +, -, /, ...min$. Using this form, various expressions can be supported for our system, including logical, arithmetic, and other. The property from Figure 1 for instance, $(0 \geq s_{en} \leq 6400) \wedge (0 \geq f_{fl} \leq 100)$, conforms to the SMT form.

In order to link properties throughout the system hierarchy with regard to three basic relations introduced in Section 2.2, we encapsulate them in assume/guarantee (A/G) contracts. According to the general contract theory in (Benveniste et al (2012)), a contract C is a tuple of assumption/guarantee pairs, i.e.:

$$C := \langle \Sigma, A, G \rangle \tag{2}$$

, where A and G are expressions over sets of variables Σ. In this way, we can split properties for each component in (a) part that has to be satisfied, i.e. *assumptions*, and (b) part that is guaranteed if assumptions hold, i.e. *guarantees*. For example, the top-level contract C_{II} for our system in Figure 1 guarantees the $40ms$ delay under assumptions that the rotational speed s_{en} and values for the throttle angle a_{tr} are within certain ranges:

$$C_{II} = \begin{cases} \text{variables} & \begin{cases} \text{inputs} & s_{en}, a_{tr} \\ \text{parameters} & - \\ \text{outputs} & t_{in}, t_{ig} \end{cases} \\ \text{types} & s_{en}, a_{tr}, t_{in}, t_{ig} \in \mathbb{N} \\ \text{assumptions} & (0 \geq s_{en} \leq 6400) \wedge (0 \geq a_{tr} \leq 100) \\ \text{guarantees} & t_{ig} - t_{in} > 40 \end{cases}$$

Based on this structure, we can link properties between dependent components in a similar way it is done when wiring components using connectors (i.e. links between their input/output variables). Figure 3 shows our example system modelled using contracts. Every component provides certain guarantees which stay in rela-

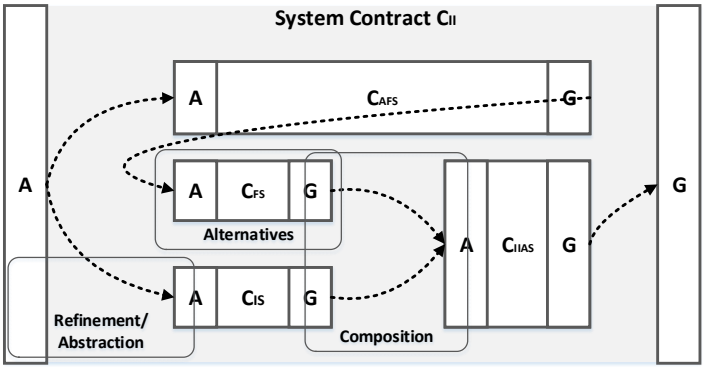


Fig. 3 The Engine Controller system represented using contracts and their basic relations (A – assumptions, G – guarantees, C – contracts)

[3] Syntax in SMT (Satisfiability Modulo Theories) allows to define advanced expressions, e.g. on integers, reals, etc.

tion to assumptions of dependent components. These components in turn provide guarantees based on their own assumptions, and so forth. In this way, all properties within a system hierarchy can be linked together. In Figure 3, we have also highlighted different types of relations between contracts, required to build such a hierarchy (see Section 2.2). These are:

- *Composition*: two contracts can interact when after connecting their guarantees and assumptions both contracts can function correctly (we discuss this in more detail in Section 3.3). We use the operator $\otimes$ to define a composition (Benveniste et al (2012)). An example of such relations is shown in Figure 3, where contracts C_{FS}, C_{IS}, and C_{IIAS} form a composite contract, i.e. $(C_{FS} \otimes C_{IS}) \otimes C_{IIAS})$.
- *Refinement/abstraction*: similar to refinement of properties, contracts refine other contracts in terms of refined assumptions and guarantees. We use the operator $\preceq$ for this relation. The top-level contract C_{II} has such a relation with the contained contracts, i.e. $(C_{FS} \otimes C_{IS}) \otimes C_{IIAS}) \preceq C_{II}$. Note that only the relation with the contract C_{IS} is highlighted here.
- *Alternatives*: when designing components for more than one context, each new context is described in a separated contract. Contracts that describe the same property for different context are alternatives. In example in Figure 3, any of contained contracts may have alternatives – here, we just highlighted C_{FS} to indicate that it may have alternative contracts.

Based on definitions for contracts and their relations, we can now define the top-level system/composition contract, C_{sys}, as follows:

$$C_{sys} := (\otimes_{i \in \mathbb{N}} C_i) \tag{3}$$

, i.e. a hierarchical composition of contracts C_i, where C_i represents further composition according to relation (3).

Finally, to relate contracts with components, i.e. the concrete implementations of contracts, we extend the relation (1) as follows:

$$M := \left\langle \Sigma^{in}, \Sigma^{out}, \Sigma^{par}, C_c, M_c \right\rangle \tag{4}$$

, where C_c is a set of contracts that the component M can implement. Based on this relation, any implementation of the C_{sys} contract represents a complete component-based system or a top-level composition. We identify this implementation as M_{sys} and use it later as a basis to define our CSP.

3.3 Ensuring Correctness of Compositions

For our component-based system defined previously, two contracts C_1 and C_2 can form a composition (i.e. can be integrated) when their connected assumptions/guarantees match in the syntax of their variables (i.e. datatypes, units, etc.), and when following holds:

$$G(C_1) \subseteq A(C_2) \tag{5}$$

In other words, the contract C_1 shall not provide values not assumed by the contract C_2. This relation is a basis in our CSP to verify the complete composition.

3.4 Composition as a Constraint Satisfaction Problem

Now, we describe how we define the composition M_{sys} as a CSP. We name our CSP representation of M_{sys} as CSP_{sys}, and define it as follows:

$$CSP_{sys} := \langle X_{CSP}, D_{CSP}, C_{CSP} \rangle \tag{6}$$

, where X_{CSP} is a finite set of variables, D_{CSP} their domains (datatypes, values), and C_{CSP} a set of constraints related to variables and constraints in C_{CSP}. In other words, the CSP represents a network of variables inter-connected with each other using constraints. The constraints set variables in relations using some operators, and in this way they form expressions. Various types of expressions can be used to define constraints (e.g. Boolean, SMT – depending on supported features of the solver). The solution of the CSP_{sys} is a set of values of X_{CSP} for which all constraints C_{CSP} are satisfied. The constraint solver performs the task of finding solutions.

In order to represent the composition M_{sys} in a CSP, we need to map the top-level contract structure ((sub-)contracts, variables, and A/G expressions) into the CSP constructs mentioned above. Important aspects of this representation are CSP definitions for (1) a type system, (2) A/G expressions or properties, (3) the structure of components and contracts and (4) the structure of compositions. We can now turn to these representations.

3.4.1 Type System

The CSP tools, i.e. constraint solvers, usually provide the support for several domains to represent various types of variables. Integers, reals, and sets are examples here, just to name few. In order to avoid type castings between modelled system M_{sys} and CSP_{sys}, we use the same domains for both M_{sys} and CSP_{sys}. Another reason is that the time needed for the constraint solver to solve the CSP strongly depends on a particular domain. For example, there is a significant difference in runtime when dealing with real numbers instead of integers. Therefore, we use integers for both system representations, i.e. M_{sys} and CSP_{sys}.

3.4.2 A/G Expressions (Properties)

Concerning the representation of values of variables in the CSP, limits have to be set on their intervals. The intervals are possible search space for the solver, and can

have significant influence on solver's runtime. It is therefore important to limit the variables on smallest possible intervals.

In our CSP_{sys}, each variable which is used in an expression is represented by two CSP variables: one indicating the begin of the interval, another one for the end of that interval. The size of this interval is determined based on intervals defined in expressions. For example, the variable s_{en} in the expression $(0 \geq s_{en} \leq 6400)$ is limited on the interval $[0, 6400]$. The reason for using two CSP variables here is that solving the CSP results with not only decision about the correctness of a composition with regard to the relation (5), but it also provides values for which the relation (5) is satisfied. In this way, we can obtain the concrete intervals (instead of just values) for all variables in all contracts (for correct compositions). This information can be useful for example when the composition M_{sys} has many alternative contracts, to observe which of them are identified as correct.

Relations or operations between variables in expressions are represented as constraints. Since both M_{sys} and CSP_{sys} use the SMT syntax for expressions, every operation is represented as a single constraint.

3.4.3 Components

From the perspective of structural organization, every component is represented in a CSP as a set of variables (inputs, outputs, parameters) from the integer domain, and a set of constraints, which correspond to the contracts implemented by that component (see Figure 4). Note that we distinguish here between variables used in

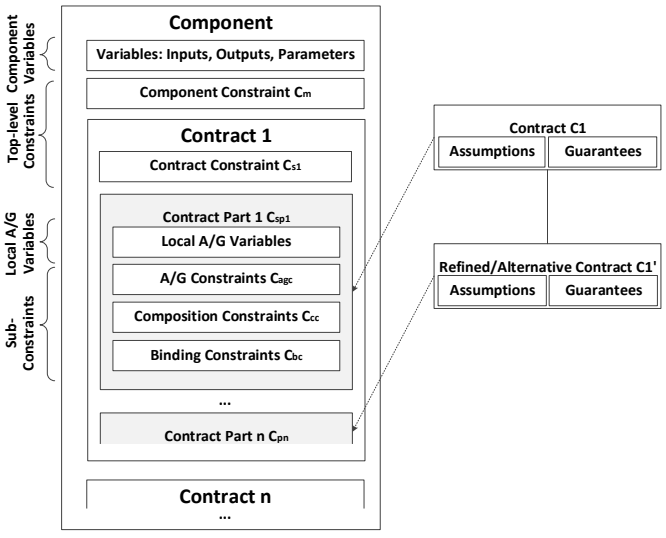


Fig. 4 Representation of a component in CSP (left) and an excerpt of the mapping the contracts to constraints and variables (right)

components, i.e. Σ in relation (4), and variables used in contracts, i.e. Σ in relation (2). Although they are identical, we define separated variables in the CSP for each of them. This means, when a component has two contracts, we have CSP variables for (a) component variables (inputs, outputs and parameters) and (b) CSP variables (inputs, outputs and parameters) for each contract. With this separation of contracts and components, we can identify which contracts are satisfied if the verification succeeds. As mentioned, the constraint solver not only responds with a decision, but it also finds all values of X_{CSP} for which the verification succeeds. Similarly, if the verification fails, the conflicting contracts can be easily identified.

Now we describe how the contracts are defined in a CSP, how they are linked with components, and how the criteria for correctness from relation (5) is represented in a CSP.

3.4.4 Contracts

As shown in Figure 4, each contract is represented as a single top-level constraint C_s. This constraint is further related to a set of local A/G variables (inputs, outputs, parameters) and a set of sub-constraints. The sub-constraints represent the constraints of the refined/abstracted or alternative contracts (contract parts Cs_p in figure). Because refined/abstracted and alternative contracts do not depend on each other, we define the top-level constraint C_s as follows: $C_s := (\vee_{i\in\mathbb{N}} Cs_{pi})$. In this relation, any contract which can satisfy the relation (5) implies that the top-level contract constraint C_s is satisfied.

As illustrated in Figure 4, every contract consists of the following sub-constraints:

- A/G constraints C_{agc}: constraints related only to local A/G variables. These constraints define the assumptions and guarantees for a contract. They are defined based on A/G expressions in contracts, as described in Section 3.4.2.
- Binding constraints C_{bc}: constraints that link the local A/G variables to the global component variables so that both types of variables get the same values. In this way, we can observe which contracts were satisfied, after successful verification.
- Composition constraints C_{cc}: constraints that integrate the contracts. These constraints express the integration or composition between two contracts, as described in Section 3.3. They link two contracts according to relation (5).

All three top-level constraints have to be satisfied for a contract C_{sp}, i.e. $C_{sp} := (C_{agc} \wedge C_{bc} \wedge C_{cc})$.

Finally, the top-level constraint of a component is satisfied, if all contract constraints C_s are satisfied, i.e. $C_m := (\wedge_{i\in\mathbb{N}} C_{si})$.

3.4.5 System/Composition

The compositions have very similar structure to basic or atomic components. Because they abstract some contracts of the contained components, additional con-

straints are defined to link these variables. An example of such a composition is given in Figure 3, where assumptions and guarantees of the contract C_{II} are an abstraction of assumptions and guarantees of the contained contracts.

Like atomic components, the complete component-based system M_{sys} is represented in a CSP as a set of variables and constraints. Within this set of constraints, there is a single top-level constraint of the composition C_m which links the complete hierarchy of the sub-constraints and variables discussed previously. The CSP has a solution only if this top-level constraint is satisfied. Finally, the C_m corresponds to the top-level constraint in the constraint set C_{CSP} from the relation (6).

4 Experimental Results

In the following, we describe the results of the preliminary evaluation and we discuss the performance of our approach.

To conduct the experiment, we used Java-based Choco constraint solver (choco Team (2010)). In our experiment, we defined the composition M_{sys} as a XML description, which is then used to generate the CSP in memory.

The main goal of this experiment is to show whether the proposed CSP is applicable to solve the composition problems defined with data properties, and for which system configurations. We conduct the experiment by showing how the verification responds with regard to attributes that might have an effect on runtime. These attributes include:

- Components and properties: how the verification scales with regard to number of components and properties, including also the presence of the alternative properties.
- Nature of properties: different properties may require different expressions in the CSP, including operations on fixed values, intervals, or more advanced operations such as ones used to define resource constraints (e.g. sum, min, etc.).

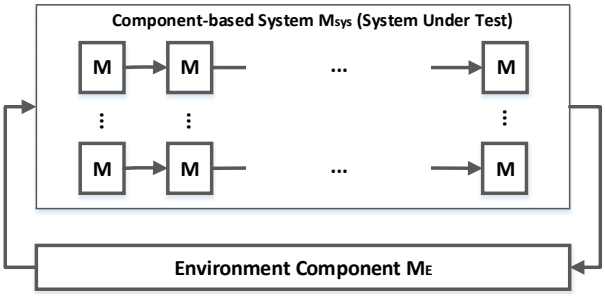


Fig. 5 System configuration used to conduct the experiments (M - component, M_E - environment component)

Figure 5 shows the system configuration used to conduct the experiments. The inputs for the verification are provided by the Environment component, which encloses the component-based system under test. All experiments were executed on Intel i7-3630QM, 4 cores, 2.40GHz.

4.1 Quantitative Results

For this experiment, we performed two measurements. In the first measurement, we show the response time with regard to the number of components, properties and alternative properties, having specified assumptions and guarantees as intervals. Then, in the second measurement, we use the same configurations but with fixed values for expressions. With these two measurements, we are able to observe the limits on modeling the component-based system with regard to number components, properties, and expressions used to describe the properties.

4.1.1 Measurements

In the first measurement, we execute several thousands of system configurations with the varying number of components and properties. The measurement has two parts. In the first part, we verify the system configurations with the varying number of components, each having varying number of properties but with constant number of assumptions or guarantees (i.e., each component variable is therefore related to only one expression). In the second part, each of the components has varying number of alternative and refined properties, so that many solutions are possible. In this case, each component variable is related to many expressions.

The expressions in the first measurement are defined in a way that always the intervals of the component variables have to be satisfied, and not the fixed values. An example for such expression is given in Section 3.2 for the contract C_{II}, which is satisfied only if the variables s_{en} and a_{tr} are in ranges $[0,6400]$ and $[0,100]$ respectively.

For the input test data, i.e. the operands of the assumption and guarantee expressions, we generate the values for each expression randomly, but with the rule that the assumptions are always satisfied. The advantage of performing the positive tests here is to get more clear statement about the runtime of the verification. In both parts of the measurement, we use the relational and logical operations on values.

In the second measurement, we execute the same system configurations as in previous measurement, but this time using the fixed values for component variables.

4.1.2 Observations

First results of the experiments are illustrated in Figure 6. On the left, an excerpt of the results for the first measurement is shown, where the properties have a constant number of assumptions and guarantees. The reason why the verification responds in

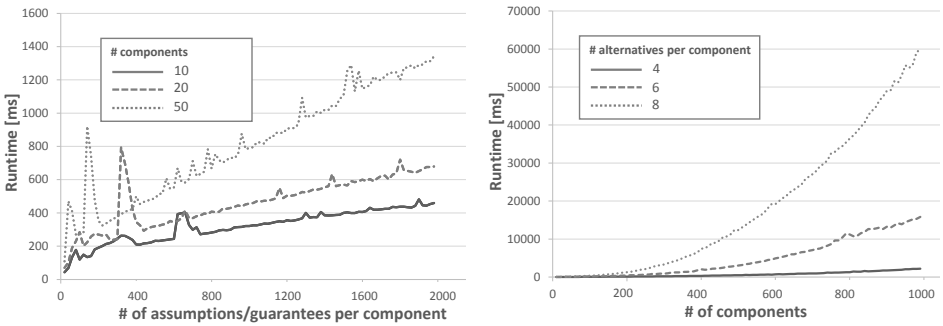


Fig. 6 Experimental results: runtime for system configuration with varying number of assumption/guarantee expressions and components (left) and varying number of components and alternative properties (right)

short time is that each component variable has only one expression (assumption or guarantee constraint, C_{agc}), and it is then immediately instantiated to a value indicated by that expression. The runtime depends in this case therefore on the number of components and properties.

On the right in Figure 6, a scenario that is more likely to occur in practice is shown. Here, each component variable has an increasing number of expressions, and these expressions are alternatives (as mentioned in the description of the measurement). The response time of the verification strongly depends on the number of alternatives, because each of the expressions represents different interval. The solver has to adjust the component variables to adequate intervals, in order to find a solution. Furthermore, since the choice of the particular alternative may influence the choice of the intervals in other connected components, often the backtracks have to be done to the state where the constraints were satisfied, which is time consuming.

In the second measurement, we observed very similar results as illustrated in Figure 6 on the left. Having fixed values on component variables, no search has to be performed, but just the constraint verification. For the case where the alternatives are used, more time is required to find a solution, but this time is negligible in contrast to situation when using intervals (i.e. Figure 6, right).

In the end, we summarize our observations with Figure 7. This figure shows the region for which the verification can complete in a ”reasonable time”. We set the limit for this time on 2 minutes, just to get a first feedback about possible configurations for the system under test. To establish this region, we used the system configuration with the worst case in response time, i.e. the one having the alternative properties from the first measurement.

4.2 Qualitative Results: Discussion

Figure 7 shows the worst-case scenario, in which a component-based system is modelled having varying number of assume guarantee expressions. The verification

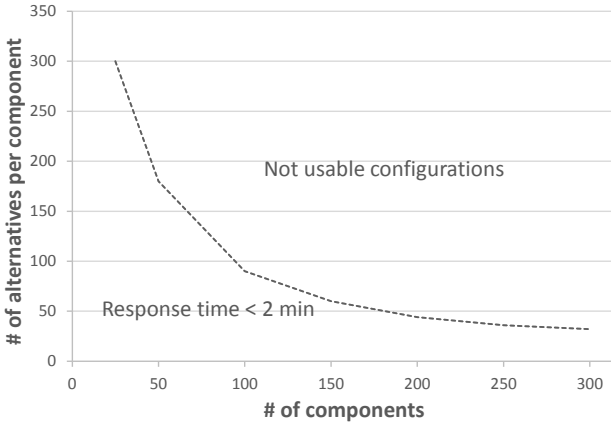


Fig. 7 Region of possible system configurations for which the verification completes within a given time

scales well but for configurations with only few instances of either components or properties. In nowadays automotive systems for example, there are more than 800 software components, that control various technical sub-processes in automobiles (Kindel and Friedrich (2009)). However, it is still possible to support these configurations, since each such sub-system can be provided to verification independently, and also, not all components are massively interconnected as in Figure 5. For example, the simplified system from Figure 1 is modelled using 13 software components (is just one option to realize that system).

5 Related Work

Now we turn to a brief overview of related studies. We summarize here some relevant articles that handle compositional verification based on data semantics.

Similar problems to those described in our problem statement were identified by Sun et. al (Sun et al (2009)) in their work on verifying the composition of analogue circuits for analogue system design. In their approach, each analogue element (resistor, capacitor, etc.) is characterized by its performance profile and this profile is used to build the contract; that is, for certain values of the inputs the element responds with certain output values. Using contracts made from performance profiles, it was possible to eliminate many integration failures early in the system design

phase. These structural compositions of analogue elements are very similar to the compositions in CBSE. However, the model of Sun et al. only considers connections between elements (horizontal relations).

Another article describes a runtime framework for dynamic adaptation of safety-critical systems in the automotive domain (Adler et al (2011)). In the event of failures or degradation of quality, the intent is to reconfigure the automotive system while it is operating. In contrast to the previous approach, the compositional verification in this case is based on a common quality type system shared among components. Two components can form a composition only when their interfaces or ports have the compatible type qualities. In this way, wrong type castings between components can be avoided. However, using a type system in our case would just verify the syntax but not the semantics of data (i.e. the concrete values).

A more advanced framework for dynamic adaptation of avionics systems was developed by Montano (Montano (2011)). The goal is to adapt the system to new, correct configurations, in case of failures. To perform this, a common quality system defines the contracts between functions and available static resources (e.g. memory consumption, CPU utilization, etc.) and in this way it restricts the possible set of correct configurations. An important aspect of this work is that it demonstrates the CSP approach to solving the composition problem. However, the quality type system only considers static resources, and does not consider contracts between functions. Ultimately, the approach is strongly focused on dynamic adaptation with human-assisted decision making.

In the field of industrial automation, the authors in (de Sousa (2012)) propose the static verification of compositions based on data types of the IEC 61131-3 component model (or standard). This model defines the standard data types but it also allows definition of customized data types (derived from existing ones) and combination of existing data types into complex structures. The authors identified ambiguities in the standard for user-defined data types and defined a proper compatibility criteria. Like the adaptation approach in the automotive domain (Adler et al (2011)), this work considers only a type system. However, the approach verifies not only compositions, but also the use of variables in IEC 61131-related languages.

In the last few years, several research projects have begun to handle the topics of compositional verification (SPEEDS (2006-2012)), (COMPASS (2011-2014)), (SAFECER (2011-2015)) by formalizing system models (component models) and languages for specification of contracts. These approaches share many concepts, especially contract-based design and formal behavioural verification of compositions. Although our model is conceptually very similar, it differs in that it considers the data semantics of property values, and it addresses a specific type of component-based systems in which data semantics can be used to express the validity criteria for compositions.

6 Conclusion

In this paper, we presented a method for the verification of compositions in component-based systems. The components modelled here are enriched with properties, which describe the data semantics of components. The novelty of our verification lies in representing the composition along with modelled properties as a Constraint Satisfaction Problem (CSP), which allows us to achieve two important objectives. First, using relational, logical and more advanced operators on data, many types of properties can be supported. Second, for properties that use basic logical and arithmetic operators, the verification can scale up to several hundreds of components, each of them consisting of few tens of properties, which makes the approach promising for the use in practice.

As part of our ongoing work, we want to characterize the runtime performance based on different types of properties, since they impact the scalability at most. In addition, we also want to investigate other parameters such as solver search policy, solver engine, etc., in order to find best configuration for the verification method.

References

Adler R, Schaefer I, Trapp M, Poetzsch-Heffter A (2011) Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. ACM Trans Embed Comput Syst 10(2):20:1–20:39, DOI 10.1145/1880050.1880056, URL http://doi.acm.org/10.1145/1880050.1880056

de Alfaro L, Henzinger TA (2001) Interface automata. SIGSOFT Softw Eng Notes 26(5):109–120, DOI 10.1145/503271.503226, URL http://doi.acm.org/10.1145/503271.503226

Apt K (2003) Principles of Constraint Programming. Cambridge University Press, New York, NY, USA

Basu A, Bensalem S, Bozga M, Combaz J, Jaber M, Nguyen TH, Sifakis J (2011) Rigorous component-based system design using the bip framework. Software, IEEE 28(3):41–48, DOI 10.1109/MS.2011.27

Benveniste A, Caillaud B, Nickovic D, Passerone R, Raclet JB, Reinkemeier P, Sangiovanni-Vincentelli A, Damm W, Henzinger T, Larsen K (2012) Contracts for Systems Design. Tech. rep., Research Report, Nr. 8147, November 2012, Inria

Butz H (-) Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. Department of Avionic Systems at Airbus Deutschland GmbH Kreetslag 10, D-21129 Hamburg, Germany

choco Team (2010) choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO, École des Mines de Nantes

Clara Benac Earle, Elena Gómez-Martínez, Stefano Tonetta, Stefano Puri, Silvia Mazzini, Jean Louis Gilbert, Olivier Hachet, Ramón Serna Oliver, Cecilia Ekelin, Katiusca Zedda (2013) Languages for Safety-Certification Related Properties. In: Proc. Work in Progress Session at 39th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA'13)

COMPASS (2011-2014) Compass - comprehensive modelling for advanced systems of systems. Homepage: http://www.compass-research.eu

Crnkovic I (2002) Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA

Frey P (2010) Case Study: Engine Control Application. Tech. rep., Ulmer Informatik-Berichte, Nr. 2010-03

Gössler G, Sifakis J (2005) Composition for component-based modeling. Sci Comput Program 55(1-3):161–183, DOI 10.1016/j.scico.2004.05.014, URL http://dx.doi.org/10.1016/j.scico.2004.05.014

Kindel O, Friedrich M (2009) Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis. dpunkt Verlag; Auflage: 1 (8. Juni 2009)

Montano G (2011) Dynamic reconfiguration of safety-critical systems: Automation and human involvement. PhD Thesis

SAFECER (2011-2015) Safecer - safety certification of software-intensive systems with reusable components. Homepage: http://safecer.eu

Sentilles S, Štěpán P, Carlson J, Crnković I (2009) Integration of extra-functional properties in component models. In: Proceedings of the 12th International Symposium on Component-Based Software Engineering, Springer-Verlag, Berlin, Heidelberg, CBSE '09, pp 173–190, DOI $10.1007/978-3-642-02414-6_1 1$, URL http://dx.doi.org/10.1007/978-3-642-02414-6_11

de Sousa M (2012) Data-type checking of iec61131-3 st and il applications. In: Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on, pp 1–8, DOI 10.1109/ETFA.2012.6489534

SPEEDS (2006-2012) Speculative and exploratory design in systems engineering - speeds. Homepage: http://www.speeds.eu.com

Sun X, Nuzzo P, Wu CC, Sangiovanni-Vincentelli A (2009) Contract-based system-level composition of analog circuits. In: Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE, pp 605–610

Tran E (1999) Verification/validation/certification. Carnegie Mellon University, 18-849b Dependable Embedded Systems

Towards Pattern-based Reuse in Safety-critical Systems

Nermin Kajtazovic, Institute for Technical Informatics, Graz University of Technology
Christopher Preschern, Institute for Technical Informatics, Graz University of Technology
Andrea Höller, Institute for Technical Informatics, Graz University of Technology
Christian Kreiner, Institute for Technical Informatics, Graz University of Technology

Challenges such as time-to-market, reduced costs for change and maintenance have radically influenced development of today's safety-critical systems. Many domains have already adopted their system's engineering to support modular and component-based architectures. With the component-based design paradigm, the system engineering is utilized allowing to distribute development among different development teams, however, with the price that there is no full trust in independently developed parts, which makes their reuse challenging. Until now, many approaches that address reuse, on conceptual or detailed level, have been proposed. A very important aspect addressed here is to document the information flow between system parts in detail, i.e. from higher abstraction levels down to the implementation details, in order to put more trust into independently developed parts of the system.
In this paper, we describe a compact pattern system with the aim to establish a link between high level concepts for reuse and detailed description of the behavior of system parts. The main goal is to document these details up to the higher levels of abstraction in more systematic way.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*

1. INTRODUCTION

Development of safety-critical systems is currently confronted with challenges related to continuous increase in complexity of hardware and software. For instance, in the avionic domain, it was reported that number of software-implemented functions in aircraft systems had an exponential growth in the last decade [Butz 2010]. Similar statistics are also known from the automotive domain [Kindel and Friedrich 2009]. As response to this problem, especially in domains that have mass production in focus, many domains have already adopted their development process to support modular and component-based architectures. With modular approaches, the system development is utilized to allow development of parts or components independently from the system, thus enabling to distribute that process across different development teams/organizations. The consequence of this strategy is a considerable reduction in costs for change and maintenance, and also improvements in reuse [Crnkovic 2002]. Another reason for adopting the component-based approach is to allow special stakeholders such as customers to define and to maintain the system functions on their own. Examples are Programmable-Logic-Controllers (PLC) based on the IEC 61131/61499 industrial standard, which allow to build the functions using pre-defined blocks (components), initially provided by developers.

Though reduction in costs for change and maintenance, adopting the component-based approaches to safety domain rises some new challenges. One of the major problems, which are also a subject of consideration in some current safety standards, is to ensure the safety of the overall system that is constructed from components. The main assumption here is that components are developed independently from the system, i.e. in an isolation, so

that a context in which they should operate is often not considered in detail. For software components, such a context may comprise the interface semantics to connected neighbouring components, representing valid ranges of transmitted data, their physical units and certain component states in which this data can be transmitted for example. On the other hand, the context may also describe an environment in which such components have to run (e.g. a platform, an embedded system with some characteristics such as the runtime memory layout and its partitions). Generally, the system integration alone is a challenging task, i.e. when just using components within a single project, and requires a good communication between teams and an adequate system engineering. However, in this case the context information is still present inside the project, i.e. development teams can resolve the integration issues at any time by collaborating with each other. A more difficult scenario is when such isolated components have to be reused in different projects, especially when there is no maintenance support for components and the only source of the context information is a component documentation. In the worst case, such documentation does not describe the context sufficiently, so that much effort has to be invested in the verification phase to identify the potential integration problems. Due to limited budged reserved to perform such an extensive verification or when not considering the reuse aspect in the development process, many faults may remain in an integrated system and can be manifested in the operation. This was a common scenario in a few cases where mission and safety-critical systems failed to perform their function and resulted with disastrous consequences [Gao et al. 2003].

In order to address reuse in a more systematic way, there is a need (1) to document the behavior of components so that they can be properly reused and (2) to provide a theory for their composition – how to estimate the functional and extra-functional system properties when connecting properties of components together [Crnkovic 2002]. Examples of these properties are aforementioned valid value ranges, assertions on behavior, timing and resource constraints, and others [Clara Benac Earle et al. 2013].

Concerning the first aspect, i.e. the documentation for reuse, some of the current safety standards have already developed concepts. One of the examples is a Safety Element Out of Context (SEooC) concept in the automotive standard ISO 26262, which provides guidelines on how to integrate components developed and provided by the suppliers. Similarly, the Reusable Software Components (RSC), made according to the DO-178B standard in the avionic domain, defines how to design and to document components so that they can be exchanged without having significant additional certification costs. In general, these concepts capture high-level aspects, and details on documentation and criteria on how much is enough for reuse are left to domain experts.

From the perspective of the second aspect, i.e. concrete theories and approaches for composition, many solutions have been developed until now. Although most of them have no direct focus on safety-critical systems, they have considered the problem of compositions earlier for general-purpose systems, which provide a helpful basis for safety-critical systems. Generally, in order to ensure the composition or integration of components, it is necessary to ensure the correctness with regard to their syntax and semantic. For syntax, the interfacing between components has to match, while for semantic more advanced engineering is required because the functional and extra-functional properties have to be considered here. Many approaches that consider semantic while integrating components, are mainly based on formal methods. They abstract the components usually as state transition systems and verify their behavior by using model checking techniques [Basu et al. 2011]. Other approaches focus on analyzing static resource properties such as timing and memory consumption [Lévêque and Sentilles 2011]. Although these approaches represent just an excerpt of the extensive verification and validation process, they provide very important information about components – the concrete and precise assumptions under which components behave correctly. Another important fact is that these assumptions often describe recurring statements over behavior or data of components, which makes them attractive from the viewpoint of reuse, as they represent some kind of specification patterns [Evans and Fowler 1997].

From the viewpoint of system integrators and component developers, it would be very beneficial if they would be able to share the knowledge using these assumptions/specifications in a structured way. Also, a better link between aspects discussed above could be established. For example, the system integrator could recognize

patterns used to describe a valid behavior of components with regard to certain inputs, and based on this, he could approve or disapprove reuse. In this paper, we describe a pattern system with the aim to link the detailed specification patterns with the high level system concepts for reuse. With these links, more precise description of the system parts throughout the system hierarchy can be achieved.

The remainder of this paper is structured as follows: Sec. 2 introduces the problem statement more in detail and outlines main objectives of this work. In Sec. 3 related work is described. Sec. 4 describes the proposed pattern system. A brief discussion about some important aspect for the practical use of the proposed pattern system is provided in Sec. 5. Finally, Sec. 6 gives concluding remarks.

2. PROBLEM STATEMENT, OBJECTIVES AND TARGET AUDIENCE

To narrow our problem statement, let us consider now the motivating example from Fig. 1. The figure shows an excerpt of the top-down component-based development process for the automotive engine control function. Basically, the objective of this control function is to decide, based on certain inputs such as the angle of the throttle position, current engine speed and air volume, when to perform the tasks of the fuel injection and ignition [Frey 2010]. Development of systems like this is usually distributed, because of a need for special knowledge in certain engineering fields, better handling of complexity (this is only a part of the overall system), and time-to-market push.

In the example above, the development process is shared between system developers (in further text manufacturer) and part/component developers (suppliers). In the first phase, the manufacturer defines the system by decomposing the requirements down to the fine grained functions/components. These functions represent the boundaries of responsibility – they are further provided as requirements to suppliers, which have to implement them using their own engineering techniques. This implementation may relate to reuse of the in-house components that match to the supplier's requirements for example. In the end, all supplied parts are integrated together in the last phase.

In order to successfully perform the integration, the information flow between manufacturers and suppliers have to be detailed enough, i.e. the manufacturers have to provide what they want, and the suppliers have to provide the information on how to use their parts. Considering time between injection and ignition for example, the question is how the manufacturer can ensure that this time does not exceed 40 ms based on properties of supplied parts/components? This is, of course, verified later in the test integration phase, but in order to early identify potential faults and to reduce the time for the integration, it is very important to previously know which properties contribute to that time. This is just an excerpt of large number of properties and aspects that might have to be considered in the integration.

Another aspect where such properties can be beneficial is the system verification phase. Usually, during this phase the manufacturers have to follow some methods provided by the safety standards to implement the integration tests. Similarly, if there are components on the supplier's site that consist of many other components, they have to perform the same procedure. These methods are collection of various types of tests for hardware and software systems, including the integration tests on hardware, software and system level, various types of black-box and white-box software tests and other [Smith and Simpson 2010]. For selected methods, appropriate test cases have to be provided, and test reports have to be produced as an evidence that the integration is performed according to the standard. With precise information about the system and its context described by properties, it would be possible to derive the test cases directly from such information, and therefore to reduce the time to implement tests and to early achieve the coverage required by certain methods (e.g. call coverage, function coverage).

Problem Statement: how to define the context information that may describe various types of functional and extra-functional requirements for a single component in a way that such information can be easily compared or matched with information of other, dependent components[1]? In our example, the supplier may need to match

[1] Components from different layers in the hierarchy, or connected components (horizontal connection).

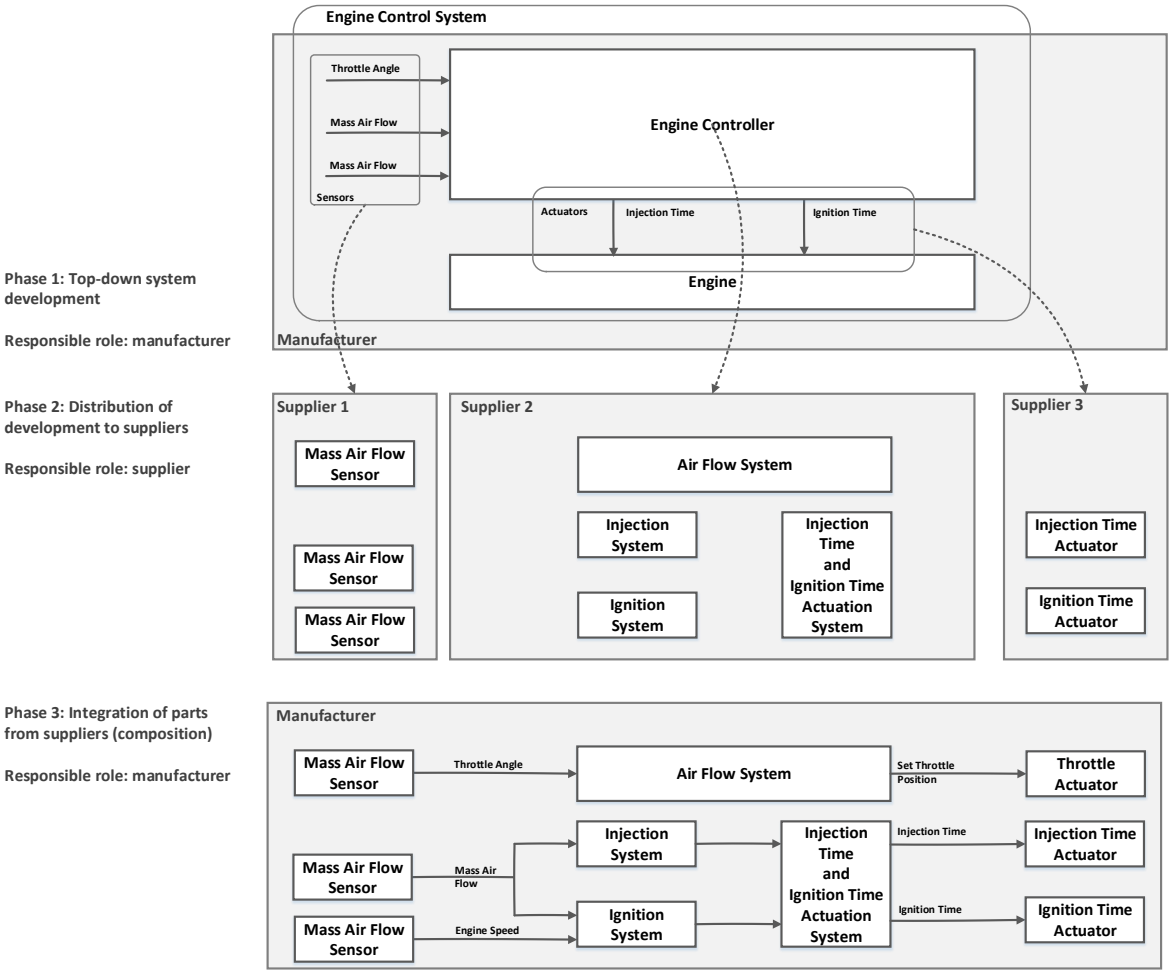


Fig. 1. Motivation example: distributed development of engine control application, adopted from [Frey 2010]

the context information provided by the manufacturer (as requirements) to identify the potential candidates of the in-house components which he may reuse. On the other hand, the manufacturer has to ensure that supplied components conform to the high level concepts, like SEooC[2].

To address this problem, following aspects/objectives have to be considered, from both system and component development perspective:

—Representation of reusable artifacts.
Like in Software Product Lines (SPL), the very first step in domain engineering is to define the product artifacts with regard to the product scope (software, documentation, requirements, etc.). The question is now how to represent different artifacts of all involved teams/organizations so that they can easily understand the information of each others. For example, the Supplier 2 from Fig. 1. can deliver a complete device including operating system, middleware and application, whereby Supplier 1 and 3 may deliver software application only, and the automotive manufacturer has just the responsibility over the system functions. The boundaries of components shall not be fixed to some specific structure.

[2]SEooC is a generic component having described context in form of Safety Requirements (typical extra-functional requirements).

—Representation of information flow between reusable artifacts.
There are two independent aspects of representing the information flow. First, from the perspective of a single team/organization, the components can be refined or abstracted, as illustrated in Fig. 1 – the Supplier 2 has refined the Engine Controller function with three sub-components. In order to satisfy the requirement(s) of the Engine Controller function, the supplier has to define criteria on how each component contributes to these requirements (because the refined components have additional properties which contribute to the top-level requirements). Second, the information flow between components that belong to different development teams has to be defined. This is more challenging because the main assumption from the perspective of the system development is that it is known which services the components provide but not how.

In following sections, we build a pattern system based on objectives discussed above. The target audience in this context are stakeholders in the system engineering (system architects) who have to design their safety-critical systems/components for reuse. Note that in this paper we only consider project artifacts and interfaces between them, and not the development process, or the project management aspect of the system.

3. RELATED WORK

In this section, we outline some relevant works related to general reuse methods, state-of-art for reuse in safety-critical systems and reuse by applying patterns.

3.1 Systematic Reuse

The motivation behind reuse of hardware/software systems were factors that have improved the production in development of automobiles in their early manufacturing phase, and later the adaptation of the same strategy for consumer electronics [Pohl et al. 2005]. In software engineering, the software product lines engineering (SPLE) define the principles of systematic reuse for software products. At this time, the SPL are the only approach that focuses on systematic reuse in software engineering. In general, the aim is to prepare the development process, the project lifecycle and different core assets/project artifacts for reuse. The main enabler technologies for the SPL are generative and compositional approaches – to highlight some of these, generative programming and component-based paradigm or aspect oriented programming are widely applied in this context. Some domains with the mass-production in focus have already adopted the concepts developed according to SPL principles, but they are still not fully utilized in production [Kindel and Friedrich 2009], [EAST-ADL 2010]. There is too much overhead of managing many artifacts and dependencies among them.

3.2 Reuse in Safety-critical Systems

Some important approaches related to reuse in safety domains were already discussed in the introduction. Basically, from the perspective of reuse, there are two trends. First, there is a need for development of the concepts and methods for integration of components based on their functional and extra-functional aspects. Examples are mentioned SEooC and RSC concepts. Common to these concepts is that they define the assumptions that have to be met by the remaining system (or environment) in order to ensure that the components will behave as expected. Although not explicitly mentioned by the approaches, this is a typical principle of assume/guarantee contracts from the contract-based design [Benveniste et al. 2012]. In this way, reuse can be approved only if the contracts of the system (or environment) and the corresponding contracts of the components are valid. Similarly, contracts can help to systematically refine the system down to the implementation details [Benveniste et al. 2012], but this is not considered in the aforementioned concepts.

The second trend is to provide the argumentation that a system is acceptably "safe". This is a structured collection of documents, usually specified in a Goal Structuring Notation (GSN), which links the arguments and evidence (such as test cases) for the overall system. For component-based systems, many of these documents are defined out-of-the-box (i.e. on the side of the component developer) and have to be reused. In the last decade,

Table I. Relevant concepts and ideas from related work

Concept	Purpose
Software product lines [Pohl et al. 2005]	Defining reusable components in very early stages of system's development – in a domain engineering.
Contract-based design [Benveniste et al. 2012]	A key technology/principle for describing information flow between components ((1) components which are owned by different development teams and (2) components in the hierarchy (refined, abstracted components)).
Property specification patterns [SPEEDS 2010], [SpecPatterns 1998]	Precise and detailed description of functional and extra-functional aspects of the components.

some methods have been developed that consider component-based argumentation using modular variant of the GSN [Kelly 2001]. Very important aspect of these methods is that they describe concepts that are practically usable, simple to maintain, and are promising for certification of component-based safety-critical systems.

3.3 Pattern-based Reuse

Reuse methods for argumentation have always in context system's safety, i.e. the argumentation that certain failure modes collected in hazard and risk analysis are addressed and eliminated. On the other side, reuse concepts SEooC and RSC describe high-level views of the system. Until now, only few works have considered reuse with regard to functional and extra-functional properties in general. In late nineties, a project Spec Patterns was started with the aim to collect patterns that describe behavior of concurrent and reactive systems [SpecPatterns 1998]. The outcome of the project was a taxonomy of patterns that can be shared among developers who want to specify behavior of systems in a standard way. Further mapping of the patterns to concrete syntax such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and others, is defined. Another, and very current, pattern collection for specifying system properties has been developed in the project SPEEDS [SPEEDS 2010]. In contrast to the Spec Patterns, component-based systems are addressed here and more attention on reuse has been given. These two projects provide a fundamental basis to reuse parts between involved development teams on a very low and detailed implementation level.

In order to establish a strong link to general concepts like SEooC and RSC, except of having pattern-like standard structure, the concrete property specifications have to be somehow documented up to the high level requirements of the system developer (e.g. how the properties of the supplier's Ignition System from Fig. 1 contribute to the requirements of the Engine Controller?). In Tab. I we summarize some relevant ideas from related work.

4. PROPOSED PATTERN SYSTEM

In this section, we describe the pattern system that address objectives outlined in Sec.2.

4.1 Overview

Fig. 2 shows the main components of the proposed pattern-based reuse approach. On the left, two mentioned levels of system abstraction and existing methods on how reuse is addressed there are depicted. A compact pattern system on the right represents a link between both levels of abstraction by (1) standardizing the representation of reusable components and (2) a way on how they exchange the information.

The central component in the figure below is a contract. It defines a "frame" for reuse for any artifact or component in any abstraction level. For a given high-level concept, such as the SEooC for example, the contract allows to stepwise refine the system down to details where the properties arise.

In general, contracts capture the information that is needed to verify syntactic and semantic compatibility (1) between dependent components and (2) between components and their platform. Some examples were already introduced in Sec.1[3]. Many of the functional and extra-functional properties that represent the context information

[3]For the complete list of properties, refer to [Clara Benac Earle et al. 2013].

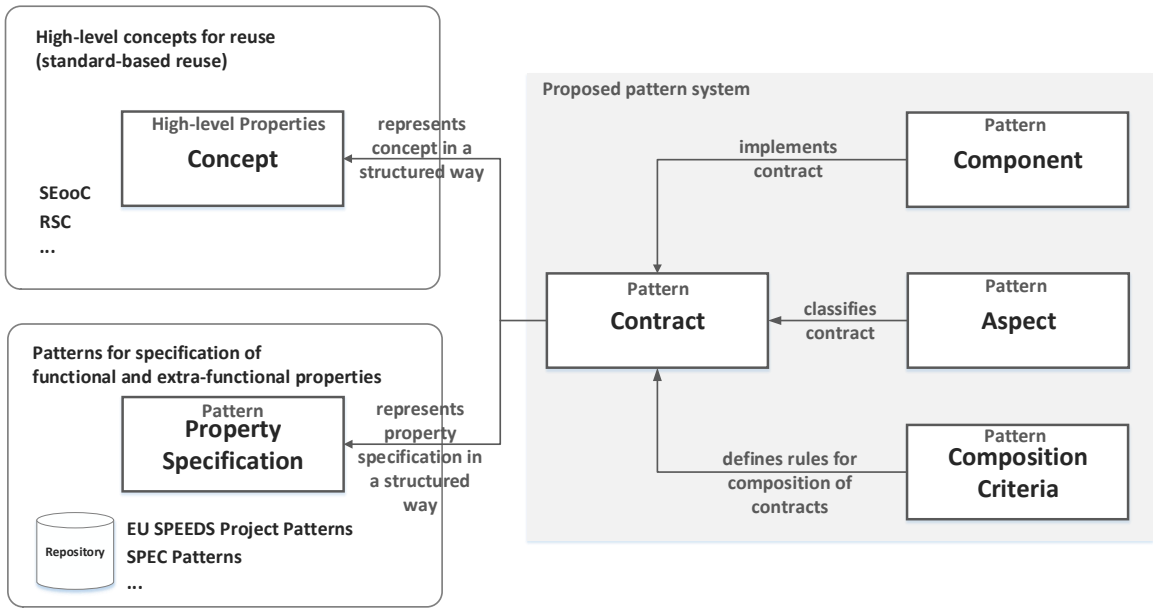


Fig. 2. Proposed patten-based reuse: high-level concepts for standard-based reuse, low-level property specification patterns (left) and pattern system (right)

can be defined in terms of data and behavior of components put into simple relations (such as aforementioned 40ms delay, resource constraints etc.), while for some of them only the natural language is applicable (e.g. read input every 20 ms). We assume that there is a repository of reusable property specification patterns that may describe semantics and behavior of components for both types of properties. Obviously, for the second type it is much more difficult to determine whether two contracts are compatible, because there is more space for ambiguities due to natural language and therefore (a lot of) manual work could be required to do such a reasoning. However, with the standard set of specification patterns many problems when comparing/matching contracts will repeat so that this information, i.e. experience from previous comparisons, can be used to simplify later reasoning. The specification patterns will enforce the developers to express their functional and extra-functional properties for components using the standard format, whereby the pattern system will support them while they are looking for identical contracts, e.g. when identifying candidate components that suppliers may reuse based on the specification of the manufacturer.

4.2 Patterns

Our pattern system is based on concepts from Tab. I, especially on work described in the contract theory [Benveniste et al. 2012]. We use the POSA-style to describe the patterns from Fig. 2 [Buschmann et al. 1996].

4.3 Contract

—**Context**

Safety-critical system is planed for the development. There are some components that have to be built for further reuse in the later projects or they have to be delegated to other teams for the development. To enable this, the developers have to define and to organize the context information thorough the complete system hierarchy.

—**Example**
Assume the following top-level requirement with regard to the system from Fig. 1.: "Time between fuel injection and fuel ignition shall not drop below 40 ms". This requirement is provided to the Supplier 2 for the Engine Controller function. The supplier has to somehow document how the entire system, the components, contribute to the timing. For example, this requirement can only be satisfied if the rotational engine speed does not exceed 6000 rpm (see engine speed signal in figure). Further, the computation can only be reliable, if the sensor signals are sampled twice than the supplier's system. Issues:
—How to link the properties such as the speed to the high-level timing?
—How to communicate the requirement on sampling period to the Supplier 1?
—How the system developer can verify the consistency with respect to the low level properties?

—**Problem**
Using typical requirements to define the responsibility of components ensures just uni-directional relation between the specification and implementation. Although requirements often may contain the context information, such information is not explicit visible, and therefore is not systematically considered as a first class entity. The requirements are not absolutely valid and they always need a context. How to structure requirements throughout the system hierarchy so that every project artifact that has a requirement includes the context information, which defines the scope of that requirement?
This implies following forces:
—The system is hierarchical and parts of that hierarchy might be in responsibility of different teams (i.e. they have to be exchangeable, reusable).
—Many project artifacts such as software components interact with other artifacts in the same layer (horizontal communication) or in different layers (vertical communication) and therefore influence their context.
—The platform such as an embedded system in which some project artifacts exist during the operation may influence their context.

—**Solution**
Structure a requirement (of any abstraction level) as a *contract* in the following way: set of *assumption* and *guarantee* pairs. The guarantees contain the original requirement, without the context information. Hence, the context information is put into assumptions. In this way, we have a structure that enforce the developers to always consider the context in which a given requirement is valid. For every requirement that have to be guaranteed by some artifact, there is always at least one assumption describing the context for which that requirement can be satisfied.

—**Implementation**
Top-down process (isolated development):
—Break down the system requirements into high-level contracts.
—Define assumptions for which each of the requirements has to be satisfied. This is very important, because with assumptions we are able to specialize the context and interfacing with other contracts. The final contract has to contains a list of assumptions and guarantee pairs that completely describe the given requirement. Early note: for defining assumptions and guarantees look at the repository of property specification patterns. Identify patterns that best describe the assumptions and guarantees (see COMPOSITION CRITERIA pattern for a reason of using property specification patterns).
—Build the system hierarchy by decomposing contracts into refined contracts.
Bottom-up process (integration or composition):
—For a given component to reuse, verify compatibility between related component contracts according to composition criteria and define a composed contract (see COMPOSITION CRITERIA pattern).

—**Consequences**
$(+)$ All parts in the system hierarchy (components, layers) have defined requirements in terms of guarantees,

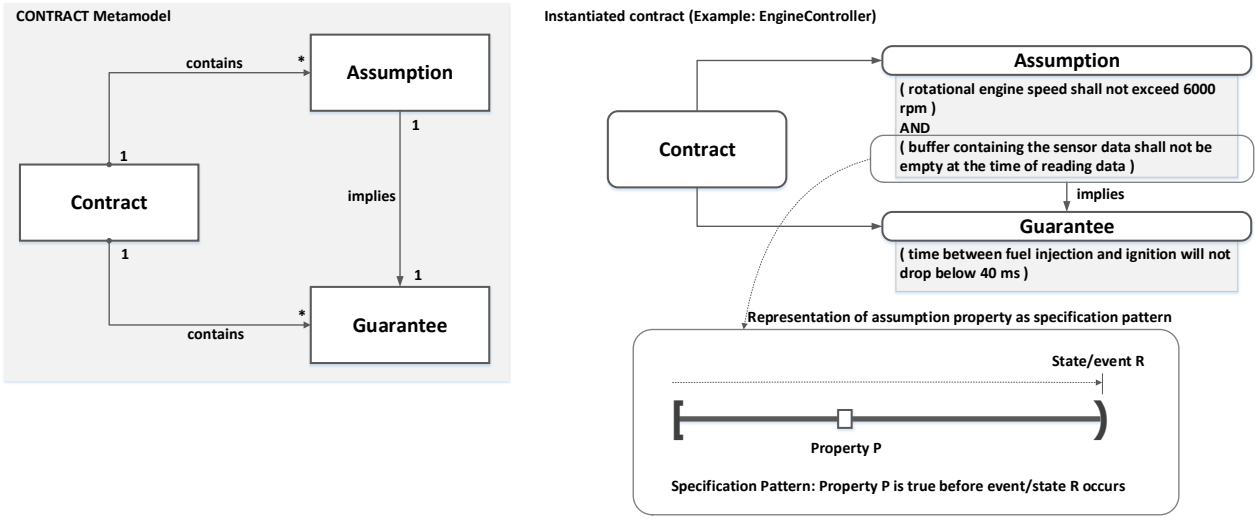


Fig. 3. Contract meta-model (left) and an exemplary instance that shows the contract for the EngineController component (assumptions and guarantees are described textually, but in original form property specification patterns are used - an example is given on bottom of figure)

and context in terms of assumptions. Each part can therefore be exchanged, without loosing the context.

(+) The context information captured in assumptions of project artifacts also includes the requirements on dependent artifacts from the same or different layers (e.g. valid range of the signal that an external component has to provide).

(+) Assumptions can be made on environment, platform. Similarly, the platform may set assumptions on project artifacts (e.g., to constrain the number of used software components to maintain memory budget).

(−) Identifying property specification patterns to describe assumptions and guarantees can be a tedious task – textual requirements have to be translated into more formal description.

—**Example Resolved**

Based on time contract given for Engine Controller component, the Supplier 2 defines its top-level subcontract as follows:

—Guarantees: (time between fuel injection and ignition will not drop below 40 ms).

—Assumptions: (rotational engine speed shall not exceed 6000 rpm) and (buffer containing the sensor data shall not be empty at the time of reading data).

Expressions in assumptions and guarantees are represented using property specification patterns. For example, the second assumption above can be defined using the occurrence pattern from the Spec Pattern collection, which states that the buffer where the sensor values are stored always contains data before a read event occurs, see Fig. 3. Using standard representation for all expressions (in form of events, states, data, as in the mentioned pattern collections) simplifies later integration of components, especially when the components contain many properties. Also, ad-hoc description of expressions is avoided.

Based on pattern description for sensor data assumption, the system developer can perform actions such as configure the sensor sub-system to be sampled at higher rate than the Engine Controller sub-system.

—**Known Uses**

—SEooC and RSC - high-level concepts for standard-based reuse. Interfaces are described in similar way as assumptions and guarantees.

—Heterogeneous Rich Components (HRC) - contracts for integrating components in safety-critical systems. Contracts are here parts of the component assemblies.

—Modular safety cases - representation of modular arguments using similar structure as contracts.
—Object-oriented software - contracts define expressions over object methods and its states.

4.4 Component

—**Context**
Organization plans to develop a safety-critical system. It has been arranged that some parts of the system have to be distributed among different teams in development. The overall system is built by reusing those parts. Further, there is no clear distinction what can be reused (complete devices, hardware, software (parts of software layers), and other). Also, the organization may reuse own parts for later products.

—**Example**
We consider again the supplier's Control Engine function in Fig. 1, which is provided as a complete device (hardware/software system). Assume now that the supplier has to take legacy software including operating system and runtime libraries for performing floating point operations for example. The libraries depend on particular processor architecture. In some cases they also depend on certain type of the same architecture, whereas for some architectures with the built-in support for aforementioned operations they are not needed at all. Libraries are just one small excerpt of a very large and complex artifact ecosystem. In the end, the objective of the supplier is to ensure that artifacts are correctly interfacing with each other and that they are properly configured for the intended context. Similarly, the manufacturer has to consider the same problem statements when reusing the Control Engine function.

—**Problem**
It is often required to reuse various existing project artifacts (various software artifacts like functional software components, middleware and operating system, libraries, and others). Without having the explicit information about these artifacts and their context, it is not possible to determine whether they implement given contracts. This implies following forces:
—Project artifacts are strongly narrowed to the context in which they live (as aforementioned architecture-dependent libraries for example).
—Defining artifacts for specific purpose like application software as standardized components for example (as it is a practice in many current component-frameworks), limits the reuse of the remaining parts of the system (operating system services, libraries, and similar).

—**Solution**
Make the information about artifacts and their context explicit so that every artifact has an additional description. Use the same structure as for contracts, i.e. assumptions and guarantees to describe functional and extra-functional properties. In contrast to contracts, this information represents the implemented specification for a particular contract. We use the term *component* to represent artifacts augmented with the context information.

—**Implementation** In case there is a contract that has to be implemented, the process of defining components is the following:
—For a given contract, develop a component or extend a function for an existing component based on guarantees that it has to provide.
—Use assumptions to verify whether a component can provide given guarantees (e.g. by deriving tests from assumptions and guarantees).
For project artifacts for which no real requirements exist (like libraries), simple data contracts can be defined based on basic knowledge about them (required memory, compliance with the processor architecture, endiannes (if binary form), etc.).

—**Consequences**
$(+)$ A basis to build contracts is provided.
$(+)$ More safe and simplified identification of the project artifacts with regard to requirements or contracts (in

some cases and for specific artifacts, automatically).
(+) Involved development teams follow the same structure for their artifacts – exchange of artifacts is simplified.
(−) Overhead of defining components – it is often difficult to capture the profile of components, so that in some cases an extensive verification of components might be required.

—**Example Resolved**
Based on contracts, which the supplier has already defined for his own sub-system previously, the corresponding libraries are identified and defined as components. Having libraries augmented with information about function and interfaces (assumptions, guarantees), it is easy to identify them for the next time they have to be reused (eventually in different context). Also, if enough information with regard to the context and interfaces is collected, it is easy to verify the conformance to the context, and in some cases to perform this verification automatically. Similar to libraries, the supplier provides the Engine Control component to the manufacturer.

—**Known Uses**
Standardizing representation of project artifacts has been used in many application fields in order to improve reuse and to reduce the costs for change and maintenance. AUTOSAR, EAST-ADL, IEC61131/499, IEC61850 are some of the component-based systems in safety domain, just to name few.

4.5 Composition Criteria

—**Context**
Safety-critical system is in the integration phase. The system developer has already the complete system hierarchy in form of contracts. Independently developed parts have to be integrated now. That means, the compatibility between guarantees and assumptions of dependent components have to be verified. Alternative scenario: the component developer tries to identify reusable components that match to contracts given by the system developers.

—**Example**
We continue with the last example where the Supplier 2 has defined the contract for the Supplier 1 so that the sensor sub-system and the Engine Controller must have certain sampling rates. The question is, how to take such a decision based only on information about the sub-system of the Supplier 2? This implies to consider following points:
(1) How to decide whether the contracts of the two sub-systems are compatible?
(2) Based on which information is the sampling rate the crucial for the fulfillment of the contract?

—**Problem**
Due to ambiguities in the natural language, which is used to define assumption and guarantees in contracts, it is difficult to verify the compatibility between contracts. How to conclude, based on definition of contracts, i.e. property specification in assumptions and guarantees, that two contracts are compatible? Forces:
—The contracts that have to be compared might be defined by the development teams independently. Therefore, it is likely to have different representations of contracts, at least for data values used in patterns. In worst case, the teams may use different patterns to describe the same contracts.

—**Solution**
Define a composition criteria for contracts based on the specification of their assumptions and guarantees. This sets the following requirement on contracts: use the standardized format to specify assumptions and guarantees in contracts (concretely, the property specification patterns). The main objective is to have a clear link between specification parts involved into guarantees and assumptions used when integrating contracts. With property specification patterns, it is more easily to judge about the compatibility of contracts.

—**Implementation**
—Identify applied property patterns in assumptions and related guarantees.

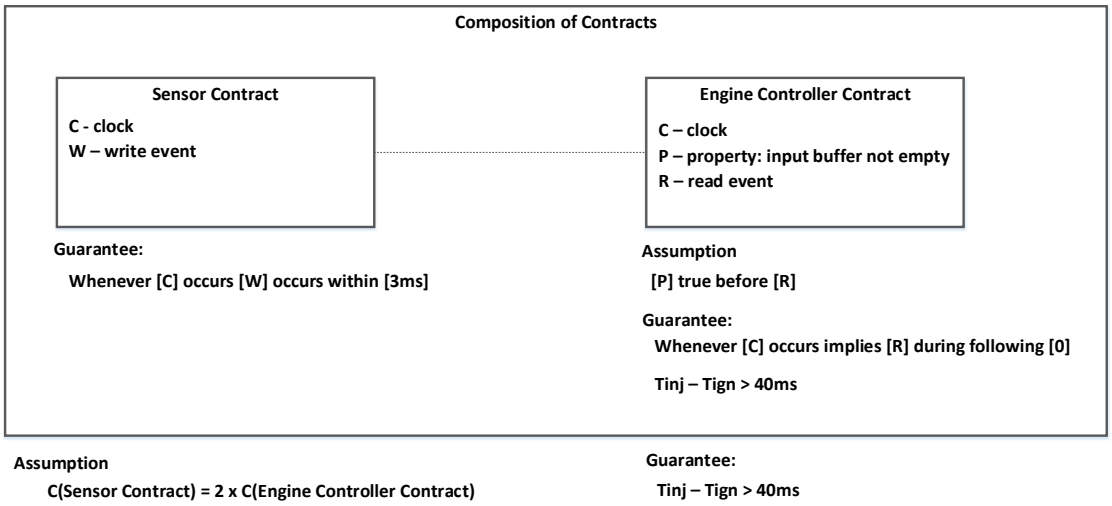


Fig. 4. Contracts of the sensor sub-system and Engine Controller

—Identify elements used in patterns (events, properties, states, data, etc.), or if the property is not described using a pattern format, try to extract the elements using same representations as in property specification patterns.
—Find relation between elements, and verify the compatibility.
—If there is no direct relation, define a global property which links the both contracts. Then, verify the compatibility.

—**Consequences**
(+) Using a structured way on verifying the compatibility between contracts reduces the overhead for the integration (or deciding whether components can be reused).
(−) It often requires manual work and can be extensive when contracts are represented using different specification patterns (like in example from Fig. 4, see description below).

—**Example Resolved**
The only way to satisfy high-level requirement related to timing from the perspective of the Supplier 2 is to define the assumption on providing the sensor data before read operation is active, as illustrated in Fig. 4. Now, the first step of the system integrator is to compose the sensor sub-system with the Engine Controller. We have following definitions of contracts (see Fig. 4):
—Assumption *[P] is true before [R]* states that the buffer that stores the sensor data must not be empty when the event of reading this buffer occurs.
—Guarantee of the sensor sub-system, *Whenever [C] occurs [W] occurs within [13ms]*, states that write event W occurs 13ms after the clock C, which is a periodic event.
Patterns used to describe the assumption and guarantee above are different. Moreover, elements used in patterns are different, so that direct compatibility cannot be concluded. However, it is obvious that the property P is related to the write event W, since W triggers the provision of data to the Engine Controller. Further, it is clearly defined how and when the sensor system provides data. On the other hand, the read event R is also periodic, which is defined in the guarantee of the Engine Controller contract. In order to satisfy the assumption, write event must always happen before R. To ensure this, the system integrator configures the sensor sub-system to operate on a double sampling rate than the Engine Controller. Finally, for correct composition, the system integrator defines a contract on composed sub-system, having now global assumption, as illustrated in Fig. 4.

—**Known Uses**
Some of the notable references that systematically handle the composition criteria:
—Contract-based design [Benveniste et al. 2012]: rules for ensuring compatibility between contracts in order to verify the composition automatically.
—BIP [Basu et al. 2011]: framework to verify composition with regard to behavior of components.
—PLCOpen [PLCopen 2006]: rules and design verification method for compositions with regard to the quality system (distinguishing between safety-related and non-safety-related components and data).

4.6 Aspect

—**Context**
Safety-critical system is under development. The requirements describe the functional and non-functional properties that have to be represented using contracts. There are many system properties which have to be captured within contracts.

—**Problem**
Capturing many properties within a single contract may impact the usability and managing contracts, especially when contracts have to be verified with respect to their compatibility. Forces:
—Contracts may describe various system quality attributes such as safety, security, real-time, availability, etc.
—There are different representations of property specification patterns for assumptions and guarantees when describing behavior of components and when using contracts to define resource constraints for example.

—**Example**
Each of the suppliers from Fig. 1 may define resource constraints on their devices such as memory usage and communication profile, in order to reuse devices for different applications. Further, they get the system properties from the manufacturers, which may have very different nature.

—**Solution**
Use aspects to categorize property types. Each aspect represents a collection of assumptions and guarantees related to particular type of properties. Each contract express properties in certain aspect.

—**Implementation**
When defining contracts, avoid to mix properties of different types.

—**Example Resolved**
Supplier can easily identify requirements and expectations for his system, and similarly the system integrator can easily verify the composition.

—**Known Uses**
Heterogeneous Rich Components [SPEEDS 2010]: different types of properties (real-time, safety, etc.) are grouped into views. Views are parts of contracts here.
Contract-based design [Benveniste et al. 2012]: representing contracts as views.

—**Consequences**
$(+)$ The team can distribute the task of solving the integration issues to members according to their skills in related quality attributes (safety, real-time, etc.).
$(+)$ The team can classify contracts according to their complexity, and define own strategy on how to solve the integration issues. For example, many contracts which describe data semantic of components (i.e. the first type in Sec. 4.1) can be verified automatically, using design tools.
$(-)$ Grouping of contracts may not be precise, if there is no consensus on used aspects.

5. DISCUSSION

From the introduced pattern system it is obvious that for some properties, in particular the ones that describe the behaviour of components, it might be difficult to take decisions about the compatibility between contracts. This

problem gets more challenging with the increasing number of components and required properties, since most of the work has to be done manually. On the other hand, properties that describe data semantics, i.e. components without any clocks, events, or behaviour, have a relatively simple and precise definition, but they alone are not powerful enough to describe all details about functional and extra-functional aspects of components[4].

To apply the proposed pattern system effectively, a trade-off between a level of detail and costs for defining contracts and verifying their compatibility has to be found. Not all components require the behavioral description – for example, the context information for libraries, some operating system services, and similar components which are proven-in-use, can be defined using properties for data semantics, i.e. to describe a tool-chain that was used to generate those components, safety integrity level of a system in which they may operate, processor architecture, and similar. In contrast, components that from a chain of complex interactions, where deadlocks or timing violations may occur, need more complex properties with behavioral descriptions.

Another important aspect that has to be considered here is an extent to which the verification and validation methods defined in safety standards can be supported by contracts and properties. As mentioned in Section 2, standards define recommendations on methods that have to be implemented in order to achieve certain level of quality for a system. The information from contracts can be used here to generate some important input data, scripts, etc. for those verification methods. However, because not all details about context information can be captured for every component, the functional and extra-functional aspects remaining have to be handled later in the system verification and validation phase. The more information is captured in contracts, the lower is the effort to perform the later verification.

6. CONCLUSION

In this paper, we have described a compact pattern system with the aim to address some problems with regard to reuse in safety-critical systems. Currently, these systems follow the guidelines of safety standards to increase the system's quality and reduce the risk of failures. Some of the standards describe also the concepts for reuse, but on higher levels of abstraction so that many details are left to domain experts. The promising approach to help developers to more systematically reuse parts of their system, is a contract-based design. We built our pattern system based on this principle, and we linked it to the low-level property specification patterns, which describe the behavior and data semantic of system parts very precisely. With the link from property specification patterns to the pattern system, the properties of the system parts can be documented up to the higher levels of abstraction more systematically.

7. ACKNOWLEDGMENTS

We would like to thank our shepherd Dietmar Schuetz for reading the paper, analyzing it and for providing us with very constructive and helpful suggestions that made the paper more readable and more easily to understand.

REFERENCES

BASU, A., BENSALEM, S., BOZGA, M., COMBAZ, J., JABER, M., NGUYEN, T.-H., AND SIFAKIS, J. 2011. Rigorous component-based system design using the bip framework. *Software, IEEE 28,* 3, 41–48.

BENVENISTE, A., CAILLAUD, B., NICKOVIC, D., PASSERONE, R., RACLET, J.-B., REINKEMEIER, P., SANGIOVANNI-VINCENTELLI, A., DAMM, W., HENZINGER, T., AND LARSEN, K. 2012. Contracts for Systems Design. Tech. rep., Research Report, NÂř 8147, November 2012, Inria.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA.

BUTZ, H. 2010. Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. Department of Avionic Systems at Airbus Deutschland GmbH Kreetslag 10, D-21129 Hamburg, Germany.

CLARA BENAC EARLE, ELENA GÓMEZ-MARTÍNEZ, STEFANO TONETTA, STEFANO PURI, SILVIA MAZZINI, JEAN LOUIS GILBERT, OLIVIER HACHET, RAMÓN SERNA OLIVER, CECILIA EKELIN, AND KATIUSCA ZEDDA. 2013. Languages for Safety-Certification Related Properties. In *Proc. Work in Progress Session at 39th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA'13).*

[4]Except of data-flow systems like ones modelled using the Matlab Simulink, where all components share the same behaviour.

CRNKOVIC, I. 2002. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA.

EAST-ADL. 2010. EAST-ADL Domain Model Specification. Tech. rep., ATTEST EAST-ADL, Homepage: `http://www.atesst.org/`.

EVANS, E. AND FOWLER, M. 1997. Specifications RTE. Tech. rep., Martin Fowler, Homepage: `http://martinfowler.com/`.

FREY, P. 2010. Case Study: Engine Control Application. Tech. rep., Ulmer Informatik-Berichte, Nr. 2010-03.

GAO, J. Z., TSAO, J., WU, Y., AND JACOB, T. H.-S. 2003. *Testing and Quality Assurance for Component-Based Software*. Artech House, Inc., Norwood, MA, USA.

KELLY, T. P. 2001. Concepts and Principles of Compositional Safety Case Construction. Tech. rep., COMSA/2001/1/1.

KINDEL, O. AND FRIEDRICH, M. 2009. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. dpunkt Verlag; Auflage: 1 (8. Juni 2009).

LÉVÊQUE, T. AND SENTILLES, S. 2011. Refining extra-functional property values in hierarchical component models. In *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*. CBSE '11. ACM, New York, NY, USA, 83–92.

PLCOPEN. 2006. Safety Software - Technical Specification, Part 1: Concepts and Function Blocks. Tech. rep., Technical Committee 5, Version 1.0,Jan 2006.

POHL, K., BÖCKLE, G., AND LINDEN, F. J. V. D. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

SMITH, D. AND SIMPSON, K. 2010. *Safety Critical Systems Handbook: A STRAIGHTFOWARD GUIDE TO FUNCTIONAL SAFETY, IEC 61508 (2010 EDITION) AND RELATED STANDARDS, INCLUDING PROCESS IEC 61511 AND MACHINERY IEC 62061 AND ISO 13849*. Elsevier Science.

SPECPATTERNS. 1998. Specification patterns. Homepage: `http://patterns.projects.cis.ksu.edu/`.

SPEEDS. 2010. Speculative and exploratory design in systems engineering - speeds. Homepage: `http://www.speeds.eu.com/`.

International Journal of Networked and Distributed Computing, Vol. 2, No. 3 (August 2014), 175-188

On Design-time Modelling and Verification of Safety-critical Component-based Systems

Nermin Kajtazovic, Christopher Preschern, Andrea Höller, and Christian Kreiner

Institute for Technical Informatics, Graz University of Technology,
Inffeldgasse 16, 8010 Graz, Austria

E-mail: {nermin.kajtazovic, christopher.preschern, andrea.hoeller, christian.kreiner}@tugraz.at

Abstract

Component-based Software Engineering (CBSE) is currently a key paradigm used for developing safety-critical systems. It provides a fundamental means to master systems complexity, by allowing to design systems parts (i.e., components) for reuse and by allowing to develop those parts independently. One of the main challenges of introducing CBSE in this area is to ensure the integrity of the overall system after building it from individual components, since safety-critical systems require a rigorous development and qualification process to be released for the operation. Although the topic of compositional modelling and verification in the context of component-based systems has been studied intensively in the last decade, there is currently still a lack of tools and methods that can be applied practically and that consider major related systems quality attributes such as usability and scalability.

In this paper, we present a novel approach for design-time modelling and verification of safety-critical systems, based on data semantics of components. We describe the composition, i.e., the systems design, and the underlying properties of components as a Constraint Satisfaction Problem (CSP) and perform the verification by solving that problem. We show that CSP can be successfully applied for the verification of compositions for many types of properties. In our experimental setup we also show how the proposed verification scales with regard to the complexity of different system configurations.

Keywords: component-based systems; safety-critical systems, compositional verification, constraint programming.

1. Introduction

Safety-critical systems are controlling the technical processes in which certain failures may lead to events causing catastrophic consequences for humans and the operating environment. Automotive, railway, and avionics are exemplary domains here, just to name few. In order to make these systems acceptably safe, their hardware/software engineering has to be rigorous and quality-assured.

Currently, rapid and continuous increase of systems complexity represents one of the major challenges when engineering safety-critical systems. The avionics domain for instance has seen an exponential growth of software-implemented functions in the last two decades ([6]), and a similar development has also occurred in other domains with a focus on mass production, such as automotive or biomedical engineering ([16]). In response, many domains have shifted towards using component-based paradigm ([24], [10]). The standards such as the automotive AUTOSAR and IEC 61131/61499 for industrial automation are examples of widely used component systems. This paradigm shift enabled the im-

Published by Atlantis Press

provement in reuse and reduction of costs in development cycles. In contrast to traditional paradigms such as the procedural and the object-oriented programming, in CBSE more attention is given on systems engineering for parts of the system rather than considering the system as a whole, i.e., on developing components. This opens many opportunities for developers and maintainers, such as more precise control and traceability over parts of the system, and possibility on their systematic reuse, which goes beyond the plain add-hoc reuse of code, objects and libraries. In some fields, the modularity of the system structure is utilized to distribute the development across different roles, in order to perform many engineering tasks in parallel. For instance, the automotive manufacturers are supplied by individually developed middleware and devices which can run their applications.

On the other side, the new paradigm also introduced some new issues. One of the major challenges when applying CBSE is to ensure the integrity of the system after building it from reusable parts (components). The source of the problem is that components are often developed in the isolation, and the context in which they shall function is usually not considered in detail. In response, it is very difficult to localize potential faults when components are wired to form a composition – an integrated system ([12]), even when using quality-assured components. The focus of the current research with regard to this problem is to enrich components with properties that characterize their correct behavior for particular context, and in this way to provide a basis for the design-time analysis or verification*of compositions ([8]).

This verification is also the subject of consideration in some current safety standards. For instance, the ISO 26262 standard defines the concept Safety Element out of Context (SEooC), which describes a hardware/software component with necessary information for reuse and integration into an existing system. Similarly, the Reusable Software Components concept has been developed for systems that have to follow the DO-178B standard for avionic software. These concepts both share the same kind of strategy for compositional verification: contract-based design. Each component expresses the assumptions under which it can guarantee to behave correctly. However, the definition of the specific contracts, component properties and validity criteria for the composition is left to the domain experts.

From the viewpoint of the concrete and automated approaches for compositional verification and reasoning, many investigations have focused on behavioural integrity, i.e., they model the behaviour of the components and verify whether the composed behaviours are correctly synchronized ([2]), ([4]). On the other side, compositions are often made based on data semantics shared between components ([5]). Here, the correct behaviour is characterized by describing valid data profiles on component interfaces. In both cases, many properties can be required to describe a single component and therefore scalability of the verification method is crucial here.

In this paper, we present a novel approach for verification of compositions based on the data semantics shared between components.† We transform the modelled composition along with properties into a Constraint Satisfaction Problem (CSP), and perform the verification by solving that problem. To realize this, we provide the following contributions:

- We define a component-based system that allows modelling properties within a complete system hierarchy.
- We define a structural representation of our modelled component-based system as a CSP, which provides us a basis to verify the preservation of properties.
- We realize the process that conducts the transformation of the modelled component-based system into a CSP and its verification automatically.

The CSP is a way to define the decision and optimization problems in the context of Constraint Programming paradigm (CP) ([3]). Using this paradigm for our component-based system, many types of properties can be supported. Also, various parameters that influence the scalability of the verification

*In the remainder of this paper, we use the term *verification* for the static, design-time verification (cf. static analysis ([25])).
†This article is an extended version of our previous work ([14]).

can be controlled (used policy to search for solutions for example). In the end of paper, we discuss the feasibility of the approach with regard to its performance.

The remainder of this paper is organized as follows: Section 2 describes the problem statement more in detail and gives some important requirements with regard to modelling a system. In Section 3 and 4, the proposed approach to systems modelling and verification is described. Section 5 describes the experimental results. A brief overview of relevant related work is given in Section 6. Finally, the concluding remarks are given in Section 7.

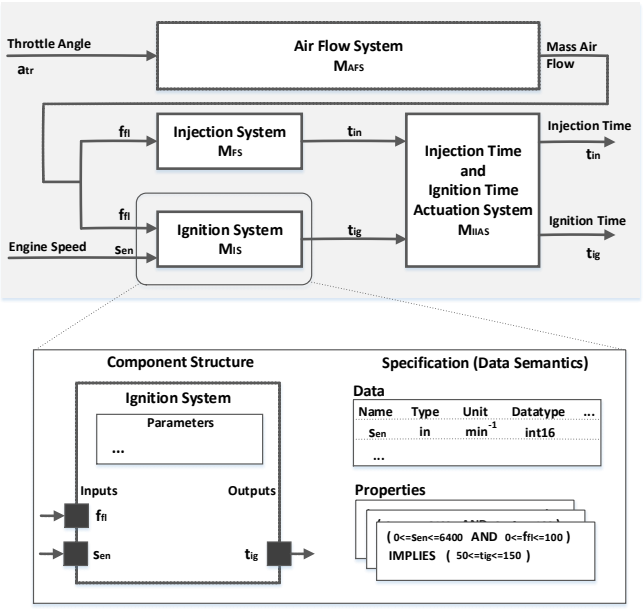


Fig. 1. Motivating example: a component-based system of automotive engine control function, adopted from ([11]) (top), and detailed view of the component Ignition System (structure and specification, bottom).

2. Problem Statement

Properties are an important means to characterize functional and extra-functional aspects of components. Safety, timing and resource budgets are examples here, just to name few ([21]). Recently, they get more and more attention in the safety community, since efficient (an practical) methods for reuse and system composition are crucial in order to reduce costs in development cycles and costs for certification of today's safety-critical systems (i.e., their extensive qualification process). In this section, we give an insight into the main challenges when using properties to verify compositions, and based on these challenges, we outline the main objectives that we handle in this paper.

2.1. Motivating Example

In our work, we address properties that in general describe data semantics. To clarify this, let us consider now the example from Figure 1. The system in this figure shows the composition of four components that form the automotive engine control application on a higher abstraction level. The basic function of this application is to decide when to activate the tasks of the fuel injection and ignition ([11]). To do this, the application takes the sensed values of the air flow volume, current speed and some parameters computed from the driver's pedal position. In a typical automotive development process[‡] the system structure from figure is made based on stepwise decomposition of top-level requirements, having several intermediate steps such as the functional and technical system architecture with several levels in the hierarchy ([20]). Let us assume now that involved components are already developed, eventually for the complete car product line, and are stored in some repository. Let us further assume that we have a top-level requirement with regard to the engine timing for particular car type, which states the following:

The minimal allowed time delay between the task of the fuel injection and ignition shall be greater than 40 ms.

The main contributors to this requirement are software components M_{AFS}, M_{FS}, M_{IS}, M_{IIAS}, and their execution platform (e.g., concrete mapping of components on real-time tasks, task configurations, and other). In order to satisfy this timing property, the developer has to analyze the specification for each component in order to find the influence of the component behaviour on that property. The example of such a specification is given in Figure 1, bot-

[‡]Note that we do not limit our approach to automotive domain.

tom. Here, the context for the component Ignition System is defined in terms of the syntax and semantics related to component inputs, outputs and parameters. With the properties shown below, the concrete behavior can be roughly described – in this example, for certain intervals of inputs, the component can guarantee that the output t_{ig} lies within the interval $[50, 150]$ (note that pseudo syntax is used here). When building compositions based on such properties, the developer has to consider their influence on the remaining, dependent components – in this case, it has to be decided whether the M_{IIAS} component can accept such values of the t_{ig} and what should components M_{FS} and M_{AFS} provide so that higher delay than $40ms$ between t_{ig} and t_{in} can be achieved. This can be very tedious and error prone task when doing it manually, because of the following reasons:

- Many components may be required to build a complete system, depending on their granularity. For example, current automotive systems comprise several hundreds of components, and many of them may depend on each other ([16]).
- Some components that directly influence the safety-critical process are usually certified, i.e., developed according to rigorous rules from safety standards. Because of costs for such a certification, the practice is to develop components for different context and to certify them just once (e.g., to support different engine types in our example). In response, many properties have to be defined for a single component to capture all context information.

The main problem here is how to define and to inter-relate all properties thorough the complete system hierarchy in a way that the preservation of properties of all components can be verified automatically? Another problem is how to complete with such a verification in a "reasonable time"?

2.2. Modelling and Verification Aspects

To narrow the problem statement above, very important prerequisite to structure properties within a system hierarchy consistently is to define basic relations among them. For example, properties of the component M_{IS} are related with properties of the component M_{IIAS}, because of direct connections between their output and input variables. On the other hand, properties of all four components influence the semantics of the mentioned top-level property. We summarize different types of these relations as following:

- *Composition*: hierarchical building of composed properties based on their contained properties (e.g., the top-level timing property is composed of properties contained in components M_{AFS}, M_{IS}, M_{FS} and M_{IIAS}). We discuss this later in more detail.
- *Refinement/abstraction*: properties characterize the component behaviour at certain abstraction level. With refined properties, more specialized behaviours can be described. For example, the property in Figure 1 may include some additional parameters to define conditions for the t_{ig} more precisely.
- *Alternatives*: properties may have alternative representations for different context (e.g., the Injection System component M_{IS} can provide different properties for different engine types).

These relations have to be supported when modelling a component-based system and they have to be considered when such a system has to be verified.

3. System Modelling, Verification and Deployment: An Overview

In this section, we summarize the workflow that integrates the proposed approach for systems modelling and verification. We use this workflow to verify the consistency of systems design, i.e., when the system is initially developed by assembling components, or when changes on that system have to be performed – such as component replacements or changes of the internal systems state, represented in terms of component or systems parameters.

In our previous work ([13]), we described a method on how to change safety-critical systems, with the aim to repair that system in the operation and maintenance phase by replacing malfunctioning software

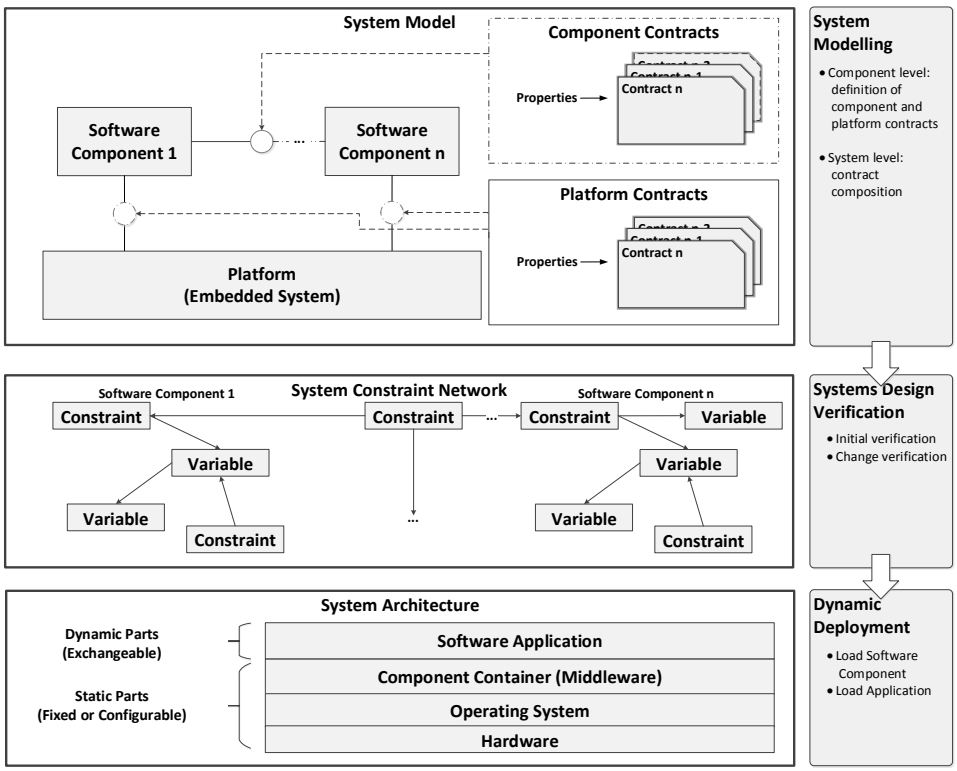


Fig. 2: Application of the proposed modelling and verification – workflow to verify an impact of changes on system integrity ([13]): system modelling using contracts (top), system design verification (middle) and dynamic deployment of software components (bottom)

components or by changing systems configuration at reduced development and maintenance costs. To this end, we defined types of supported changes and properties that have to be considered in the modelling and verification. Further, to allow to change the system in the operation we introduced a runtime support to load software components into a real-time operating systems used for safety applications ([15]). The overall workflow for the modelling, verification and deployment is depicted in Figure 2. In this paper, we focus only on modelling and verification parts of the workflow.

In the first step of the workflow, a model of a system is provided. This model basically captures properties on a level of software components, i.e., (i) to express their behaviour and relationships they have to neighbouring components, and (ii) to express relationships between components and the platform (i.e., an embedded system). Properties are here structured using contracts, which are constructs very similar to system requirements – they express what components shall do (functional) or how they shall be (extra-functional or non-functional), while at same time they define a context in which components have to satisfy those requirements. A very important role of contracts in system design is that they allow for defining specific relationships, so that the information about system integrity expressed through functional and extra-functional requirements can be maintained. Based on this fundamental feature, the impact of changes can be easily estimated and also necessary measures to handle changes can be easily identified. The next step of the workflow deals with the analysis of the system modelled using contracts. Here, a complete system is translated into a so called constraint network – a collection of inter-connected variables and constraints. This network represents contracts in a problem domain using CP. In this way, we are able to analyse whether a modelled system violates any of the contracts. In the same way, we can verify whether changes within a system design eventually require to change requirements.

Finally, the last step of the workflow is an archi-

tectural support to perform changes. To this end, we have realized a dynamic linker that is customized for the use in real-time operating-systems for safety applications. The distinct feature of this linker is that its behaviour is predictable, and the mechanism itself is designed to meet software safety regulations (please refer to ([15]) for more details).

In the following, we describe the modelling and verification parts of this workflow more in detail.

4. Constraint-based Verification

To get a rough image of the proposed approach, we highlight the modelling and verification steps in Figure 3. The input to the verification is a modelled component-based system, enriched with properties, which are structured in contracts – M_{sys} in figure. This model is further transformed into a Constraint Satisfaction Problem (CSP) – CSP_{sys} in figure, which corresponds to the problem domain mentioned in the previous section (we discuss this later). The CSP model is processed by the constraint solver, i.e., a tool to solve the CSPs, in order determine the preservation of all properties in the system. As a result, we get a decision about such a preservation. In addition, we get concrete values of data (i.e., inputs, outputs and parameters), for which properties are preserved. All steps in the process are performed automatically.

In the following, we describe how we defined each model described above. We first give some basic assumptions for our system M_{sys}. Then we describe the main elements of that system, including properties. In the end, we describe its representation as a CSP.

4.1. General: Components and Compositions

In our system, we define a component M as follows:

$$M := \left\langle \Sigma^{in}, \Sigma^{out}, \Sigma^{par}, M_c \right\rangle \quad (1)$$

, where Σ^{in}, Σ^{out}, and Σ^{par} are inputs, outputs and parameters respectively (i.e., Σ-alphabets define input, output and parameter variables in terms of datatypes, values, and some additional attributes), whereby M_c is an optional set of contained components, and is defined according to relation (1). To clarify this, we distinguish between following two types of components:

- *Atomic components*: components that can not be further divided to form hierarchies, i.e., components for which $M_c = \emptyset$. They perform the concrete computation. The Ignition System for example may contain many atomic components, such as integrators, limiters, simple logical elements and other.
- *Composite components*: hierarchical components that may contain one or more atomic and composite components, i.e., $M_c \neq \emptyset$. Note that we use the term *composition* to indicate composite components, which also may represent a complete component-based system (cf. our system in Figure 1).

The component model introduced above is typical for data-flow systems such as the ones modelled in the Matlab Simulink for example. Similar models of component-based systems are used when considering properties for resource budgets ([5]).

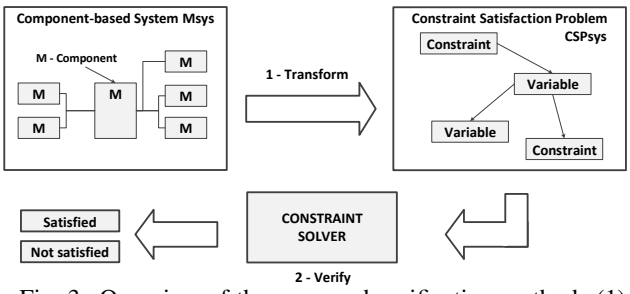


Fig. 3. Overview of the proposed verification method: (1) transformation of the component-based system M_{sys} into the CSP representation CSP_{sys}, (2) verification of the composition CSP_{sys} by solving a CSP.

4.2. Modelling Compositions Enriched with Properties

As illustrated in Figure 1, properties are defined as expressions over component variables. In order to be able to interpret these expressions during the verification, we formulate them in a SMT form[§] each expression can be represented in terms of basic symbols, such as $0, 1, ...s_{en}, ..., +, -, /, ...min$.

[§] Syntax in SMT (Satisfiability Modulo Theories) allows to define advanced expressions, e.g., on integers, reals, etc.

Using this form, various expressions can be supported for our system, including logical, arithmetic, and other. The property from Figure 1 for instance, $(0 \leqslant s_{en} \leqslant 6400) \wedge (0 \leqslant f_{fl} \leqslant 100)$, conforms to the SMT form.

In order to link properties throughout the system hierarchy with regard to three basic relations introduced in Section 2.2, we encapsulate them in assume/guarantee (A/G) contracts. According to the general contract theory in ([5]), a contract C is a tuple of assumption/guarantee pairs, i.e.:

$$C := \langle \Sigma, A, G \rangle \tag{2}$$

, where A and G are expressions over sets of variables Σ. In this way, we can split properties for each component in (a) part that has to be satisfied, i.e., *assumptions*, and (b) part that is guaranteed if assumptions hold, i.e., *guarantees*. For example, the top-level contract C_{II} for our system in Figure 1 guarantees the 40*ms* delay under assumptions that the rotational speed s_{en} and values for the throttle angle a_{tr} are within certain ranges:

$$C_{II} = \begin{cases} \text{variables} & \begin{cases} \text{inputs} & s_{en}, a_{tr} \\ \text{parameters} & - \\ \text{outputs} & t_{in}, t_{ig} \end{cases} \\ \text{types} & s_{en}, a_{tr}, t_{in}, t_{ig} \in \mathbb{N} \\ \text{assumptions} & (0 \leqslant s_{en} \leqslant 6400) \wedge (0 \leqslant a_{tr} \leqslant 100) \\ \text{guarantees} & t_{ig} - t_{in} > 40 \end{cases}$$

Based on this structure, we can link properties between dependent components in a similar way it is done when wiring components using connectors (i.e., links between their input/output variables). Figure 4 shows our example system modelled using contracts. Every component provides certain guarantees which stay in relation to assumptions of dependent components. These components in turn provide guarantees based on their own assumptions, and so forth. In this way, all properties within a system hierarchy can be linked together. In Figure 4, we have also highlighted different types of relations between contracts, required to build such a hierarchy (see Section 2.2). These are:

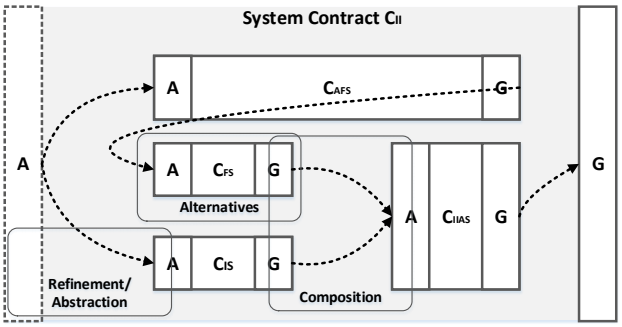


Fig. 4. The Engine Controller system represented using contracts and their basic relations (A – assumptions, G – guarantees, C – contracts).

- *Composition*: two contracts can interact when after connecting their guarantees and assumptions both contracts can function correctly (we discuss this in more detail in Section 4.3). We use the operator $\otimes$ to define a composition ([5]). An example of such relations is shown in Figure 4, where contracts C_{FS}, C_{IS}, and C_{IIAS} form a composite contract, i.e., $((C_{FS} \otimes C_{IS}) \otimes C_{IIAS})$.
- *Refinement/abstraction*: similar to refinement of properties, contracts refine other contracts in terms of refined assumptions and guarantees. We use the operator $\preceq$ for this relation. The top-level contract C_{II} has such a relation with the contained contracts, i.e., $((C_{FS} \otimes C_{IS}) \otimes C_{IIAS}) \preceq C_{II}$. Note that only the relation with the contract C_{IS} is highlighted here.
- *Alternatives*: when designing components for more than one context, each new context is described in a separated contract. Contracts that describe the same property for different context are alternatives. In example in Figure 4, any of contained contracts may have alternatives – here, we just highlighted C_{FS} to indicate that it may have alternative contracts.

Based on definitions for contracts and their relations, we can now define the top-level system/composition contract, C_{sys}, as follows:

$$C_{sys} := (\otimes_{i \in \mathbb{N}} C_i) \tag{3}$$

, i.e., a hierarchical composition of contracts C_i, where C_i represents further composition according to relation (3).

Finally, to relate contracts with components, i.e., the concrete implementations of contracts, we extend the relation (1) as follows:

$$M := \left\langle \Sigma^{in}, \Sigma^{out}, \Sigma^{par}, C_c, M_c \right\rangle \tag{4}$$

, where C_c is a set of contracts that the component M can implement. Based on this relation, any implementation of the C_{sys} contract represents a complete component-based system or a top-level composition. We identify this implementation as M_{sys} and use it later as a basis to define our CSP.

4.3. Ensuring Correctness of Compositions

For our component-based system defined previously, two contracts C_1 and C_2 can form a composition (i.e., can be integrated) when their connected assumptions/guarantees match in the syntax of their variables (i.e., datatypes, units, etc.), and when following holds:

$$G(C_1) \subseteq A(C_2) \tag{5}$$

In other words, the contract C_1 shall not provide values not assumed by the contract C_2. This relation is a basis in our CSP to verify the complete composition.

4.4. Composition as a Constraint Satisfaction Problem

Now, we describe how we define the composition M_{sys} as a CSP. We name our CSP representation of M_{sys} as CSP_{sys}, and define it as follows:

$$CSP_{sys} := \langle X_{CSP}, D_{CSP}, C_{CSP} \rangle \tag{6}$$

, where X_{CSP} is a finite set of variables, D_{CSP} their domains (datatypes, values), and C_{CSP} a set of constraints related to variables and constraints in C_{CSP}. In other words, the CSP represents a network of variables inter-connected with each other using constraints. The constraints set variables in relations using some operators, and in this way they form expressions. Various types of expressions can be used to define constraints (e.g., Boolean, SMT – depending on supported features of the solver). The solution of the CSP_{sys} is a set of values of X_{CSP} for which all constraints C_{CSP} are satisfied. The constraint solver performs the task of finding solutions.

In order to represent the composition M_{sys} in a CSP, we need to map the top-level contract structure ((sub-)contracts, variables, and A/G expressions) into the CSP constructs mentioned above. Important aspects of this representation are CSP definitions for (1) a type system, (2) A/G expressions or properties, (3) the structure of components and contracts and (4) the structure of compositions. We can now turn to these representations.

4.4.1. Type System

The CSP tools, i.e., constraint solvers, usually provide the support for several domains to represent various types of variables. Integers, reals, and sets are examples here, just to name few. In order to avoid type castings between modelled system M_{sys} and CSP_{sys}, we use the same domains for both M_{sys} and CSP_{sys}. Another reason is that the time needed for the constraint solver to solve the CSP strongly depends on a particular domain. For example, there is a significant difference in runtime when dealing with real numbers instead of integers. Therefore, we use integers for both system representations, i.e., M_{sys} and CSP_{sys}.

4.4.2. A/G Expressions (Properties)

Concerning the representation of values of variables in the CSP, limits have to be set on their intervals. The intervals are possible search space for the solver, and can have significant influence on solver's runtime. It is therefore important to limit the variables on smallest possible intervals.

In our CSP_{sys}, each variable which is used in an expression is represented by two CSP variables: one indicating the begin of the interval, another one for the end of that interval. The size of this interval

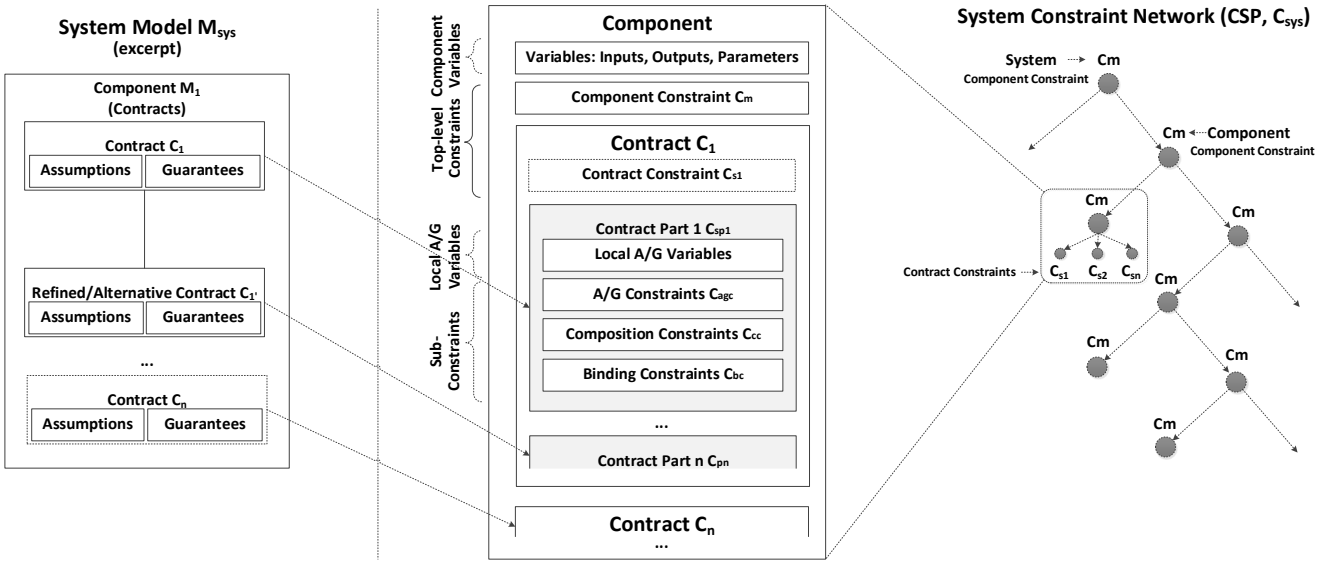


Fig. 5: Representation of a component in CSP: an exemplary component with three contracts (left) and an excerpt of the mapping of contracts to constraints and variables (right)

is determined based on intervals defined in expressions. For example, the variable s_{en} in the expression $(0 \leqslant s_{en} \leqslant 6400)$ is limited on the interval $[0, 6400]$. The reason for using two CSP variables here is that solving the CSP results with not only decision about the correctness of a composition with regard to the relation (5), but it also provides values for which the relation (5) is satisfied. In this way, we can obtain the concrete intervals (instead of just values) for all variables in all contracts (for correct compositions). This information can be useful for example when the composition M_{sys} has many alternative contracts, to observe which of them are identified as correct.

Relations or operations between variables in expressions are represented as constraints. Since both M_{sys} and CSP_{sys} use the SMT syntax for expressions, every operation is represented as a single constraint.

4.4.3. Components

From the perspective of the structural organization, every component is represented in a CSP as a set of variables (inputs, outputs, parameters) from the integer domain, and a set of constraints, which correspond to the contracts implemented by that component (see Figure 5).

Note that we distinguish here between variables used in components, i.e., Σ in relation (4), and variables used in contracts, i.e., Σ in relation (2). Although they are identical, we define separated variables in the CSP for each of them. This means, when a component has two contracts, we have CSP variables for (a) component variables (inputs, outputs and parameters) and (b) CSP variables (inputs, outputs and parameters) for each contract. With this separation of contracts and components, we can identify which contracts are satisfied if the verification succeeds. As mentioned, the constraint solver not only responds with a decision, but it also finds all values of X_{CSP} for which the verification succeeds. Similarly, if the verification fails, the conflicting contracts can be easily identified.

Now we describe how the contracts are defined in a CSP, how they are linked with components, and how the criteria for correctness from relation (5) is represented in a CSP.

4.4.4. Contracts

As shown in Figure 5, each contract is represented as a single top-level constraint C_s. This constraint is further related to a set of local A/G variables (inputs,

outputs, parameters) and a set of sub-constraints. The sub-constraints represent the constraints of the refined/abstracted or alternative contracts (contract parts Cs_p in figure). Because refined/abstracted and alternative contracts do not depend on each other, we define the top-level constraint C_s as follows: $C_s := (\vee_{i\in\mathbb{N}} Cs_{pi})$. In this relation, any contract which can satisfy the relation (5) implies that the top-level contract constraint C_s is satisfied.

As illustrated in Figure 5, every contract consists of the following sub-constraints:

- A/G constraints C_{agc}: constraints related only to local A/G variables. These constraints define the assumptions and guarantees for a contract. They are defined based on A/G expressions in contracts, as described in Section 4.4.2.
- Binding constraints C_{bc}: constraints that link the local A/G variables to the global component variables so that both types of variables get the same values. In this way, we can observe which contracts were satisfied, after successful verification.
- Composition constraints C_{cc}: constraints that integrate the contracts. These constraints express the integration or composition between two contracts, as described in Section 4.3. They link two contracts according to relation (5).

All three top-level constraints have to be satisfied for a contract C_{sp}, i.e., $C_{sp} := (C_{agc} \wedge C_{bc} \wedge C_{cc})$.

Finally, the top-level constraint of a component is satisfied, if all contract constraints C_s are satisfied, i.e., $C_m := (\wedge_{i\in\mathbb{N}} C_{si})$.

4.4.5. System/Composition

The compositions have very similar structure to basic or atomic components. Because they abstract some contracts of the contained components, additional constraints are defined to link these variables. An example of such a composition is given in Figure 4, where assumptions and guarantees of the contract C_{II} are an abstraction of assumptions and guarantees of the contained contracts.

Like atomic components, the complete component-based system M_{sys} is represented in a CSP as a set of variables and constraints. Within this set of constraints, there is a single top-level constraint of the composition C_m which links the complete hierarchy of the sub-constraints and variables discussed previously (the top-level constraint C_m is shown in Figure 5 right). The CSP has a solution only if this top-level constraint is satisfied. Finally, the C_m corresponds to the top-level constraint in the constraint set C_{CSP} from the relation (6).

5. Experimental Results

In the following, we describe the results of the preliminary evaluation and we discuss the performance of our approach.

To conduct the experiment, we used Java-based Choco constraint solver ([7]). In our experiment, we defined the composition M_{sys} as a XML description, which is then used to generate the CSP in memory.

The main goal of this experiment is to show whether the proposed CSP is applicable to solve the composition problems defined with data properties, and for which system configurations. We conduct the experiment by showing how the verification responds with regard to attributes that might have an effect on runtime. These attributes include:

- Components and properties: how the verification scales with regard to number of components and properties, including also the presence of the alternative properties.
- Nature of properties: different properties may require different expressions in the CSP, including operations on fixed values, intervals, or more advanced operations such as ones used to define resource constraints (e.g., sum, min, etc.).

Figure 6 shows the system configuration used to conduct the experiments. The inputs for the verification are provided by the Environment component, which encloses the component-based system under test. All experiments were executed on Intel

i7-3630QM, 4 cores, 2.40GHz.

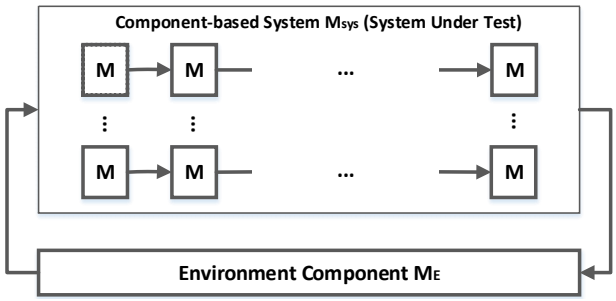


Fig. 6. System configuration used to conduct the experiments (M - component, M_E - environment component).

5.1. Quantitative Results

For this experiment, we performed two measurements. In the first measurement, we show the response time with regard to the number of components, properties and alternative properties, having specified assumptions and guarantees as intervals. Then, in the second measurement, we use the same configurations but with fixed values for expressions. With these two measurements, we are able to observe the limits on modeling the component-based system with regard to number of components, properties, and expressions used to describe the properties.

5.1.1. Measurements

In the first measurement, we execute several thousands of system configurations with the varying number of components and properties. The measurement has two parts. In the first part, we verify the system configurations with the varying number of components, each having varying number of properties but with constant number of assumptions or guarantees (i.e., each component variable is therefore related to only one expression). In the second part, each of the components has varying number of alternative and refined properties, so that many solutions are possible. In this case, each component variable is related to many expressions.

The expressions in the first measurement are defined in a way that always the intervals of the component variables have to be satisfied, and not the fixed values. An example for such expression is given in Section 4.2 for the contract C_{II}, which is satisfied only if the variables s_{en} and a_{tr} are in ranges $[0, 6400]$ and $[0, 100]$ respectively.

For the input test data, i.e., the operands of the assumption and guarantee expressions, we generate the values for each expression randomly, but with the rule that the assumptions are always satisfied. The advantage of performing the positive tests here is to get more clear statement about the runtime of the verification. In both parts of the measurement, we use the relational and logical operations on values.

In the second measurement, we execute the same system configurations as in previous measurement, but this time using the fixed values for component variables.

5.1.2. Observations

First results of the experiments are illustrated in Figure 7. On the left, an excerpt of the results for the first measurement is shown, where the properties have a constant number of assumptions and guarantees. The reason why the verification responds in short time is that each component variable has only one expression (assumption or guarantee constraint, C_{agc}), and it is then immediately instantiated to a value indicated by that expression. The runtime depends in this case therefore on the number of components and properties.

On the right in Figure 7, a scenario that is more likely to occur in practice is shown. Here, each component variable has an increasing number of expressions, and these expressions are alternatives (as mentioned in the description of the measurement). The response time of the verification strongly depends on the number of alternatives, because each of the expressions represents different interval. The solver has to adjust the component variables to adequate intervals, in order to find a solution. Furthermore, since the choice of the particular alternative may influence the choice of the intervals in other connected components, often the backtracks have to

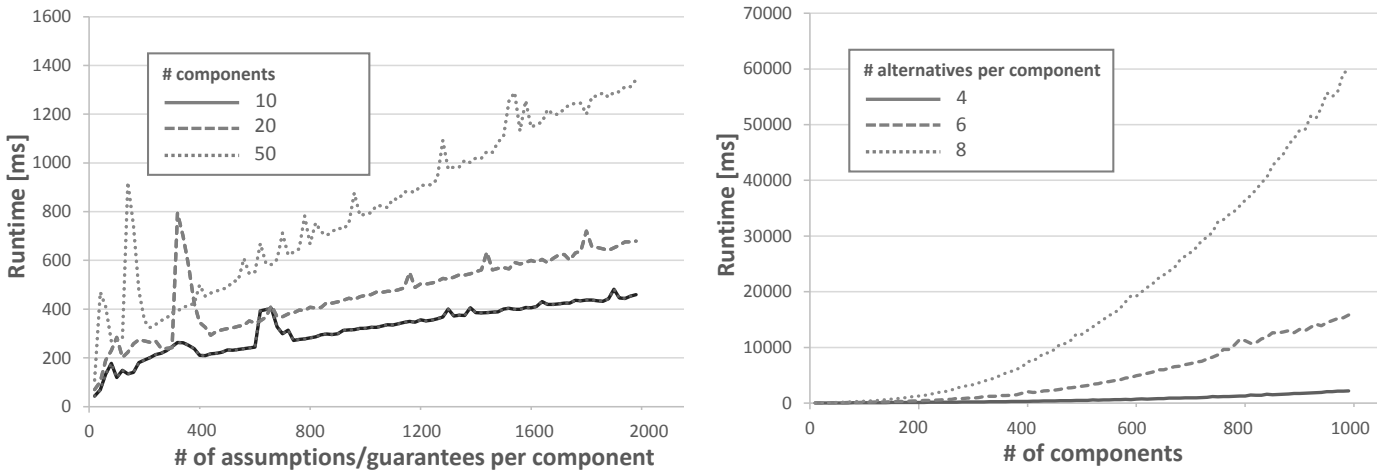


Fig. 7: Experimental results: runtime for system configuration with varying number of assumption/guarantee expressions and components (left) and varying number of components and alternative properties (right)

be done to the state where the constraints were satisfied, which is time consuming.

In the second measurement, we observed very similar results as illustrated in Figure 7 on the left. Having fixed values on component variables, no search has to be performed, but just the constraint verification. For the case where the alternatives are used, more time is required to find a solution, but this time is negligible in contrast to situation when using intervals (i.e., Figure 7, right).

In the end, we summarize our observations with Figure 8. This figure shows the region for which the verification can complete in a ”reasonable time”. We set the limit for this time on 2 minutes, just to get a first feedback about possible configurations for the system under test. To establish this region, we used the system configuration with the worst case in response time, i.e., the one having the alternative properties from the first measurement.

5.2. Qualitative Results: Discussion

Figure 8 shows the worst-case scenario, in which a component-based system is modelled having varying number of assume guarantee expressions.

The verification scales well but for configurations with only few instances of either components or properties. In nowadays automotive systems for example, there are more than 800 software components, that control various technical sub-processes in automobiles ([16]). However, it is still possible to support these configurations, since each such sub-system can be provided to verification independently, and also, not all components are massively interconnected as in Figure 6. For example, the simplified system from Figure 1 is modelled using 13 software components (is just one option to realize that system).

6. Related Work

Now we turn to a brief overview of related studies. We summarize here some relevant articles that handle compositional verification based on data semantics.

Similar problems to those described in our problem statement were identified by Sun et al. ([23]) in their work on verifying the composition of analogue circuits for analogue system design. In their approach, each analogue element (resistor, capacitor, etc.) is characterized by its performance profile and this profile is used to build the contract; that is, for certain values of the inputs the element responds with certain output values. Using contracts made from performance profiles, it was possible to eliminate many integration failures early in the system

design phase. These structural compositions of analogue elements are very similar to the compositions in CBSE. However, the model of Sun et al. only considers connections between elements (horizontal relations).

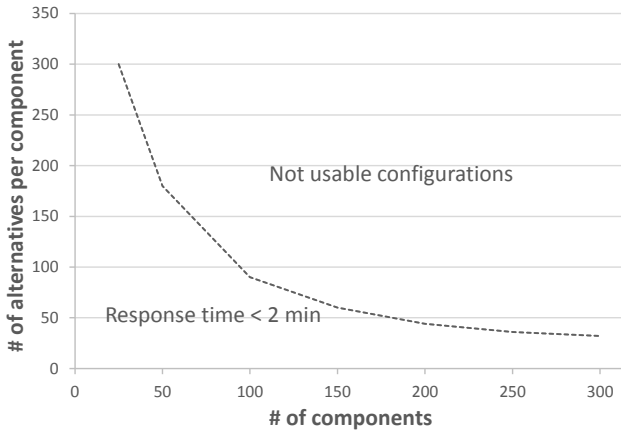


Fig. 8. Region of possible system configurations for which the verification completes within a given time.

Another article describes a runtime framework for dynamic adaptation of safety-critical systems in the automotive domain ([1]). In the event of failures or degradation of quality, the intent is to reconfigure the automotive system while it is operating. In contrast to the previous approach, the compositional verification in this case is based on a common quality type system shared among components. Two components can form a composition only when their interfaces or ports have the compatible type qualities. In this way, wrong type castings between components can be avoided. However, using a type system in our case would just verify the syntax but not the semantics of data (i.e., the concrete values).

A more advanced framework for dynamic adaptation of avionics systems was developed by Montano ([18]). The goal is to adapt the system to new, correct configurations, in case of failures. To perform this, a common quality system defines the contracts between functions and available static resources (e.g., memory consumption, CPU utilization, etc.) and in this way it restricts the possible set of correct configurations. An important aspect of this work is that it demonstrates the CSP approach to solving the composition problem. However, the quality type system only considers static resources, and does not consider contracts between functions. Ultimately, the approach is strongly focused on dynamic adaptation with human-assisted decision making.

In the field of industrial automation, the authors in ([17]) propose the static verification of compositions based on data types of the IEC 61131-3 component model (or standard). This model defines the standard data types but it also allows definition of customized data types (derived from existing ones) and combination of existing data types into complex structures. The authors identified ambiguities in the standard for user-defined data types and defined a proper compatibility criteria. Like the adaptation approach in the automotive domain ([1]), this work considers only a type system. However, the approach verifies not only compositions, but also the use of variables in IEC 61131-related languages.

In the last few years, several research projects have begun to handle the topics of compositional verification ([22]), ([9]), ([19]) by formalizing system models (component models) and languages for specification of contracts. These approaches share many concepts, especially contract-based design and formal behavioural verification of compositions. Although our model is conceptually very similar, it differs in that it considers the data semantics of property values, and it addresses a specific type of component-based systems in which data semantics can be used to express the validity criteria for compositions.

7. Conclusion

In this paper, we presented a method for modelling and verification of compositions in component-based systems. The components modelled here are enriched with properties, which describe the data semantics of components. The novelty of our verification lies in representing the composition along with modelled properties as a Constraint Satisfaction Problem (CSP), which allows us to achieve two important objectives. First, using relational, logical

and more advanced operators on data, many types of properties can be supported. Second, for properties that use basic logical and arithmetic operators, the verification can scale up to several hundreds of components, each of them consisting of few tens of properties, which makes the approach promising for the use in practice.

As part of our ongoing work, we want to characterize the runtime performance based on different types of properties, since they impact the scalability at most. In addition, we also want to investigate other parameters such as solver search policy, solver engine, etc., in order to find best configuration for the verification method.

References

1. Adler, R., Schaefer, I., Trapp, M., Poetzsch, A.: Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. ACM Trans. Embed. Comput. Syst. **10**(2) (2011)
2. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes **26**(5), 109–120 (2001)
3. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)
4. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the bip framework. Software, IEEE **28**(3), 41–48 (2011)
5. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.: Contracts for Systems Design. Tech. rep., Research Report, Nr. 8147, November 2012, Inria (2012)
6. Butz, H.: Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. Dept. of Avionic Systems at Airbus Deutschland, Hamburg, Germany (-)
7. choco Team: choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO, École des Mines de Nantes (2010)
8. Clara Benac Earle: Languages for Safety-Certification Related Properties. In: WIP Session at SEAA'13
9. COMPASS: Compass - comprehensive modelling for advanced systems of systems. Homepage: http://www.compass-research.eu (2011-2014)
10. Crnkovic, I.: Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA (2002)
11. Frey, P.: Case Study: Engine Control Application. Tech. rep., Ulmer Informatik, Nr. 2010-03 (2010)
12. Gössler, G., Sifakis, J.: Composition for component-based modeling. Sci. Comp. Prog. **55** (2005)
13. Kajtazovic, N., Preschern, C., Höller, A., Kreiner, C.: Towards assured dynamic configuration of safety-critical embedded systems. In: Computer Safety, Reliability, and Security, *LNCS*, vol. 8696, pp. 167–179. Springer International Publishing (2014)
14. Kajtazovic, N., Preschern, C., Höller, A., Kreiner, C.: Constraint-based verification of compositions in safety-critical component-based systems. In: SNPD, *Studies in Computational Intelligence*, vol. 569, pp. 113–130. Springer International Publishing (2015)
15. Kajtazovic, N., Preschern, C., Kreiner, C.: A component-based dynamic link support for safety-critical embedded systems. In: 20th IEEE International Conference and Workshops on the Engineering of Computer Based Systems, pp. 92–99 (2013)
16. Kindel, O., Friedrich, M.: Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis. dpunkt Verlag; Auflage: 1 (2009)
17. M., D.S.: Data-type checking of iec61131-3 st and il applications. In: 2012 IEEE 17th Conference on Emerging Technologies Factory Automation (ETFA), pp. 1–8 (2012)
18. Montano, G.: Dynamic reconfiguration of safety-critical systems: Automation and human involvement. PhD Thesis (2011)
19. SAFECER: Safecer - safety certification of software-intensive systems with reusable components. Homepage: http://safecer.eu (2011-2015)
20. Schäuffele, J., Zurawka, T.: Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen. V+T Verlag (2010)
21. Sentilles, S., Štěpán, P., Carlson, J., Crnković, I.: Integration of extra-functional properties in component models. In: Proceedings of the 12th International Symposium on Component-Based Software Engineering, pp. 173–190. Springer-Verlag, Berlin, Heidelberg (2009)
22. SPEEDS: Speculative and exploratory design in systems engineering - speeds. Homepage: http://www.speeds.eu.com (2006-2012)
23. Sun, X., Nuzzo, P., Wu, C.C., Sangiovanni-Vincentelli, A.: Contract-based system-level composition of analog circuits. In: Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE, pp. 605–610 (2009)
24. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley LP Co., Inc., Boston, MA, USA (2002)
25. Tran, E.: Verification/validation/certification. Carnegie Mellon University, 18-849b Dependable Embedded Systems (1999)

Bibliography

[AFPdS11] J.B. Almeida, M.J. Frade, J.S. Pinto, and S.M. de Sousa. *Rigorous Software Development: An Introduction to Program Verification.* UTiCS. Springer London, 2011.

[AIKR13] H. Alemzadeh, R.K. Iyer, Z. Kalbarczyk, and J. Raman. Analysis of safety-critical computer failures in medical devices. *IEEE Security Privacy*, 2013.

[Air14] Airbus. *Airbus Report: Commercial Aviation Accidents 1958-2013 – A Statistical Analysis.* Accidents & Incidents, Design & Certification, Fixed Wing, Safety Management, Aberdeen, Scotland, UK, 2014.

[AK13] Jakob Axelsson and Avenir Kobetski. On the conceptual design of a dynamic component model for reconfigurable autosar systems. *SIGBED Rev.*, 10(4):45–48, December 2013.

[AM07] Vasu Alagar and Mubarak Mohammad. Specification and verification of trustworthy component-based real-time reactive systems. In *Proceedings of the 2007 Conference on Specification and Verification of Component-based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, SAVCBS '07, pages 89–93, New York, NY, USA, 2007. ACM.

[Apt03] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, New York, NY, USA, 2003.

[ASTPH11] Rasmus Adler, Ina Schaefer, Mario Trapp, and Arnd Poetzsch-Heffter. Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(2):20:1–20:39, January 2011.

[Bar13] Clark Barret. From sat to smt : The dpll(t) architecture. Technical report, Stanford University, CS357, 2013.

[BBB+11] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, Thanh-Hung Nguyen, and J. Sifakis. Rigorous component-based system design using the bip framework. *Software, IEEE*, 28(3):41–48, May 2011.

[BC11] Etienne Borde and Jan Carlson. Towards verified synthesis of procom, a component model for real-time embedded systems. In *Proceedings of the*

14th International ACM Sigsoft Symposium on Component Based Software Engineering, CBSE '11, pages 129–138, New York, NY, USA, 2011. ACM.

[BCF+11] E. Borde, J. Carlson, J. Feljan, L. Lednicki, T. Leveque, J. Maras, A. Petricic, and S. Sentilles. Pride - an environment for component-based development of distributed real-time embedded systems. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 351–354, June 2011.

[BCN+12] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim Larsen. Contracts for Systems Design. Technical report, Research Report, Nr. 8147, 2012, Inria, 2012.

[BD00] B. Bruegge and A.H. Dutoit. *Object-oriented software engineering: conquering complex and changing systems*. Prentice Hall, 2000.

[Bel06] Ron Bell. Introduction to iec 61508. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55*, SCS '05, pages 3–12, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, January 2009.

[BLBvV04] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (alma). *J. Syst. Softw.*, 69(1-2):129–147, January 2004.

[BM06] Michael Barr and Anthony Massa. *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, Inc., 2006.

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, September 2005.

[Bre05] Brenda S. Ocker. Software Change Impact Analysis. Research report, Federal Aviation Administration, 2005.

[BS93] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, Jul 1993.

[BSAB14] Klaus Becker, Bernhard Schätz, Michael Armbruster, and Christian Buckl. A formal model for constraint-based deployment calculation and analysis for fault-tolerant systems. In *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 205–219, 2014.

[But08] Henning Butz. Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland. Department of Avionic Systems at Airbus Deutschland GmbH Kreetslag 10, D-21129 Hamburg, Germany, 2008.

[CAD03] I. Crnkovic, U. Asklund, and A.P. Dahlqvist. *Implementing and Integrating Product Data Management and Software Configuration Management*. Artech House computer library. Artech House, Incorporated, 2003.

[CB08] Yunja Choi and Christian Bunse. Towards component-based design and verification of a u-controller. In MichelR.V. Chaudron, Clemens Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering*, volume 5282 of *Lecture Notes in Computer Science*, pages 196–211. Springer Berlin Heidelberg, 2008.

[CC08] Michel Chaudron and Ivica Crnkovic. Component-based software engineering, January 2008.

[Cho10] H. Chockler. Pincette 2014: Validating changes and upgrades in networked software. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 277–277, Oct 2010.

[CHP06] Jan Carlson, John Håkansson, and Paul Pettersson. Saveccm: An analysable component model for real-time systems. *Electronic Notes in Theoretical Computer Science*, 160(0):127 – 140, 2006. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005) Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[CISW05] Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau. The comfort reasoning framework. In Kousha Etessami and SriramK. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 164–169. Springer Berlin Heidelberg, 2005.

[CLC05] I. Crnkovic, S. Larsson, and M.R.V. Chaudron. Component-based development process and component lifecycle. *Journal of Computing and Information Technology – CIT*, 13(4):321–327, 2005.

[Crn02] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[CSVC11] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, Sept 2011.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, FMSP '98, pages 7–15, New York, NY, USA, 1998. ACM.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, September 2001.

[DCL+09] Wei Dong, Chun Chen, Xue Liu, Jiajun Bu, and Yunhao Liu. Dynamic linking and loading in networked embedded systems. In *Mobile Adhoc and Sensor Systems, 2009. MASS '09. IEEE 6th International Conference on*, pages 554–562, Oct 2009.

[dlVPW13] JoseLuis de la Vara and RajwinderKaur Panesar-Walawege. Safetymet: A metamodel for safety standards. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 69–86. Springer Berlin Heidelberg, 2013.

[Dub13] Elena Dubrova. *Fault-Tolerant Design*. Springer, 2013.

[DVM+05] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Böde. Boosting re-use of embedded automotive applications through rich components. *Proceedings of Foundations of Interface Technologies*, 2005, 2005.

[EJ09] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009.

[FAA00] FAA. Guidelines for the Oversight of Software Change Impact Analyses used to Classify Software Changes as Major or Minor. Notice 8110.85, FAA, 2000.

[FAA04] FAA. AC20-148 Reusable Software Components. Tr, FAA, 2004.

[Fer09] Maribel Fernndez. *Models of Computation: An Introduction to Computability Theory*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. Technical report, Martin Fowler Homepage, 2004.

[Fre10] Patrick Frey. Case Study: Engine Control Application. Technical report, Ulmer Informatik-Berichte, Nr. 2010-03, 2010.

[Gru04] Manfred Gruber. Decos – dependable embedded components and systems. Technical report, DECOS consortium, 2004.

[GS05] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, March 2005.

[GW97] D.P. Gluch and C.B. Weinstock. *Workshop on the State of the Practice in Dependably Upgrading Critical Systems: April 16-17, 1997*. Special report, (CMU/SEI-97-SR-014). Carnegie Mellon University, Software Engineering Institute, 1997.

[HC01] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Hen92] Thomas A. Henzinger. Sooner is safer than later. *Inf. Process. Lett.*, 43(3):135–141, September 1992.

[Hen08] Thomas A Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3727–3736, 2008.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall College Division. p. 320. ISBN 978-0-13-458761-5. Retrieved 17 September 2014, New Jersey, USA, 1998.

[HKS+05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM.

[HMC13] Jon Holt, Alvaro Miyazawa, and Ana Cavalcanti. Refinement strategies for sos models. COMPASS Consortium, http://www.compass-research.eu, 2013.

[HMTN+08] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K.-L. Lundback. The rubus component model for resource constrained real-time systems. In *International Symposium on Industrial Embedded Systems, 2008. SIES 2008.*, pages 177–183, June 2008.

[HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, November 2005.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[IEC10a] IEC. Functional safety of electrical/electronic/programmable electronic safety-related systems. *IEC Standard 61508-1:2010*, 2010.

[IEC10b] IEC. Ieee standard for property specification language (psl). *IEEE Std 1850-2010 (Revision of IEEE Std1850-2005)*, pages 1–188, April 2010.

[Jan03] Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[JAY84] F. JAY. *IEEE Standard Dictionary of Electrical and Electronic Terms*. IEEE, 1984.

[Kaz09] M.P. Kazmierkowski. Embedded systems design and verification (zurawski, r.; 2009) [book news]. *Industrial Electronics Magazine, IEEE*, 3(3):56–57, Sept 2009.

[Kel07] Stephen Kell. Rethinking software connectors. In *International Workshop on Synthesis and Analysis of Component Connectors: In Conjunction with the 6th ESEC/FSE Joint Meeting*, SYANCO '07, pages 1–12, New York, NY, USA, 2007. ACM.

[Kel08] Stephen Kell. A survey of practical software adaptation techniques. *jjucs*, 14(13):2110–2157, jul 2008.

[KF09] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis.* dpunkt Verlag; Auflage: 1, 2009.

[KPK13] N. Kajtazovic, C. Preschern, and C. Kreiner. A component-based dynamic link support for safety-critical embedded systems. In *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, pages 92–99, April 2013.

[KRSV13] Antoaneta Kondeva, Daniel Ratiu, Bernhard Schatz, and Sebastian Voss. Seamless model-based development of embedded systems with af3 phoenix. In *Proceedings of the 20th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems*, ECBS '13, pages 212–, Washington, DC, USA, 2013. IEEE Computer Society.

[KSA07] Xu Ke, K. Sierszecki, and C. Angelov. Comdes-ii: A component-based framework for generative development of distributed real-time control systems. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 199–208, Aug 2007.

[KT08] S. Kelly and J.P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley, 2008.

[LAC12] Rikard Land, Mikael Akerholm, and Jan Carlson. Efficient software component reuse in safety-critical systems — an empirical study. In *Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'12, pages 388–399, Berlin, Heidelberg, 2012. Springer-Verlag.

[Leh11] Steffen Lehnert. A review of software change impact analysis. Technical report, Department of Software Systems / Process Informatics, Germany, 2011.

[Lev86] Nancy G. Leveson. Software safety: Why, what, and how. *ACM Comput. Surv.*, 18(2):125–163, June 1986.

[Lev00] J.R. Levine. *Linkers and Loaders*. Operating Systems Series. Morgan Kaufmann, 2000.

[LFR12] Steffen Lehnert, Q. Farooq, and Matthias Riebisch. A taxonomy of change types and its application in software evolution. In *Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on*, pages 98–107, April 2012.

[LNRT12] Kung-Kiu Lau, Keng-Yap Ng, Tauseef Rana, and Cuong M. Tran. Incremental construction of component-based systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '12, pages 41–50, New York, NY, USA, 2012. ACM.

[LPC+13] Kung-Kiu Lau, Marc Pantel, DeJiu Chen, Magnus Persson, Martin Törngren, and Cuong Tran. Component-based development. In Ajitha Rajan and Thomas Wahl, editors, *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*, pages 179–212. Springer Vienna, 2013.

[Mai98] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

[MAP+07] R. Marau, L. Almeida, P. Pedreiras, M.G. Harbour, D. Sangorrin, and J.L. Medina. Integration of a flexible time triggered network in the frescor resource contracting framework. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 1481–1488, Sept 2007.

[Mar10] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[MFRV13] Hélène Martorell, Jean-Charles Fabre, Matthieu Roy, and Régis Valentin. Towards dynamic updates in autosar. In *Computer Safety, Reliability, and Security Lecture Notes in Computer Science*, pages –1–1, 2013.

[Mon11] Giuseppe Montano. Dynamic reconfiguration of safety-critical systems: Automation and human involvement. *PhD Thesis*, 2011.

[MSKC04] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical report, MSU-CSE-04-1, 2004.

[MVFR14] H. Martorell, R. Valentin, J.C. Fabre, and M. Roy. Dynamic software updates vs autosar embedded real-time software and systems. In *ERTS2 - Congress on Embedded Real Time System and Software*, pages –1–1, 2014.

[NKA14] Ze Ni, A. Kobetski, and J. Axelsson. Design and implementation of a dynamic component model for federated autosar systems. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6, June 2014.

[OKB14] Markus Oertel, Omar Kacimi, and Eckard Böde. Proving compliance of implementation models to safety specifications. In Andrea Bondavalli, Andrea Ceccarelli, and Frank Ortmeier, editors, *Computer Safety, Reliability, and Security*, volume 8696 of *Lecture Notes in Computer Science*, pages 97–107. Springer International Publishing, 2014.

[OP92] Gustaf Olsson and Gianguido Piani. *Computer Systems for Automation and Control*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[OR13] Markus Oertel and Achim Rettberg. Reducing re-verification effort by requirement-based change management. In Gunar Schirner, Marcelo Götz, Achim Rettberg, MauroC. Zanella, and FranzJ. Rammig, editors, *Embedded Systems: Design, Analysis and Verification*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 104–115. Springer Berlin Heidelberg, 2013.

[Per09] Magnus Persson. *Adaptive Middleware for Self-Configurable Embedded Real-Time Systems*. Stockholm: KTH, 2009.

[PG13] Steffen Peter and Tony Givargis. Utilizing intervals in component-based design of cyber physical systems. In *Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering*, CSE '13, pages 635–642, Washington, DC, USA, 2013. IEEE Computer Society.

[PHB+09] Roberto Passeron, Imene Ben Hafaiedh, Albert Benveniste, Daniela Cancila, arnaud Cuccuru, Werner Damm, Alberto Ferrari, Sebastien Gerard, Susanne Graf, Bernhard Josko, Leonardo Mangeruca, Thomas Peikenkamp, Alberto Sangiovanni-Vincentelli, and Francois Terrie. Meta–models in europe: Languages, tools and applications. -, 12 2009.

[POS06] J. Polakovic, A.E. Ozcan, and J.-B. Stefani. Building reconfigurable component-based os with think. In *Software Engineering and Advanced Applications, 2006. SEAA '06. 32nd EUROMICRO Conference on*, pages 178–185, Aug 2006.

[PS08] Juraj Polakovic and Jean-Bernard Stefani. Architecting reconfigurable component-based operating systems. *J. Syst. Archit.*, 54(6):562–575, June 2008.

[PSS+13] Nikolaos Priggouris, Adeline Silva, Markus Shawky, Magnus Persson, Vincent Ibanez, Joseph Machrouh, Nicola Meledo, Philippe Baufreton, and JasonMansell Rementeria. The system design life cycle. In Ajitha Rajan and Thomas Wahl, editors, *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*, pages 15–67. Springer Vienna, 2013.

[PTV+13] Paul Pop, Leonidas Tsiopoulos, Sebastian Voss, Oscar Slotosch, Christoph Ficek, Ulrik Nyman, and Alejandra Ruiz. Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the recomp approach. *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems, WICERT*, 2013.

[PV11] Marco Panunzio and Tullio Vardanega. Pitfalls and misconceptions in component-oriented approaches for real-time embedded systems: lessons learned and solutions. *SIGBED Review*, pages 6–13, 2011.

[PV14] Marco Panunzio and Tullio Vardanega. A component-based process with separation of concerns for the development of embedded real-time software systems. *Journal of Systems and Software*, 96(0):105 – 121, 2014.

[Qi 14] Qi Van Eikema Hommes. Applying STAMP Framework to Analyze Automotive Recalls. Research report, The National Transportation Systems Center, 2014.

[Rid12] S. Riddle. Contract-based modelling and analysis technologies for systems-of-systems. In *2012 7th International Conference on System of Systems Engineering (SoSE)*, pages 469–470, July 2012.

[Rie01] L.K. Rierson. Changing safety-critical software. *Aerospace and Electronic Systems Magazine, IEEE*, 16(6):25–30, Jun 2001.

[SBBK12] R. Schneider, W. Brandstaetter, M. Born, and O. Kath. Safety element out of context - a practical approach. In *SAE Technical Paper 2012-01-0033*, 2012.

[Sch06] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[SEI01] SEI. *Dependable Systems Upgrade Initiative*. Special report. Carnegie Mellon University, Software Engineering Institute, 2001.

[SFA04] Kristian Sandstrom, Johan Fredriksson, and Mikael Akerholm. Introducing a component technology for safety critical embedded real-time systems. In Ivica Crnkovic, JudithA. Stafford, HeinzW. Schmidt, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2004.

[SMLA13] C. Sala, R. Moreno, M.C Lomba, and E. Alana. Proceedings of dasia 2013 conference. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2013.

[SNWSV09] Xuening Sun, P. Nuzzo, Chang-Ching Wu, and A. Sangiovanni-Vincentelli. Contract-based system-level composition of analog circuits. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 605–610, July 2009.

[SRH+09] T. Strasser, M. Rooker, I. Hegny, M. Wenger, A. Zoitl, L. Ferrarini, A. Dede, and M. Colla. A research roadmap for model-driven design of embedded systems for automation components. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 564–569, June 2009.

[SS10] D.J. Smith and K.G.L. Simpson. *A Straightforward Guide to Functional Safety, IEC 61508 (2010 Edition) and Related Standards, Including Process IEC 61511 and Machinery IEC 62061 and ISO 13849*. Elsevier Science, 2010.

[Sta01] John A. Stankovic. Vest - a toolset for constructing and analyzing component based embedded systems. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 390–402, London, UK, UK, 2001. Springer-Verlag.

[SvCC09] Séverine Sentilles, Petr Štěpán, Jan Carlson, and Ivica Crnković. Integration of extra-functional properties in component models. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 173–190, Berlin, Heidelberg, 2009. Springer-Verlag.

[SVCDBS04] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and challenges for platform-based design. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 409–414, New York, NY, USA, 2004. ACM.

[SVDP12] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems*. *European Journal of Control*, 18(3):217 – 238, 2012.

[SVM01] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6):23–33, November 2001.

[SVP09] C. Seceleanu, A. Vulgarakis, and P. Pettersson. Remes: A resource model for embedded systems. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 84–94, June 2009.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-oriented Programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[VSC+09] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson. Formal semantics of the procom real-time component model. In *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 478–485, Aug 2009.

[WSO01] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Component-based software engineering. In George T. Heineman and William T. Councill, editors, *Component-based Software Engineering*, chapter Overview of the CORBA Component Model, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[YVNC08] ChenWei Yang, V. Vyatkin, N.C. Nair, and J. Chouinard. The choco constraint programming solver. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, 2008.

[YVNC11] ChenWei Yang, V. Vyatkin, N.C. Nair, and J. Chouinard. Programmable logic for iec 61850 logical nodes by means of iec 61499. In *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, pages 2717–2723, Nov 2011.

[ZOF13] Min Zhang, Kazuhiro Ogata, and Kokichi Futatsugi. Formalization and verification of behavioral correctness of dynamic software updates. *Electronic Notes in Theoretical Computer Science*, 294(0):12 – 23, 2013. Proceedings of the 2013 VSSE Workshop.