CARL
VON
OSSIETZKY
universität OLDENBURG

Fakultät II
Department für Informatik
Eingebettete Hardware-/Software Systeme

A Linear Scaling Change Impact Analysis Based on a Formal Safety Model for Automotive Embedded Systems

Von der Fakultät für Informatik, Wirtschafts und Rechtswissenschaften der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels eines

Doktors der Ingenieurswissenschaften (Dr.-Ing.)

angenommene Dissertation von

Dipl. Inf. Markus Oertel

geboren am 11.05.1984 in Bielefeld

Gutachter:
Prof. Dr. Achim Rettberg

Weitere Gutachter:
Prof. Dr. Bernhard Josko
Prof. Dr. Marcelo Götz

Tag der Disputation: 2. September 2016

Zusammenfassung

Die Kosten für Verifikations- und Validierungsaktivitäten eines sicherheitskritischen eingebetteten Systems können bis zu 70% der gesamten Entwicklungskosten betragen. Da Systeme in der Automobilbranche selten von Grund auf neu entwickelt werden, sondern bestehende Systeme modifiziert und erweitert werden, ist es nachteilig, dass die Auswirkungen von Änderungen auf die Sicherheit des Systems nicht präzise identifiziert werden können. Aus diesem Grund ist häufig eine erneute Verifikation des gesamten Systems notwendig, selbst bei minimalen Änderungen. Weitere Anpassungen, die zusätzlich zur initialen Modifikation durchgeführt werden müssen, um ein operables und sicheres System zu erhalten, sind verantwortlich für diese teure Verifikationsstrategie. Impact-Analyse-Techniken existieren für Software oder im Bereich der Avionik, welche jedoch nicht auf die Automobilbranche übertragbar sind, entweder, weil die Systeme zum Zeitpunkt der Erstellung der Sicherheitskonzepte noch nicht implementiert sind, oder weil die in der Luftfahrtbranche üblichen Isolationsmechanismen und zunächst überdimensionierten Ressourcen fehlen. In dieser Arbeit wird eine neue Impact Analyse vorgestellt, die einen linearen Zusammenhang zwischen dem Verifikationsaufwand und der Größe der Änderung herstellt wobei die Sicherheit des Systems weiterhin garantiert werden kann. Darüberhinaus kann der Ansatz den Entwickler bei der Auswahl weiterer Komponenten zur Kompensation einer Änderung unterstützen. In einigen Situationen können potenzielle Änderungen an einer Implementation durch eine oder mehrere Änderungen an Anforderungen ersetzt werden, um so Kosten einzusparen. Die Impact-Analyse basiert auf einem formalen Sicherheitsmodell, welches Contracts benutzt um das Fehlerfortpflanzungsverhalten und die Sicherheitsmechanismen zu beschreiben. Die Beschreibungssprache wurde konform zu den Anforderungen der ISO 26262, dem aktuellen funktionalen Sicherheitsstandard der Automobilbranche, entwickelt. Im Gegensatz zu existierenden Ansätzen ist ein Abstraktionsmechanismus integriert, welcher eine Top-Down-Entwicklung des Systems ermöglicht. So behalten die bereits erzielten Verifikationsergebnisse ihre Gültigkeit, selbst wenn die Spezifikation und die Architektur im Verlauf der Entwicklung verfeinert werden. Um die Benutzung der Sprache zu vereinfachen werden Templates für die gängigen Sicherheitsmechanismen bereitgestellt und ein Anwendungsleitfaden angeboten. Die Semantik der Sprache ist formal definiert um automatische Analysen zu ermöglichen. Es wird beschrieben, wie die Korrektheit einer Anforderungsverfeinerung sichergestellt werden kann, und wie analysiert wird, ob eine Implementation den Sicherheitsanforderungen entspricht. Die Impact-Analyse wird anhand einer Fallstudie evaluiert um die Ausdrucksstärke und Anwendbarkeit des Ansatzes zu demonstrieren. Damit die Effektivität der neuen Impact Analyse mit dem aktuell praktizierten Ansatz der kompletten Neuverifikation quantitativ verglichen werden kann wurde ein stochastisches Simulationsframework entwickelt. In dieser Simulation werden beide Methoden im Hinblick auf verschiedene Parameter wie die Größe des Systems, die Größe der Änderung, die Genauigkeit der Verifikationsaktivität sowie der resultierende Verifikationsaufwand verglichen. So kann eine präzise Bestimmung des optimalen Wirkungsbereichs durchgeführt werden. Darüber hinaus wird eine Integration der Impact-Analyse in ein verteiltes Entwicklungsszenario präsentiert.

Abstract

The effort for verification and validation activities of safety critical embedded systems may consume up to 70% of the total development costs. Since automotive systems are rarely developed from scratch, but are based on existing systems that are modified, it is unfortunate that the impact of changes on the safety of the system cannot precisely be determined. Therefore, a re-verification of the whole system might be necessary even in case of small changes. Change propagation (i.e., additional necessary modifications to maintain an operable and safe system) that might occur at even distinct parts of the item are responsible for this expensive verification strategy. Impact analysis techniques exist for software or avionic systems but cannot be applied to automotive safety concepts. Either the system is not yet implemented at the safety concept level or the systems components do not provide a sufficient degree of isolation and slack. A new impact analysis is presented in this work that provides a linear relation between the re-verification effort and the size of the change by still guaranteeing the safety of the device. Furthermore the developer is supported in the decisions how to compensate a change by modifying a set of requirements instead of needing to change an implementation. The impact analysis is based on a formal safety model using contracts to express fault containment properties and safety mechanisms. The specification means in this model have been developed to cover the needs from functional safety concepts as stated by the current automotive safety standard ISO 26262. In contrast to other safety specifications we provide an abstraction technique, which allows the development of a system in a top-down manner. Hence, already performed verification results remain valid even if the specification and the architecture of the components are refined. To ease the applicability of the safety specification language, we provide templates for the most common types of safety mechanisms as well as application guidelines. The semantics of the language are formally defined to allow automatic analyses. Therefore, the refinement of safety requirements can be checked as well as the correctness of an implementation with respect to the safety specification. We evaluate the impact analysis on a case study to demonstrate the expressiveness and applicability. To quantify the effectiveness of the new approach compared with the currently used “re-verify all” technique, a stochastic simulation framework has been developed. In the simulation both approaches are compared using multiple parameters such as the size of the system, the size of the change, the accuracy of the verification activities or the resulting verification effort. Hence, a precise determination of the circumstances in which the approach performs best can be determined. In addition we provide an integration of the change impact analysis in an distributed development environment.

Authors Declaration

Material presented within this thesis has previously been published in the following articles:

- Oertel, M. & Rettberg, A. (2013). Reducing re-verification effort by requirement-based change management. In G. Schirner, M. Götz, A. Rettberg, M. Zanella, & F. Rammig (Eds.), *Embedded systems: design, analysis and verification* (Vol. 403, pp. 104–115). IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg

- Oertel, M., Mahdi, A., Böde, E., & Rettberg, A. (2014). Contract-based safety: specification and application guidelines. In *Proceedings of the 1st international workshop on emerging ideas and trends in engineering of cyber-physical systems (eitec 2014)*

- Oertel, M., Gerwinn, S., & Rettberg, A. (2014, July). Simulative evaluation of contract-based change management. In *Industrial informatics (indin), 2014 12th ieee international conference on* (pp. 16–21)

- Oertel, M. & Josko, B. (2012). Interoperable requirements engineering: tool independent specification, validation and impact analysis. In *Artemis technology conference 2012*

- Oertel, M., Kacimi, O., & Böde, E. (2014). Proving compliance of implementation models to safety specifications. In A. Bondavalli, A. Ceccarelli, & F. Ortmeier (Eds.), *Computer safety, reliability, and security* (Vol. 8696, pp. 97–107). Lecture Notes in Computer Science. Springer International Publishing

- Oertel, M., Malot, M., Baumgart, A., Becker, J., Bogusch, R., Farfeleder, S., ... & Rehkop, P. (2013). Requirements engineering. In A. Rajan & T. Wahl (Eds.), *Cesar - cost-efficient methods and processes for safety-relevant embedded systems* (pp. 69–143). Springer Vienna

- Oertel, M., Battram, P., Kacimi, O., Gerwinn, S., & Rettberg, A. (2015). A compositional safety specification using a contract-based design methodology. In W. Leister & N. Regnesentral (Eds.), *Pesaro 2015: the fifth international conference on performance, safety and robustness in complex systems and applications* (pp. 1–7). IARIA (Best Paper Award Winner)

All the work contained within this thesis represents the original contribution of the author and only the indicated resources have been used.

Contents

List of Tables

List of Figures

List of Symbols

$\otimes$	binary contract parallel composition operator
$\preceq$	contract refinement: contract c_1 refines c_2 iff $[\![c_1]\!] \subseteq [\![c_2]\!]$
$\npreceq$	contract not refinement: contract c_1 does not refine c_2 iff $[\![c_1]\!] \nsubseteq [\![c_2]\!]$
$\models$	binary satisfaction operator
$\overleftarrow{I}$	Implemented by: returns the implementation for a provided component
$\overrightarrow{I}$	Implements: returns the component that a implementation is connected to
Φ	Function returning the functional deviation represented by the expression in the promise of a safety contract
$\mathcal{F}$	Function returning the description associated to a malfunction identifier
$\overrightarrow{P}$	part of: function returning the parent component for a provided component
$\overleftarrow{P}$	parts: function returning the child component for a provided component
$\overset{p}{=}$	Port Equivalence: binary relation between ports and signals matching type and name
$\square$	Ports: Function returning the ports including their type (if available) for a given implementation or component
$[\![\,]\!]$	Set of traces for a given specification or implementation

$\overleftarrow{R}$	Refinees: returns the requirements that a given requirement r refined. These are the child requirements in the requirements-breakdown structure
$\overrightarrow{R}$	Refines: returns the requirement that a given requirement r is refining. This is the parent requirement in the requirements-breakdown structure
$\overrightarrow{S}$	Satisfies: returns all requirements that are satisfied by a given component c
$\overleftarrow{S}$	Satisfied by: returns the component that is allocated to a given requirement r
$\mathbb{S}$	Signals: Function returning the signals used in a requirement
V_i	Interface Analysis: function returning the result of the interface check, given two argument combinations, a component and a requirement or a component and an implementation
V_r	Refinement Analysis: function returning the result of the refinement check, given two arguments, the top-level requirements and the set of refined requirements
V_s	Satisfaction Analysis: function returning the result of the satisfaction check, given two arguments, the requirements and the implementation

It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change.

Charles Darwin

CHAPTER 1

Introduction

Multiple times a day we put our lives in the hands of computers that control mechanical or electrical appliances around us. Malfunctions in these *embedded systems* (Lee & Seshia, 2010) introduce risks that are not always obvious. Examples of easily identifiable risks that appeared in recent media are failures in the control software of a plane engine (Kelion, 2015), a car suddenly accelerating at full power (Barr, 2013) or burning batteries of electric vehicles (Madslien, 2011). In contrast, the awareness of the potential danger of an airbag system in a road vehicle is comparatively low, since it is especially designed to prevent harm during a crash. Nevertheless, the unintended deployment of an airbag can easily lead to a loss of control of the car and potentially to the driver's death. Special care has to be taken while designing such *safety critical systems* (Bozzano & Villafiorita, 2011; Storey, 1996) to prevent situations in which individuals could be harmed. It can never be ensured that a system is *safe* under all circumstances, but multiple techniques exist to establish a certain degree of confidence in the safety of a device. One major aspect is an intensive analysis and testing procedure of the system, also in the final context of use, to ensure, that the systems (and also its components) behave exactly as specified. Furthermore, since even the best quality assurance process cannot guarantee absence of problems in design or material of system elements, the potential malfunctions of the used components need to be taken into account while designing the system (Ye & Kelly, 2004). Hence, strategies to detect and react to these malfunctions need to be integrated into the system. Safety standards like ISO 26262 (2011) require these considerations to be noted down in a *safety concept*, which needs to be analyzed with respect to various properties like completeness, consistency and the ability to prevent harm to involved individuals. The most common techniques are summarized in safety standards, which are often domain specific, such as the DO 178C (2011) as well as ARP 4761 (1996) for the avionic domain or EN 50129 (2003) for rail. The standards recommend applicable analysis methods, testing techniques and development processes. Although the individual requirements differ from domain to domain, they all seek to drastically increase the effort spent on analysis and testing the more critical a potential harm is classified.

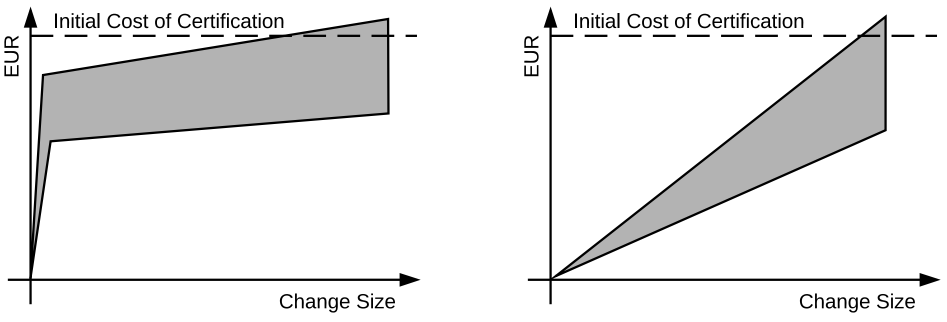
1.1 Motivation

The effort for verification and validation (V&V) of safety critical systems can consume more than 75% (Laprie, 1994) of the whole development costs. Therefore, special care has to be taken to avoid individual V&V activities being executed multiple times. Nevertheless, changes to requirements occur frequently (Terwiesch & Loch, 1999; Fricke, Gebhard, Negele, & Igenbergs, 2000) during the development of a system. One type of changes affects the current running development resulting from requests of the customer, technological evolution or competitors (Fricke et al., 2000). The other type of changes stems from the fact that system are very seldom developed from scratch and instead existing products are modified or parts are re-used. These changes are an important factor for the overall verification and validation effort, since after changes often the whole system needs to be re-verified (Nicholson, Conmy, Bate, & McDermid, 2000; Fenn et al., 2007). This observation holds even though studies have shown that most companies have implemented dedicated change management processes (Huang & Mak, 1999). Since it is common for some subdomain of the automotive industry to change only 10% of the functionality from one generation to its next (Broy, 2006), this complete re-verification results in tremendously high costs and suggests therefore a high potential of savings in verification effort.

Still, this effort currently seems necessary since a change in a system very often requires additional changes to maintain an operational item. This effect is called change propagation (Eckert, Clarkson, & Zanker, 2004; Dick, 2005) or ripple effect (Bohner, 2002; Bohner, 1996) or also a change snowball-effect (Terwiesch & Loch, 1999). The main reason for a full re-verification of the whole system is, that even for experienced engineers it is impossible to be sure that after performing the initial change and a set or corrective measures, that none of the other components of the system is affected (Clarkson, Simons, & Eckert, 2004). Studies have also shown (Ciolkowski, Laitenberger, & Biffl, 2003) that reviews are often not performed in a systematic way and design faults and associated necessary changes are not identified.

This re-verification of the whole system has been illustrated by Fenn et al. (2007) in Figure 1.1. The verification activities necessary for the certification of a system is currently independent from the size of the change. A linear relation between the size of the change and the verification effort is the goal to be reached in the future.

Also Espinoza, Ruiz, Sabetzadeh, and Panaroni (2011) discovered that the monolithic and process oriented structure of the safety cases required by nearly all domain-specific safety standards may require an entire re-certification of the system after changes. Hence, two mostly separately discussed activities, namely *Impact Analysis Techniques* and *Modular System Specifications* need to be integrated. Impact analysis techniques (Arnold & Bohner, 1993; Lehnert, 2011) are meant to identify the affected changes or give an estimate of the adaptation costs before the change is actually implemented. Various solution proposals for source code changes (Ryder & Tip, 2001; Gallagher & Lyle, 1991;



(a) Current practice **(b)** Goal for the future

Figure 1.1 – Relation between the size of the change and the certification effort, according to Fenn et al. (2007)

Law & Rothermel, 2003), avionic architectures (Nicholson et al., 2000) or generic system models (Lock & Kotonya, 1999) exist. However, none of the existing techniques could be applied to identify change effects in automotive safety concepts.

It is the focus of this thesis to overcome these shortcomings, as will be detailed in the next section.

1.2 Scientific Question and Success Criteria

In contrast to the existing technologies, a solution to the following research question shall be developed:

> *How to achieve a linear relation between the size of the change and the re-verification effort while still guaranteeing the overall safety of the system?*

Based on the scientific question the following success criteria have been defined:

1. The effort to determine that a system is still safe after a change is incorporated has a linear relation to the number of development artifacts that have been changed.

2. The confidence in the safety of the system after the change is incorporated is identical or higher compared with the current practiced approach.

3. The support of the engineer during the adaptation of the system as well as all used analysis techniques are fully automated.

4. The developed impact analysis approach is easy to apply in practice. In particular guidance for the engineers is available.

5. The approach is in line with the current automotive safety standard ISO 26262.

1.3 Basic Idea and Contribution

The need for a complete re-certification after changes on safety critical systems stems from the inability to precisely predict the effects of changes on the rest of the system.

Multiple types of impact analyzes exist that try to identify a set of affected components. Two basic problems prevent this information from being used efficiently in the re-certification process of safety critical automotive systems: First, the set of affected components has a magnitude of the size of the whole system. This is mainly caused by the use of tracelinks to determine dependencies between system artifacts. Hence, the resulting set is an over-approximation of the really affected system elements. The second problem refers to approaches that try to limit the set of components that need to be re-verified. Either the probability of false negatives is too high to argue that the system is sufficiently safe if only the detected elements are considered for re-verification, or, the techniques are just applicable for source-code and address compilability instead of correct execution.

To overcome these limitations we developed an impact analysis that is based on the semantics of formalized requirements. I.e., the intended behavior of a component indicates if a change propagates to other components, rather than predefined information such as tracelinks or data flow. To argue on the safety of the system, the fault propagation behavior is captured in a formal notation. Contracts, an assume-guarantee approach, is used as an underlying theory for compositionality. In this thesis algorithms for detecting change propagation using contract theory are presented together with optimization strategies to reduce verification effort during the selection of compensation candidates. Furthermore, existing formal languages for the description of failure propagation are analyzed and extended to the needs of the impact analysis. This extension results in the first safety specification language providing means for abstraction, allowing a refinement of requirements without the problem of invalidating already gained verification results. Guidelines and contract templates to describe safety concepts are presented to enable engineers to specify systems in an ISO 26262 compliant way without detailed knowledge of formal methods. Analysis techniques have been developed that automate the verification activities performed during the impact analysis process. This encompasses the analysis of refinement properties and the compliance of an implementation to its requirements. A Prototype tooling is presented. The evaluation of the approach has been performed in an probabilistic simulation environment identifying the relevant parameters and their value range in which the approach outperforms the state of the art technique. Furthermore the impact analysis has been integrated in an OSLC-based tool landscape to demonstrate how an automated change impact analysis can be used in distributed development environments.

1.4 Assumptions and Scope of the Work

To develop an impact analysis applicable for all development stages of safety critical systems exceeds by far the scope of a single dissertation. Therefore, the scope has been limited to a well defined subset.

This work employs a system structure based on *Perspectives* and *Aspects* (see fundamentals in Section 2.2). This structure is particularly useful to determine the assumptions and the scope this work. Perspectives represent different structural stages of a system that correspond to individual engineering challenges. For instance, there exists a *functional* perspective a *technical* perspective or a *geometrical* perspective. Aspects represent the different behavioral views, such as *safety*, *timing*, *functional behavior* or *heat*.

The scope of this thesis is confined to the safety aspect with an established link to the functional behavior aspect. Hence, the impact within the safety aspect only is calculated, while detailing the interface to the functional behavior aspect of the system. Since many more aspects might be relevant for a system, it is the assumption that they can be also modeled in a way compatible with the safety specification. Since we choose contracts as the most suitable specification mechanism to support impact analyses (see Section 3.1), compatibility is in this case given as a formal, contract-based aspect definition. Currently many different aspects are being developed in such a manner: the real-time aspect has been extensively worked on by Reinkemeier, Stierand, Rehkop, and Henkler (2011) as well as Gezgin, Weber, and Oertel (2014) providing a specification language integrated in contracts to express timing requirements. Power aspects of a system, detailing the energy consumption and leakage of components have been presented recently by Nitsche, Gruttner, and Nebel (2013). However, the research focused in the last years on functional requirements, stating the dependencies and values of signals in a system, for example, by Rajan and Wahl (2013) or Mitschke et al. (2010). Additional aspects, such as geometric installations (Baumgart, 2013) and electro magnetic interference (Baumgart, Hörmaier, & Deuter, 2014), have started to be investigated. Still, the safety aspect is not yet developed at a stage that it could be used for conducting a change impact analysis (for existing techniques see Section 4.1.3). To be able to analyze safety critical systems, the safety aspect is detailed in this work with all necessary requirements to be usable for a change impact analysis. Nevertheless, not all aspects are relevant for all systems. For example, it is not necessary to consider the heat aspect for non-actuating low power devices like rain sensors. Hence, the assumption that all relevant aspects of a system can be represented using contracts seems valid for the future, even if at the current point in time some aspects are still not fully elaborated.

The perspective has been limited to the logical one. In this perspective it is not distinguished between hardware and software, the system is developed conceptually and higher level requirements are stated. Nevertheless, from the view of current safety standards (see introduction to ISO 26262 in Section 2.3) this system representation is essential for the safe operation of a system, since the functional safety concept expressed in this perspective defines the system-wide fault mitigation and degradation concepts. Since at this time implementation dependent aspects like heat, electro magnetic compatibility (EMC) or geometric installation are not relevant the system can be fully described by the already well-defined aspects. Therefore, within the defined scope, the change impact analysis will deliver a semantically accurate determination of all development artifacts affected by a change. The application of the approach for technical perspectives is dependent upon the used aspects and can therefore not be answered in general.

To support the engineer in reacting faster and more accurately on changes in requirements or implementations it is essential to give feedback to the user in a reasonable time frame. Hence, automation of the identification of the affected elements that need additional corrective measures is necessary. Still, since all of the automated analyses we provide are based on model checking technology (Baier & Katoen, 2008; Clarke, Grumberg, & Peled, 1999), there might be situations in which the analysis is running for a very long time. However, as many researchers are exclusively working on improving model checkers, this task was intentionally excluded from this thesis.

Since there exist various ways to approach changes in systems (see Section 3.1) it is worth mentioning that this work does not aim to improve the extendability or maintainability of a system. Also, it is the goal to develop an online approach detecting impacts of changes, we do not want to estimate the effort for a change in advance to any engineering activity.

1.5 Terms and Definitions

This thesis connects multiple research and application areas such as systems engineering, requirements engineering, formal methods and also application of safety standards. Since many terms are used differently among these domains, we briefly introduce the most important terms and topics:

1.5.1 Embedded Systems, Models and Safety

This thesis deals with changes in *embedded systems*, which are defined by Marwedel and Wehmeyer (2007) as "information processing systems that are embedded into a larger product." While this larger product is rather unspecific, Lee and Seshia (2010) clarify that "embedded computers and networks monitor and control the physical process, usually with feedback loops where physical processes affect computations and vice versa." By the coupling to physical processes embedded systems have an immediate impact on the environment and are therefore often considered as *safety-critical* systems (Marwedel & Wehmeyer, 2007). Hence, the property *safety*, "the degree to which accidental harm is prevented, detected and reacted to" (Firesmith, 2003), is a strong influencing factor of the systems design. This definition, as it is formulated similarly by Storey (1996), Avizienis, Laprie, Randell, and Landwehr (2004) or Nancy (1995), extends the definition by Lamport (1977) and Alpern and Schneider (1985), defining a safety property by a set of states of a system that shall not occur. Domain specific safety standards like the ISO 26262 (2011) (for an introduction see Section 2.3), ARP 4761 1996 or DO178c 2011 give guidance on how to analyze the potential risks of a system and how to prevent them.

To avoid faults in the design of safety critical systems it is helpful to use models to structure the system for a better understanding and avoid ambiguities in natural language descriptions (Pohl, Hönninger, Achatz, & Broy, 2012). This type of systems development is called a *model-based design* process. The models can have well-defined semantics, and code generators can exist to produce the code based on the model. The models are described by so-called *meta-models* that describe the language, that is, the "building blocks," that can be used in the model (Atkinson & Kühne, 2001). There exist meta-models for various aspects of the system design, ranging from high-level system description languages like SysML (OMG SysML, 2012) to low-level control loop descriptions like Targetlink[1]. Models that are capable of describing the actual behavior of a component are called *behavioral models* (DoDAF/DM2 2.02, 2010). *Structural models* represent the structure and system without detailing the explicit behavior. Schaetz classifies these models as *product models* (Schätz, Pretschner, Huber, & Philipps, 2002), since they describe entities of the item to be developed or its environment. In contrast, *process models* describe the development process and contain the activities and work products that need to be produced. The benefits of using models in the dvelopment process is to gain tool support while generating code, generating test cases, performing early verification and validation activities or virtual integration testing (Baumgart et al., 2011).

1.5.2 Fault, Error and Failure

All three terms describe conditions of a system that are not intended. A *fault* is defined by the ISO 26262 as an "abnormal condition that can cause an element or an item to fail" (ISO 26262, 2011). Bozzano states this abnormal condition more precisely as "a defect or an anomaly in an item or a system" (Bozzano & Villafiorita, 2011). Faults can occur during design time, by making a mistake in the development of hardware or software components. These faults are called *systematic faults* (Storey, 1996). Faults can also occur during the operation of the system, if a hardware component stops working as expected. These *random faults* may have various reasons, such as aging or material defects. Multiple safety standards agree, that random faults may only occur in hardware components (Baufreton et al., 2010). Faults can be further classified according to their persistence (Avizienis et al., 2004): A fault is considered as *permanent* if its presence is continuous in time, a fault is considered as *transient* if its presence is bounded in time, that is, the fault occurs and disappears. If the fault occurs and disappears frequently, the fault is called *intermittent* (ISO 26262, 2011). Koren and Krishna (2007) calls this behavior oscillation of a fault. A typical example is a loose electrical connection.

A fault may become apparent by leading to an *error*, the "discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition"(ISO 26262, 2011). For example, a software bug in a rarely used routine could be in the system for a very long time (potentially for the whole lifetime of the system) until that software part is executed and a wrong value is calculated (Storey, 1996).

This deviation from the intended behavior could lead to a failure, namely the "inability to perform its intended function" (Bozzano & Villafiorita, 2011). Nevertheless, an error does not necessarily lead to a failure, since the fault could be detected (see Section 1.5.3) and appropriate countermeasures could be started. A mechanism to prevent an error

[1]https://www.dspace.com

from becoming a failure is redundancy.

1.5.3 Fault Avoidance, Fault Removal and Fault Tolerance

It is unavoidable that systematic or random faults are present in the system. Hence, techniques are needed to handle faults during design time or during run time. Storey (1996) and Bozzano and Villafiorita (2011) identified four different types of techniques: *fault avoidance*, *fault removal*, *fault detection* and *fault tolerance*. Although newer classifications exist (Avizienis et al., 2004) that define fault detection and *fault correction* as subactivities of fault tolerance, we stick to Storey's classification since the ISO 26262 explicitly distinguishes between fault detection and fault tolerance.

Fault avoidance techniques try to improve the quality of the development process to prevent faults from being introduced by, for example, usage of model-based techniques or formal system specification, and thereby avoid ambiguities of natural language descriptions. These techniques are called *fault prevention* by Avizienis et al. (2004). Fault removal techniques are still applied at design time, but try to identify faults in the system (e.g., by testing or formal verification) and then remove the faults from the design. In contrast to fault removal techniques, which target only systematic faults in the design of the system, fault detection techniques also allow identification of random faults during runtime of the system to activate appropriate countermeasures in order to avoid failures. Classical fault detection techniques include memory checks, consistency checks or watchdogs. Fault tolerance mechanisms keep the system correctly working, even in case of faults that occur (Avizienis et al., 2004). Fault tolerance mechanisms have a hypothesis under which a mechanism is able to handle a fault correctly. Examples for this hypothesis are the assumption of a maximum number of simultaneous occurring faults in the system or the assumption that some components, like the voting component, do not fail at all. Frequently, both assumptions are used in combination (Bozzano & Villafiorita, 2011). Fault tolerance mechanisms often rely on detection techniques to, for example, switch to another channel and deactivate the defective one. A typical fault tolerance mechanism is triple modular redundancy (TMR) (Von Neumann, 1956; Moore & Shannon, 1956).

1.5.4 Verification and Validation

"Verification is the process of determining that a system, or module, meets its specification and validation is the process of determining that a system is appropriate for its purpose" (Storey, 1996). Verification and validation techniques can be categorized by multiple criteria. *Static analysis methods* analyze the system without executing it. Classical walk-throughs or reviews, which can be used both for validation and verification, as well as automated techniques such as static-code analysis to determine execution times (Shaw, 1989; Gustafsson, Ermedahl, Sandberg, & Lisper, 2006) belong to this class of analyses. Another popular static analysis method for verification is *model checking* (Clarke et al., 1999; Baier & Katoen, 2008), which uses a model of the system, typically represented as a state-machine or even JAVA-code (Visser, Havelund, Brat, Park, & Lerda, 2003), to prove that a property (for example, a requirement) holds. *Dynamic analysis methods* execute the system, or parts of it, to judge the correctness of the results to given stimuli. If stimuli are passed to the final system or system part, this activity is called *testing*. If a model of the system is used, the activity is called *simulation* (Bozzano & Villafiorita, 2011).

1.5.5 Change and Configuration Management

Change management is applied in many domains to implement new processes and structures in a company or react to new or changing market situations. This work is concerned with *engineering change*, "an alteration made to parts, drawings or software that have already been released during the product design process" (Jarratt, Eckert, Caldwell, & Clarkson, 2011). Although some authors disagree (Wright, 1997), we understand the release in the development process as an agreement to having a design stage complete to continue with the preceding ones, rather than a release for production. In particular, we assume that a release of one design stage, for example, completing the functional safety concept (see Section 2.3), includes the successful completion of all verification and validation activities required for that particular stage.

The activities of handling changes are subsumed under the term *configuration management* and detailed in standards like ISO 10007 (2003) (Guidelines for configuration management) or ISO/IEC 12207 (2008) (Software life cycle processes). A *Configuration item* is "an aggregation of work products that is designated for configuration management and treated as a single entity in the configuration management process" (CMMI Product Team, 2010). Hence, it is the objective of configuration management to identify and describe the configuration items, to control changes that are performed to the configuration items, to "record and report change processing and implementation status, and verify compliance with specified requirements" (CMMI Product Team, 2010). It is an essential result of a configuration management process to define *baselines*. A basline is a "specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures" (ISO/IEC 12207, 2008). Often a baseline is considered as the configuration information at a specific time. Before implementing a change an impact analysis shall be carried out, to assess the potential conflicts with other configuration items (CMMI Product Team, 2010; ISO 26262, 2011). In this work the term impact analysis is extended from an analysis that is carried out before the change is implemented to an analysis that is performed constantly during the change implementation process to identify only the affected verification and validation activities.

1.6 Outline

A more in-depth overview of the fundamental theories will be given in Chapter 2. In particular, this includes the principles of designing a system by contracts, using perspectives and aspects, as well as the fundamentals of the higher level parts of the ISO 26262.

We chose a two-stage approach for the development of the new impact analysis. First, we review the existing impact analysis techniques and determine the shortcomings that prevent them from being used for automotive safety concepts in Chapter 3. Based on this analysis a new change impact process is developed that is able to scale linearly with the size of the change. Furthermore, we describe how the engineers applying the new approach can be supported by the analysis in finding efficient ways to deal with change propagation, which were very difficult to detect without automation support. Based on this process we state requirements on a formal system model that need to be fulfilled in order to support the change impact process.

In the second stage, this formal model capturing the safety concept is described in Chapter 4. After analyzing the existing safety modeling approaches we select the most appropriate technique and extend it with the still missing features. In particular, this is the introduction of abstraction techniques to safety specifications, to support bottom-up as well as top-down design processes, and support for multiple possible malfunctions on functional signals. Furthermore we present solutions of how to automatically perform the needed automatic analysis techniques evaluating refinement and satisfaction properties of the safety model. Furthermore, specification templates and guidelines for building these models are provided.

In Chapter 5 we evaluate the approach in two different ways. First, the approach is applied to its full extend to an real-life example. In this example the specification mechanisms and change procedures are demonstrated. Second, to allow a quantitative comparison to the current state of the art technique, a simulation framework has been developed to prove the gain in efficiency of the new approach. Furthermore, multiple tools have been developed to prove the applicability and implement ability of the previously theoretically described approach. These tools are discussed briefly in Chapter 6. It is not the intention to elaborate on the development of the tools, rather on the application scenarios and discovered challenges while deploying a change impact analysis service in a distributed development environment. Final conclusions and impact of the presented results are discussed in Chapter 7.

CHAPTER 2

Fundamentals

This work does rely on some fundamental concepts and theories. Some specification concepts and the operators for composition are based on the contract-based design principle. Therefore, the basic concepts and definitions of contracts are introduced in section 2.1. Contracts are strongly connected with a structuring of the system in so called perspectives and aspects. In this work we focus on one particular aspect, namely safety, and the logical perspective. Hence, it is necessary to understand this structure, which is described in section 2.2. The targeted application of the presented specification mechanisms in Chapter 4 is a formalization of the functional safety concept of the ISO 26262. Therefore, a short overview of this automotive functional safety standard is given in section 2.3 with a focus on the requirements regarding the safety concepts.

2.1 Contract-based Design

Compositional reasoning strives for structuring and simplifying verification efforts by exploiting the particular structure of the system under verification, as given by the functional architecture and its different components forming the building block of a complex system. The analysis of such an integration of system components has become ever more complicated, caused by a development shift in the supply chains from an integration on element level by the OEM towards an integration of whole systems. To tackle this problem, structured specifications of the components to be integrated are necessary.

Contracts provide such structure by providing dedicated component specifications separating requirements into an *assumption*, which describes the expected properties of the environment, and a *guarantee*, which describes the desired behavior of the component under analysis should the assumption be met by the operational context, *i.e.,* the environment. This separation allows the building of a sound theory that enables reasoning in a formal way about the composition of systems. Contracts, belonging to the class of assume-guarantee reasoning techniques (Henzinger, Qadeer, & Rajamani, 1998), are a widely adopted approach for compositional verification (de Roever, 1998; Peng & Tahar, 1998).

The origin of such modular, contract-like specifications can be traced all the way back to pre- and post-conditions of sequential program snippets, such as Hoare triplets (Hoare, 1969), yet the most influential step towards their adoption as design contracts in software engineering can be found in Bertrand Meyer's work (Mandrioli & Meyer, 1992; Meyer, 1992) related to the programming language Eiffel. Driven by European projects like SPEEDS[1], the focus of contracts has recently shifted from the description of pure software systems to a holistic systems engineering approach (Benveniste et al., 2008; Damm, 2005). Many different specializations and extensions have been presented, for example, probabilistic contracts (Delahaye, Caillaud, & Legay, 2011) or contracts for hybrid systems (Damm, Dierks, Oehlerking, & Pnueli, 2010).

To illustrate the underlying principles, however, we will skip over these specialized theories and instead focus on the standard literature (Benveniste et al., 2012; Baumgart et al., 2011; Hungar, 2011b) introducing the basic formalism and definitions together with the most commonly used theorems.

2.1.1 Trace Semantics

Contract semantics for reactive and embedded systems are defined over traces of a system (Hungar, 2011b; Baumgart et al., 2011). Components, in the following denoted with $\mathtt{M}$, are characterized by ports (P), that are either defined as input or as output ports. Note, that in contrast to the SPEEDS meta-model (SPEEDS, 2007), we do not further refine a port into multiple flows. A trace assigns a value V out of the value domain $\mathcal{V}$ to each of the ports at any given point in time $t \in \mathcal{T}$. Therefore, a trace is of the form:

$$[\mathcal{T} \to [P \to \mathcal{V}]]$$

The traces of a component $\mathtt{M}$ are denoted $[\![\mathtt{M}]\!]$. This set comprises all possible behaviors of a component (implementation), even in case of unacceptable inputs due to a general requirement of input openness, *i.e.,* requiring systems to never confine or otherwise refuse input. The (semantically concurrent) composition of multiple subcomponents $\mathtt{M}_1 \ldots \mathtt{M}_n$ to a component $\mathtt{M}$ then is defined (Baumgart et al., 2011) as the set of traces acceptable by all components:

$$[\![\mathtt{M}]\!] = \left[\!\!\left[\bigtimes_{i=1}^{n} \mathtt{M}_i \right]\!\!\right] = \bigcap_{i=1}^{n} [\![\mathtt{M}_i]\!]$$

For the sake of simplicity port renaming in assembly- and delegation-connectors are neglected, therefore identity between connected input and output ports is assumed.

[1]http://www.speeds.eu.com/

2.1.2 Contracts

A contract is a tuple $C = (A, G)$ describing a set of traces using the assumption A and the guarantee G:

$$[\![C]\!] = [\![A]\!]^{-1} \cup [\![G]\!]$$

with $(.)^{-1}$ denoting the complement of a set. Although not formally required by the definition of contracts, assumptions typically only specify constraints on the input, whereas guarantees reflect the input output relation and therefore contain restrictions of allowed outputs while being input-open. We assume all contracts to be stated in the *canonical form* (Benveniste et al., 2008) which allows some simplifications in the following definitions of the operators and relations. A contract is said so be in canonical form if $[\![G]\!]$ at least contains $[\![A]\!]^{-1}$. Note that this implies that the traces $[\![G]\!]$ associated with the guarantee G and the traces $[\![C]\!]$ of the whole contract $C = (A, G)$ do agree; yet the contract C syntactically is a pair (A, G) explicitly distinguishing the roles of assumptions and guarantees, which is instrumental to the pragmatics of contract-based design.

Some sources in the pertinent literature additionally distinguish the concept of a "Strong Assumption" and "Weak Assumption" (e.g., Damm, Hungar, Josko, Peikenkamp, and Stierand (2011) and Baumgart et al. (2011)). Strong assumptions are meant to express the overall "operational envelope" for a component, for example, the conditions that need to be fulfilled to perform any meaningful operation. The weak assumption allows the specification of different guarantees of a component for multiple different environmental situations, thus permitting case-based definition of desired behavior. Since the safety specifications used in this work consider only the latter, we will not further elaborate on strong assumptions.

2.1.3 Contract Relations and Operators

In order to reason about the correctness of the composition of a system, a set of basic relations and operators need to be defined on contracts and implementations.

Definition 1 (Satisfaction)**.** The satisfaction relation defines when the component complies to its contract. A component $\mathtt{M}$ satisfies a contract $C = (A, G)$, denoted $\mathtt{M} \models (A, G)$, iff all its traces are permitted by the contract, *i.e.,* $[\![\mathtt{M}]\!] \subseteq [\![C]\!]$, and its inputs and outputs coincide to those underlying the contract. Consequently,

$$\mathtt{M} \models (A, G) \Leftrightarrow [\![\mathtt{M}]\!] \cap [\![A]\!] \subseteq [\![G]\!]$$

This definition is non-controversial in literature (Baumgart et al., 2011; Delahaye et al., 2011; Benveniste et al., 2008).

Definition 2 (Refinement)**.** Refinement is a relation between two contracts, stating that the refined contract is a valid concretion of the other, *i.e.,* the refining contract is a valid replacement in all possible operational contexts satisfying all (and maybe more) requirements satisfied by the refined contract. According to Benveniste et al. (2012) and

Delahaye et al. (2011) a contract C_1 refines C_2 if it has the same signature, *i.e.,* same input and output ports, yet imposes relaxed assumptions and more precise guarantees:

$$C_1 \preceq C_2 \Leftrightarrow A_1 \supseteq A_2 \wedge G_1 \subseteq G_2$$

Baumgart et al. (2011) and Damm et al. (2011) call this relation dominance. Also Benveniste et al. (2008) use the term dominance, since they use the term refinement to denote a relation between different implementations.

Definition 3 (Parallel Composition)**.** For two contracts C_1 and C_2, the parallel composition $C_1 \otimes C_2$ is defined as:

$$((A_1 \cap A_2) \cup \neg(G_1 \cap G_2), (G_1 \cap G_2))$$

Parallel composition ($\otimes$) is used to combine multiple contracts into a new contract that represents the intended behavior of an ensemble of components individually satisfying these contracts. In contrast to conjunction (see below), the operator is used to combine contracts applying to different components at the same hierarchy level in the architecture. The operator creates a new contract with an assumption and a guarantee. This composition is typically used to combine all contracts of the subcomponents to perform a virtual integration test.

To be composable contracts need to be compatible, that is, their variable types match and an environment exists in which the two contracts interact properly in terms of appropriately connected ports (Benveniste et al., 2012).

This definition is equivalently stated by Delahaye et al. (2011) and Benveniste et al. (2008), who also claim to be in line with Henzingers definition used for Interface Automatons (De Alfaro & Henzinger, 2001). The parallel composition preserves canonicity (Delahaye et al., 2011). Hungar (Hungar, 2011a) defines the parallel composition as:

$$\left(\left(\bigcap_i A_i\right) \cup \bigcup_i \left(A_i \cap G_i^{-1}\right), \bigcap_i G_i\right)$$

This notation is equivalent to the above-mentioned one, considering that Hungar is not using a canonical form. Therefore his observation regarding the intersection of the complete trace-sets is also valid:

$$[\![\|_{i=1}^n (A_i, G_i)]\!] = \bigcap_{i=1}^n [\![(A_i, G_i)]\!]$$

Definition 4 (Conjunction)**.** Conjunction is an operator used to combine multiple contracts that are associated to one component:

$$C_1 \wedge C_2 = (A_1 \cup A_2, G_1 \cap G_2)$$

Conjunction may be necessary if for different environmental situations a separate behavior shall be described or different viewpoints are used (Benveniste et al., 2008) (there called greatest lower bound). The given definition is identical to Delahaye et al. (2011) and Benveniste et al. (2012). The result of a conjunction is still a contract in canonical form (Benveniste et al., 2008).

2.1.4 Theorems

Compositional verification refers to deducing the correctness of a global system by observing its atomic components only (Hungar, 2011b). This property is deduced by the following theorems which are equivalently stated in literature (Delahaye et al., 2011; Hungar, 2011a; Baumgart et al., 2011; Benveniste et al., 2008; Benveniste et al., 2012):

Theorem 1. *If a model $\mathtt{M}$ satisfies a contract C_1, it also satisfies all contracts that C_1 is refining:*

$$M \models C_1 \wedge C_1 \preceq C_2 \rightarrow M \models C_2$$

Theorem 2. *If the compliance of the models M_1 and M_2 have been shown to their respective contracts C_1 and C_2, the composition of the models satisfies the parallel composition of the contracts:*

$$M_1 \models C_1 \wedge M_2 \models C_2 \rightarrow (M_1 \times M_2) \models (C_1 \otimes C_2)$$

From the theorems it is directly following:

Corollary 1 (Virtual Integration)**.** *According to (Damm et al., 2011; Baumgart et al., 2011; Gezgin et al., 2014; Hungar, 2011a) virtual integration is given iff all subcomponents $\mathtt{M}_i$ of component $\mathtt{M}$ satisfy their respective contracts and the parallel composition of these contracts refine the top-level contract C then the top-level contract is satisfied by $\mathtt{M}$:*

$$\left[\bigwedge_{i=1}^{n} \mathtt{M}_i \models C_i \wedge \left(\bigotimes_{i=1}^{n} C_i\right) \preceq C\right] \rightarrow \mathtt{M} \models C$$

Hence, it is sufficient to prove the compliance of the parts to their specification to prove the complete design, if refinement of the contracts to the system specification is given.

For the sake of completeness it has to be stated that in case of the usage of strong assumptions additionally a condition has to hold to establish virtual integration:

$$\left(\bigotimes_{i=1}^{n} C_i\right) \wedge (A) \Rightarrow \bigwedge_{i=1}^{n} A_i$$

It needs to be ensured that strong assumptions are not violated, either because the strong assumption of the top-level component already requires that property, or the guarantees of the subcomponents establish them.

2.1.5 Contracts in Formulas

A contract can be represented as a formula if the selected logic supports negation or implication (Baumgart et al., 2011; Hungar, 2011a; Damm et al., 2011; Benveniste et al., 2012):

$$\text{form}((A, G)) = A \Rightarrow G$$

The parallel composition of contracts $(A, G) \otimes (B, H)$ can be expressed as one of the following (Baumgart et al., 2011):

$$((A \wedge \neg G) \vee (B \wedge \neg H), false) \tag{2.1}$$

$$((A \wedge B) \vee (A \wedge \neg G) \vee (B \wedge \neg H), G \wedge H) \tag{2.2}$$

$$(A \vee B, (\neg A \wedge H) \vee (\neg B \wedge G) \vee (G \wedge H)) \tag{2.3}$$

$$(true, \neg(A \wedge B) \vee (\neg A \wedge H) \vee (\neg B \wedge G) \vee (G \wedge H)) \tag{2.4}$$

The differences to the definition 3 can be compensated by using the canonical form. Therefore, $((A \wedge B) \vee \neg(G \wedge H), G \wedge H)$ is valid for contracts in canonical form.

Refinement can be expressed as:

$$C' \preceq C, \text{ if } A' \Leftarrow A \text{ and } G' \Rightarrow G$$

Virtual integration in formulas first expressed by Damm et al. (2011) and Hungar (2011a) as:

$$\left[\bigwedge_{i=1}^{n} (\mathtt{M}_i \models C_i)\right] \wedge \left[\left(\bigwedge_{i=1}^{n} C_i\right) \Rightarrow C\right] \Rightarrow \mathtt{M} \models C$$

This definition is not in line with the virtual integration definition given for trace semantics in this work. This is caused since Hungar and Damm specify refinement for non-canonical contracts and require strengthening of assumptions only for strong assumptions. They used the following refinement definition:

$$C' \preceq C, \text{ if } C' \Rightarrow C$$

consequently for trace sets:

$$C' \preceq C, \text{if } [\![C']\!] \subseteq [\![C]\!]$$

To adapt virtual integration to the definition of refinement for canonical contracts it needs to be slightly modified:

Given M with associated contract $C_g = (A_g, G_g)$ and subparts M_i each connected with a contract C_i (which is the conjunction of all contracts connected to M_i) the parallel composition of the C_i is given:

$$\bigotimes_{i=1}^{n} C_i = (A_p, G_p)$$

Virtual integration is defined as:

$$\left[\bigwedge_{i=1}^{n} (\mathtt{M}_i \models C_i)\right] \wedge [A_g \Rightarrow A_p \wedge G_g \Leftarrow G_p] \Rightarrow \mathtt{M} \models C$$

2.2 System Design using Aspects and Perspectives

One of the most important approaches to understanding, developing and maintaining complex system is the introduction of structure (Baumgart et al., 2011; Storey, 1996). For computing systems the usage of so-called, modules dates back to the mid 1960s (Baldwin & Clark, 2000), first introduced with the IBM system 360. The goal of this structure was to enable a development of the system by different specialist teams in an independent manner (Baldwin & Clark, 2000). It is still an important requirement for system structures to ease the collaboration among many development parties. While modules are considered to be a vertical structure (Benveniste et al., 2012; Damm, 2005) (i.e., a separation of functions at the same level of abstraction), Neumann (Neumann, 1986) introduced in 1986 a horizontal structure that he called *layered architectures.* Higher levels of software use the underlying layers to accomplish their tasks, and calls from lower-level software to higher-level software can be avoided. Layering is still a commonly used architectural pattern (Fowler, 2002) with popular specializations, such as the *model-view-controller* pattern (Krasner, Pope, et al., 1988) used *e.g.,* in the AUTOSAR software stack (AUTOSAR GbR, 2014). This structure allows the focus to be on the user-centric functionality first and the necessary services later on. Hence, this software structure is already going in the direction of a concept of multiple layers of abstraction in which the lower levels refine the entities on higher abstraction levels (Sage & Rouse, 2009; Sommerville, 2010). Nevertheless, there is a fundamental difference between the early concept of layered software and abstraction techniques. In a layered software architecture each component in a layer fulfills a special unique functionality, but uses other functions, that are described in lower levels. Hence, one component can only be found in exactly one layer of the software. If abstraction levels are used, the components on lower abstraction levels inherit the requirements of the higher layers, since the lower layers describe the identical entity already described in higher layers, but with more details.

With the frequently used model-based design principle, structure is not only a means to organize implementations, but also a central aspect of systems design. Structure is created first, functionality is added afterwards. Model-based languages and tools exist for many different engineering activities like controller design (SIMULINK), automotive software architectures (AUTOSAR) or GSN (goal structure notation) for representing safety cases (Kelly, 1999). But even the development processes themselves have become so complex that guidance by additional structures are welcome. One of the most frequently used process models is the V-model (Dröschel & Wiemers, 1999) or V-model XT (Friedrich, Hammerschall, Kuhrmann, & Sihling, 2009).

Nevertheless, all mentioned techniques and models are either technology, domain or abstraction level dependent. A system-wide structure to integrate the different existing architectural, behavioral and process models was missing. Approaches from the military domain, such as the Department of Defence Architectural Framework (DoDAF/DM2 2.02, 2010), have not gained much attention in other domains (Rajan & Wahl, 2013) because of their very domain specific nature. To develop a cross-domain usable system structure, the goals for structure need to be kept in mind, namely to provide support for distributed development and to better understand a system by a separation of concerns. Hence, a modern top level structure of the system is driven by the different development disciplines and experts groups involved in todays system's development. To identify the most important structural elements, intensive discussions with the projects partners from multiple engineering disciplines, such as aerospace, automotive and rail, has been held in the projects SPES2020 (Pohl et al., 2012) and CESAR (Rajan & Wahl, 2013), partly re-using outcomes from the project SPEEDS (Enzmann, Döhmen, Andersson, & Härdt, 2008). The result was a structure of the system that introduced to the abstraction levels two further concepts called *perspectives* (see Section 2.2.1) to refine structural levels of the system and *aspects* (see Section 2.2.2) to further classify behavioral descriptions. These concepts have also been taken up in the projects MBAT and SPES XT. Besides the already mentioned goals special attention was given to the traceability between development steps, requirements and development artifacts.

2.2.1 Structural Organization of the System

To separate the concerns of different stakeholders, the IEEE 1471 (2000) suggests using *views* and *viewpoints*. While a view is marking a concrete part of a system, a viewpoint is the convention of how to construct and use a view. To be able to cover the complete engineering space, from the initial description of the problem to concrete technical components, the framework developed in SPES and CESAR combines abstraction levels and structural viewpoints of the system into a two-dimensional space (see Figure 2.1). The structural viewpoints, called perspectives, describe the subsequent stages of a system separated by the types of structures used. The first two perspectives, the *operational perspective* and the *functional perspective* are problem oriented and, describe the intended use-cases of the element to be developed and derive, based on these scenarios, the items functions. The other perspectives (logical, technical, geometric) focus on the design of the solution.

The perspectives have been chosen to be able to map the V-process model to the resulting two-dimensional engineering space. The left "development side" of the V is going diagonally from the top left corner to the bottom right corner. This diagonality can be considered more as a "rule of thumb" than a necessary requirement, since operational descriptions typically do not cover very low abstraction levels (e.g., a circuit level). Furthermore, the technical level is frequently started at an intermediate abstraction level, where a refined logical architecture is mapped to a technical element, and not

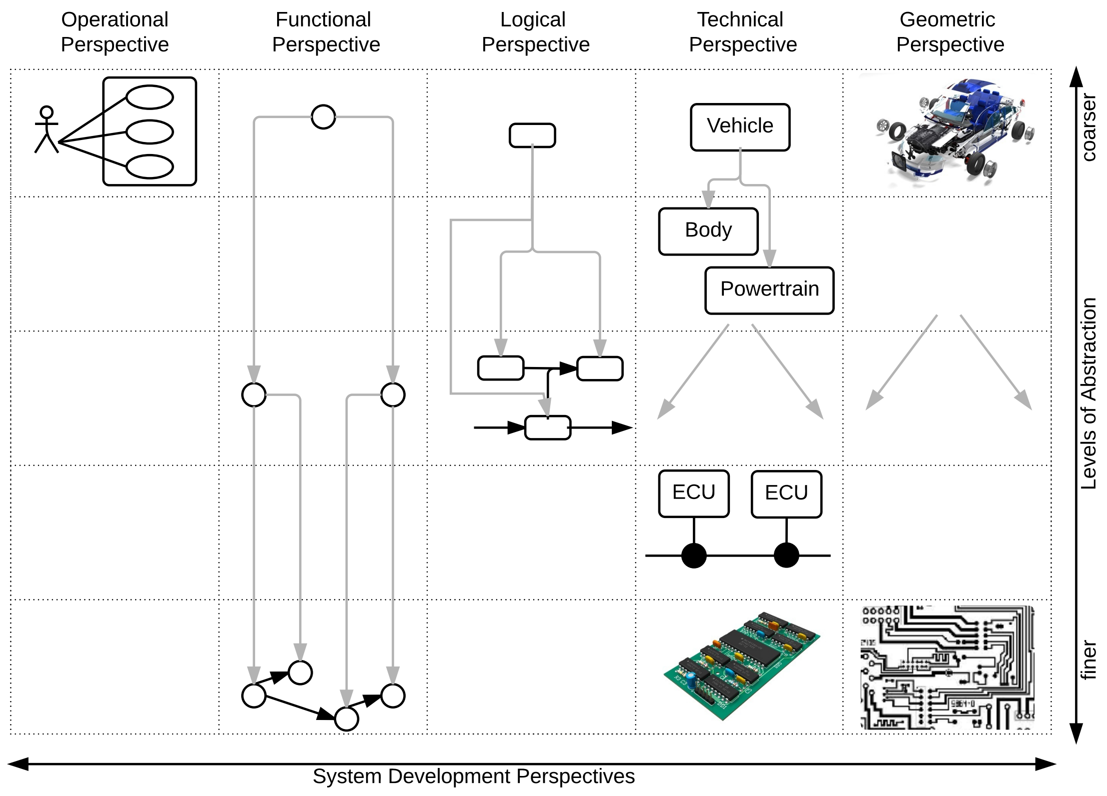


Figure 2.1 – Structural organization of the system in perspectives on the example of an aerospace system. Based on Baumgart et al. (2011) and Rajan and Wahl (2013)

reconstructed on the layers above. Nevertheless, the logical perspective is likely to be present on all higher abstraction levels up to a certain level, where the switch to the technical perspective occurs.

The relations between the different abstraction levels and perspectives are represented by two tracelinks (Baumgart et al., 2010; Baumgart et al., 2013) (see Figure 2.1), the *realize* and the *allocate* link. The realize links indicates that a component is refined by another component. The equivalent link between requirements is called *refine*. The realization can be expressed by a state-machine that creates the relation between the events in the refined and more abstract model (Baumgart et al., 2010). The allocation link indicates that a component or port is mapped to a more technical perspective. This mapping can also be expressed by a state-machine. This state-machine expresses the semantics of the mapping, since *e.g.,* a logical value in a port needs to be converted in a defined manner to a signal on hardware pins.

In the following the usage and the elements of the five perspectives are presented:

Operational Perspective We take the name operational perspective from the CESAR (Rajan & Wahl, 2013) project, while in SPES2020 (Pohl et al., 2012) this perspective is called the requirements viewpoint. Nevertheless, the intended use of the perspective is mostly identical. The item to be developed is considered as a black box and scenarios are described how the potential users shall interact with the system. Hence, it is also necessary to describe the environment in which the system shall operate. This perspective is used to analyze the customer needs and elicit the capabilities and activities of the system (Baumgart et al., 2013) as well as the goals of the stakeholders (Rajan & Wahl, 2013).

Additionally, in the SPES2020 requirements perspective, requirements for technical elements can be expressed, which is not the intention of the CESAR and MBAT models. Therefore, we consider the operational perspective in the meaning of CESAR and MBAT, since technical requirements in the operational perspective would break the idea of keeping the first two perspectives problem oriented, without performing a design of the solution. This approach fits better the requirements of safety standards like the ISO 26262 (2011) (see also section 2.3) in which the initial description of the system, which is used for the hazard analysis, should be performed without any technical assumptions on the system.

Functional Perspective In the functional perspective a hierarchical structure of functions is built. The top-level functions are called user functions (Pohl et al., 2012), and these functions can be defined using input from the user and an expected output from the system. The user functions can be refined in *functions* in a so-called whitebox model of the functional perspective. Hence, the functions represent the summarized capabilities and scenarios from the operational perspective (Rajan & Wahl, 2013). The here defined functions shall be independent of any architectural assumptions, if possible (Baumgart et al., 2011). Furthermore, constraints and restrictions to the functions can be stated (Baumgart et al., 2013). This could be mapping constraints, that two functions shall be mapped to an identical technical target, or be at least "close" together, because of performance reasons. Vice versa, independence constraints, requiring a different allocation target, could result from safety considerations (Ellen, Etzien, & Oertel, 2012).

Logical Perspective In contrast to the operational and functional perspective, the logical perspective is not problem but solution oriented. Hence, the actual design of the item is described. Therefore, components and their part relations describe the hierarchy of system elements. This representation does not yet distinguish between software or hardware components and considers them in an equal, logical, manner (Rajan & Wahl, 2013). Furthermore the interface of the top level component, characterizing the system's boundary, details the interaction with the environment (Baumgart et al., 2011). The functions modeled in the functional perspective are allocated to the components of the system. This allocation should be performed only within the same abstraction levels and should respect the constraints given in the functional perspective. The separation of the system into multiple logical parts is a means to start early with a distributed development process. In addition, costs can be saved by grouping functions, which are reused by multiple user functions (Pohl et al., 2012).

Technical Perspective The technical perspective describes the physical architecture of the system (Pohl et al., 2012). Hence, components are either hardware- or software-components. It is the goal to describe the different ECUs of the target system with their peripherals and the communication infrastructure. Also the allocation of software tasks to hardware elements and their intended interface is specified. The technical perspective is much more detailed than the logical to support more specific analyses. To be able to perform, for example, a timing analysis, the scheduler, resources and task parameters also need to be specified (Pohl et al., 2012; Baumgart et al., 2011). Considering that the lowest abstraction level of the technical viewpoint is intended to act as the implementation specification of the atomic components, the degree of details does not come as a surprise (Rajan & Wahl, 2013). Hence, many domain specific meta-models exist to describe parts of the technical perspective, such as AUTOSAR (AUTOSAR GbR, 2014) in the automotive domain, or IMA (integrated modular avionics) (Prisaznuk, 1992) in the avionics domain.

Although the projects deal with E/E systems to a great extent, the hardware parts are not limited to electronic components but can also contain mechanical systems like cams, shafts and switches or hydraulic elements like valves and cylinders (Baumgart et al., 2011).

Geometrical Perspective The geometrical perspective is not mentioned in SPES2020 but exists in the CESAR framework, the Architecture Modelling technical report (Baumgart et al., 2011) and publications of the MBAT Project (Baumgart et al., 2013). In CESAR this perspective is marked as experimental, and, although current research still tries to elaborate on this perspective (Baumgart, 2013), we only briefly introduce this perspective, for the sake of completeness.

It is the intention of the perspective to cover the physical installation properties of the system and their spatial dimensions. This covers the placement of components, the routing of cables and size limitations (Baumgart et al., 2011). The elements inside the geometric perspective can be based on CAD models *e.g.,* from CATIA or AUTOCAD.

Since the geometric properties are closely connected to the technical realizations of the components, it is expected that the technical and geometrical perspective will be jointly developed.

2.2.2 Classification of Dynamic Behavior

Orthogonal to the perspective describing the structure based on their “technicality” a structure called *aspects* is introduced to describe multiple behavioral concerns of the system. These aspects are oriented on the various analysis techniques, which are typically performed by different expert teams. In the SPEEDS project, the aspects safety, real-time and functional behavior have been defined (Böde, Gebhardt, & Peikenkamp, 2010). This selection is identical to the architecture modeling technical report (Baumgart et al., 2011).

In addition, in the CESAR book (Rajan & Wahl, 2013) the aspects performance, interface and product line are mentioned. This already indicates that the aspects, in contrast to the perspectives, are not predefined and vary from product to product. Research is currently being performed in the direction of further aspects such as power (Nitsche et al., 2013) or EMC (Baumgart et al., 2014).

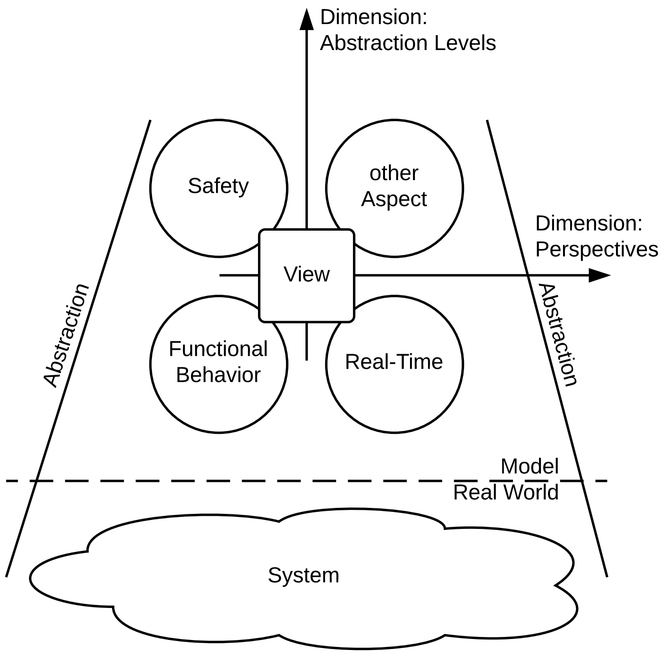


Figure 2.2 – Classification of dynamic behavior of the system using aspects.

The aspects comprise a structure that is set on top of the two-dimensional engineering space given by the abstraction levels and perspectives (see Figure 2.2). Hence, they use the same structural component model but describe different properties providing a structure to the requirements. Not all aspects are relevant in all perspectives (Baumgart et al., 2010) (e.g., a real-time aspect would not be used in the geometrical perspective) nor it is unlikely that the aspect for electromagnetic compatibility would be considered in the operational perspective, unless the device is expected to control radiation by itself.

A good intuition of how the requirements within the different aspects may look is given by the requirements specification language (RSL) (Reinkemeier et al., 2011; Mitschke et al., 2010; Baumgart et al., 2011) developed in the CESAR Project. This RSL provides requirement templates, called patterns, for selected aspects.

Functional Behavior Requirements assigned to the aspect functional behavior define the order and values of events or conditions on ports of the architectural components. It is also possible to define intervals in which events are expected. These intervals might be expressed by conditions, or by time. If timing information is used in these requirements they are likely to be present in both aspects, functional behavior and timing.

Real-Time Requirements assigned to the real-time aspect describe the timing behavior of events or conditions. Patterns exist to describe properties like the period of periodic events, the minimum inter-arrival time of sporadic events or the jitter of task activations. Also, properties like the maximum distance between two events can be specified.

Safety Requirements assigned to the safety aspect describe the relation between faults, failures and functions. Also hazards and the criticality of failures can be specified. A major part of this thesis (see chapter 4) is detailing the safety view by providing extended specification concepts and analysis methods.

2.3 ISO 26262

The ISO 26262 (2011) is the standard for the development of E/E (electrical/electronic) automotive systems for vehicles below 3.5t of weight. With this standard the automotive industry aimed at a summary of the current state-of-the-art techniques for developing safety critical automotive systems to give a legal border in case of claim for damages. Therefore, there is no need for certification, as it is the current practice in the avionics domain, but it is the decision of the individual companies if they want to perform a qualification of their item, which needs to be performed by a sufficiently independent assessor. The ISO 26262 standard consists of ten parts; the first nine are normative and the last one is informative, containing explanatory texts, examples and guidelines. Each part consists of multiple clauses representing process steps that contain the requirements and have defined input and output workproducts.

Parts 3–7 describe the main elements of the underlying V-Process-Model (Dröschel & Wiemers, 1999; Friedrich et al., 2009) (see Figure 2.3) proposing a top-down approach. Parts 1, 2, 8 and 9 are applicable in parallel to the other parts, covering general techniques applicable in multiple phases of the V-model. Part 3 is the first item-related part in the V-model of the ISO 26262 and subsumes the requirements concerned with the description of the item, how risks are identified and classified, as well as how it is planned to react to the risks and build a safe item. The clauses of this part are described in more detail in the following sections. Part 4, covering the system level, is separated into two phases. Clauses 4-5 to 4-7 shall be performed before the development of hard- and software components and result in a description of the system, while clauses 4-8 to 4-11 are performed after the item has been built and include requirements for the final safety assessment and system level tests. Parts 5 and 6 can be executed in parallel and includes the requirements for the development of hardware and software elements.

It is the main idea of this standard to start the development process with an analysis of the risks of the item and then derive and refine requirements that state how to avoid these risks. Hence, the safety case (the argument that an item is sufficiently safe) is focused on a correct refinement of the safety requirements and the correct implementation of the most detailed requirements in hardware and software. Therefore, it is essential to establish and

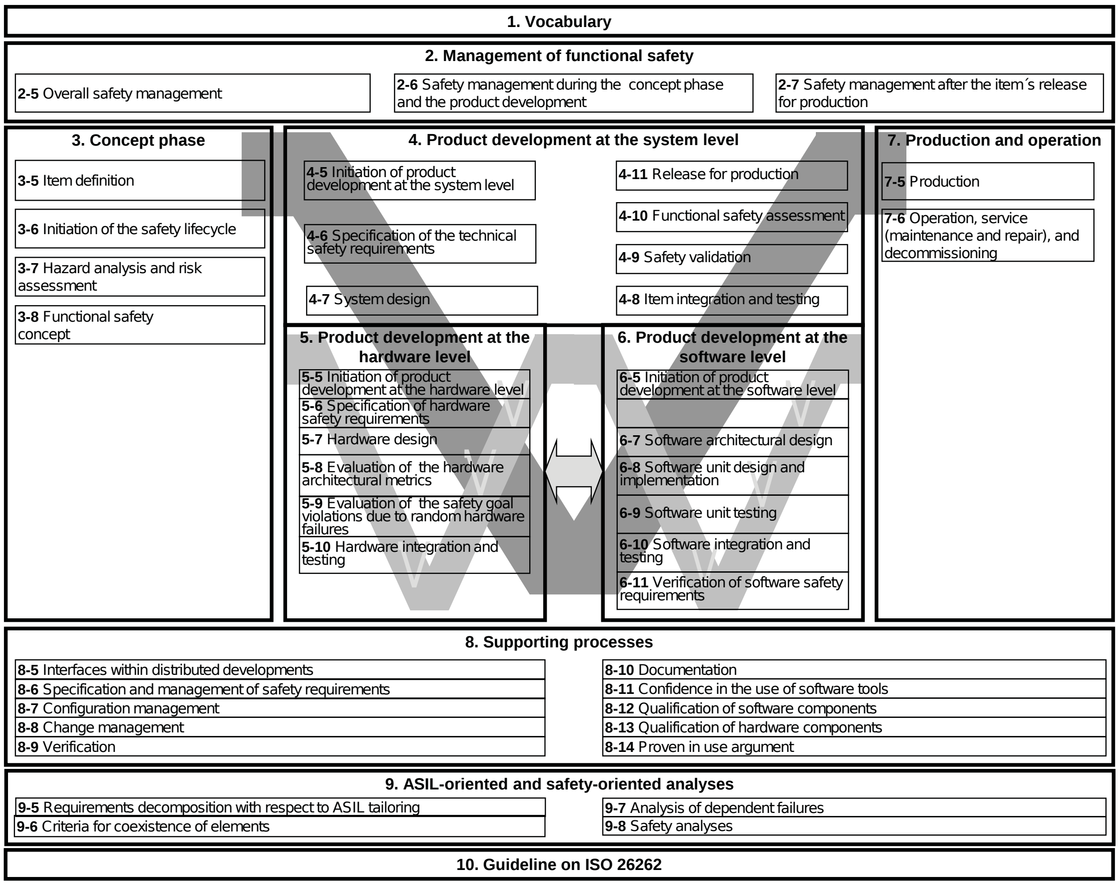

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-8 Item integration and testing
4-9 Safety validation
4-10 Functional safety assessment
4-11 Release for production
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration and testing
6-11 Verification of software safety requirements
7. Production and operation
7-5 Production
7-6 Operation, service (maintenance and repair), and decommissioning
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Figure 2.3 – The main development part of the ISO 26262 V-process model. Source: ISO 26262 (2011)

maintain the traceability between all safety requirements, the safety goals and the design artifacts. To ease the traceability between artifacts and create a consistent system view, model-based approaches are considered the state of the art development approach in the automotive industry (Armengaud et al., 2012; Born, Favaro, & Kath, 2010; Biehl, DeJiu, & Törngren, 2010; Armengaud, Bourrouilh, Griessnig, Martin, & Reichenpfader, 2012) as well in the avionics industry (Peikenkamp et al., 2006). Even though tools exist (ikv++ technologies ag, 2010; Büchner, Glöe, & Mainka, 2003; MathWorks, 2011) to ease the handling of traceability links and guide the engineer though the development process, most of the validation and verification tasks are solved using reviews or manual analyses.

The literature extensively covers all parts of this safety standard either for engineers (Ross, 2014) or for academics (Gebhardt, Rieger, Mottok, & Gießelbach, 2013). Instead, we will focus on part 3 of the ISO 26262.

2.3.1 Item Definition

The *Item Definition* is a clause and a work-product containing the description of the item in a high-level way. The basic functions are described along with the assumptions on the environment. The environment description comprises the functional interface of the component like buses with their messages and also information regarding the nonfunctional environment like maximum operating temperature, humidity or physical shock. Furthermore, references to similar, already built, systems shall be given together with all already known requirements from the customer. The item definition is the base for all the succeeding phases of the ISO 26262.

2.3.2 Hazard Analysis and Risk Assessment

A principle of the risk-based approach is to determine early how malfunctions in the system could harm humans. These potential sources of harm are called *hazards*. This analysis is performed in the "Hazard and Risk Analysis" (HARA) based on the item definition. This analysis is therefore intentionally performed without knowledge of the implementation of the system, but purely on the intended functionality. This is an important factor in the argument about the safety of the system, since on the functional level a complete analysis can be performed, which is much more difficult on the implementation level.

Hence, for each function of the item, the potential malfunctions are described and classified. The classification is based on three parameters: *Severity*, *Exposure* and *Controllability*. The severity classifies the potential damage of a malfunction in terms of the harm to persons affected by the item ranging from no injuries to possible death. The exposure classifies the likeliness of the situation in which the hazard could occur. This is different from the likeliness of the hazard to occur. For example, if the hazard can occur only during highway driving, the probability of driving on a highway is considered to be greater than 10% of the average operating time. For multiple driving situations example values are given in the standard. The last parameter of the classification of a hazard is given by the controllability of the hazard. This value reflects how well the driver or other affected persons can react to the hazard.

Resulting from the individual values for severity, exposure and controllability an ASIL (automtotive safety integrity level) is calculated. This ASIL indicates how "critical" a hazard is. The ASIL ranges from A to D, where A is the lowest criticality and D is the most critical one. For hazards with a higher criticality more and also stricter requirements apply than for lower classified hazards. There is a fifth classification, QM, which is even lower than ASIL A and indicates, that the normal quality management is sufficient for this kind of hazard. Figure 2.4 depicts the relation between the classifiers. The more severe the consequences of a hazard may be, the more effort needs to be invested in the development process of the item to ensure that the hazard does not occur. This effort can be reduced if the situation occurs seldom or a good controllability is given. The resulting effort is represented by the ASIL.

If all hazards are classified, they are formulated as a requirement, typically that the hazard shall not occur, with the identified ASIL as a parameter of this high level safety

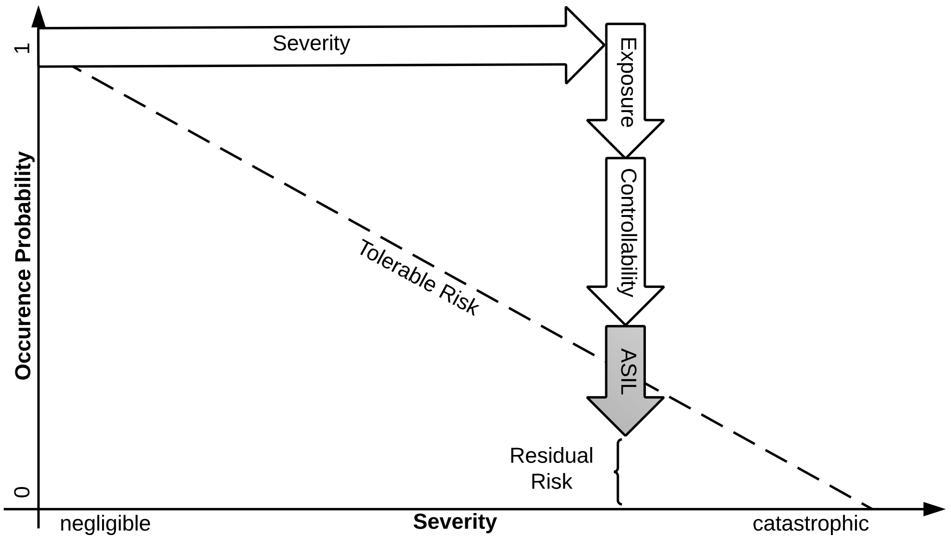


Figure 2.4 – Relation between the three classification factors resulting in the ASIL of a hazardous event.

requirement. These requirements are called safety goals and are the resulting workproduct of the hazard analysis and risk assessment phase.

2.3.3 Functional Safety Concept

The ISO 26262 requires that proof be given how the safety goals shall be fulfilled. The functional safety concept (FSC) is a functional, high level description of how to achieve this. The FSC consists of a first draft of the architecture of the system, called *preliminary architecture*, and requirements that refine the safety goals, the *functional safety requirements*. The functional safety requirements shall cover (ISO 26262, 2011):

- **Fault detection and failure mitigation requirements:** This implies that faults and failures are identified in the requirements and the architectural description. The malfunctions described in the safety goals can be used as the failures of the top level component. The faults are still on a functional level (e.g., that a calculation is performed incorrectly) rather than, for example, a broken hardware device or software bug. Furthermore, it needs to be described which component is able to detect faults and how to react to them. Hence, failure mitigation is here understood as graceful degradation.

- **Transition to a safe state:** One possible failure mitigation strategy is to switch to a safe state after detecting a fault. This action is considered as a degrading fault tolerance mechanism, that is, the functionality of the system is reduced while still being safe. The degree of the inability to perform its intended operation may vary (e.g., from only limiting the maximum performance to a complete shut down of the system). Since each hazardous event has its own safe state(s), each has to be noted and the components that perform a potential switch to a safe state need to be clearly identifiable.

- **Fault tolerance mechanisms:** All fault tolerance mechanisms (i.e., components that stop a fault from propagating in its original from) need to be identifiable in the architecture of the functional safety concept. According to Bozzano and Villafiorita (2003) and Storey (1996), a fault tolerance mechanism has a hypothesis stating a set of assumptions when the fault tolerance mechanism is working correctly and when it fails. Such assumptions are, for example, the freedom from faults of the voting component or the independence of the involved components.
- **Fault detection and driver warning:** If a fault cannot be mitigated and the fault tolerance time interval allows a late removal, it is a valid technique to alert the driver about the fault and request a repair.
- **Arbitration logic:** Between multiple events, that are generated by components simultaneously and shall describe the same property, a decision needs to be made as to which values shall be processed and which discarded. Hence, arbitration is often used in voting components to select which value of the multiple channels shall be passed on.

The FSC needs to be verified with respect to two goals: The consistency and compliance with the safety goals and the ability to mitigate or avoid the hazardous events.

2.3.4 Technical Safety Concept

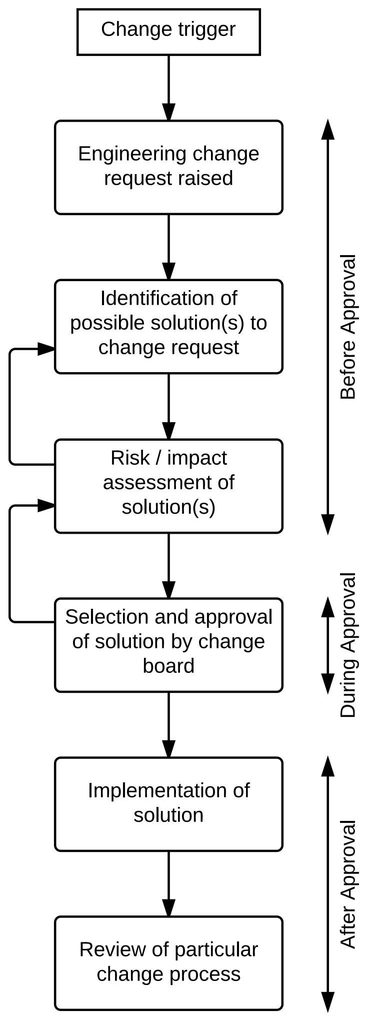
The distinction between the functional and the technical safety concept not only is an essential aspect of the ISO26262 but also is reflected in many development methodologies. See, for example, the distinction between Virtual Function Bus view (AUTOSAR GbR, 2010) and Basic Software View in AUTOSAR, or in the integrated modular avionics approach (Prisaznuk, 1992) from the avionic domain. The basic idea behind this approach is to provide an early statement about how to establish a safe system without considering the actual implementation. This opens up the possibility of detecting design errors impacting safety at a early stage of the development process and avoiding having to "add" safety to the product after the technical specification of the system functions is completed or parts of the system have already been built. This modus operandi directly implies strict consistency and completeness rules that have to be applied to the mapping of functional to technical elements.

CHAPTER 3

Development of a Semantic-based Impact Analysis

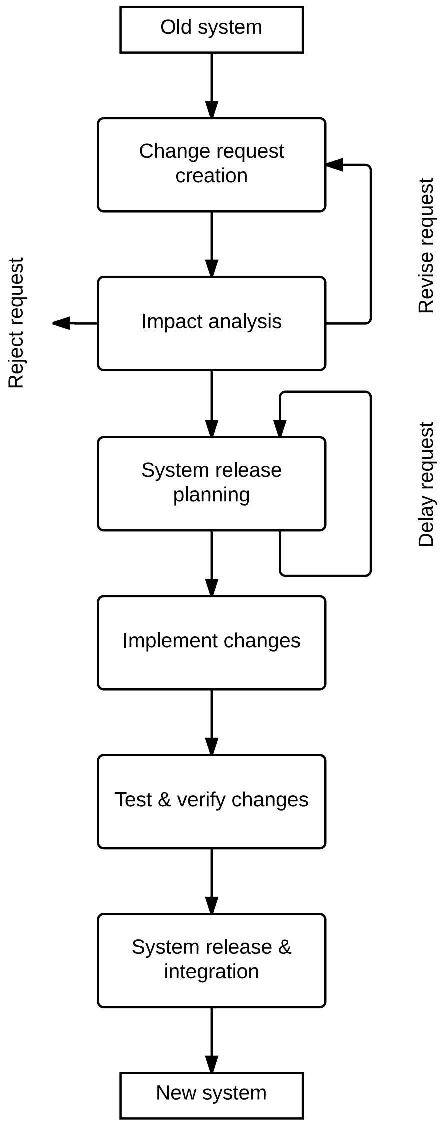
Changes are an inevitable part of today's system engineering practice. Changes are discussed in a very controversial way, from being the main driver of innovation (Eckert et al., 2004) to being considered as "the real killers" (Standish Group, 1995). Nevertheless, it is common sense that changes are a main factor in a system's overall costs. Fricke et al. (2000) argues that, according to his case studies, about 30% of the work effort is due to changes. This is not surprising, since changes can result from many different sources (Eckert et al., 2004; Eckert, Weck, Keller, & Clarkson, 2009): changes on requirements introduced by the customer, changes in standards relevant for certification, adaptation to newly emerging technologies or even detected problems in the design. Additionally, new systems are rarely built from scratch, but rely on modifications of already existing ones. Hence, the management of changes has long been considered as a requirements and systems engineering subdiscipline (Sage & Rouse, 2009; Sommerville & Sawyer, 1997; Robertson & Robertson, 1999). Since an imprudent acceptance of a change can easily cause a whole project to fail, "all possible implications should be considered before accepting or rejecting the proposal" (Kidd & Thompson, 2000). In the currently available literature, change management is mostly (Kidd & Thompson, 2000; Jarratt et al., 2011; CMMI Product Team, 2010; ISO 10007, 2003; Lock & Kotonya, 1999) approached from a process perspective. Figure 3.1 depicts two representative change management processes that are very similar. After having received a change request, the impact of the change is analyzed, then the changes are implemented, and then an assessment of the implementation is performed. In none of these process models is the impact analysis coupled with the implementation task.

This coupling becomes more important if the challenges of impact analyses are considered. Arnold and Bohner (1993) defined an impact analysis as "the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a



Change trigger
Engineering change request raised
Identification of possible solution(s) to change request
Risk / impact assessment of solution(s)
Selection and approval of solution by change board
Implementation of solution
Review of particular change process
Before Approval
During Approval
After Approval

(a) Change Management Process according to Jarratt (2004)



(b) Change Management Process according to Lock and Kotonya (1999)

Figure 3.1 – Impact analysis as a discipline within change management

change.” Hence, he already mentions the two targets of impact analysis: being able to estimate cost and time, and correctly implementing a change.

The correctness of implementing a change is especially challenging, since nearly all changes require further modification in the system to be implementable. A single change to a requirement or an implementation will most likely create an inconsistency in the system, for example, a conflict with other requirements, an exceedance of resource usage or simply the violation of cost restrictions. These further adaptations in the system are called change propagation (Eckert et al., 2004; Dick, 2005) or ripple effects (Bohner, 2002; Bohner, 1996) or also a change snowball-effect (Terwiesch & Loch, 1999). These effects have been investigated by researchers across the globe on various targets, like software or complete systems, including requirements or model artifacts. A summary of existing impact analysis techniques dealing with change propagation are presented in Section 3.1.

In the context of safety critical embedded system, impact analyses become even more important. These systems are subject to a strict quality assurance process guided by domain specific standards like ISO 26262 (2011) (automotive) or ARP 4761 (1996) (aerospace). These standards aim to reduce the risk of harming people by a systematic hazard analysis and structured breakdown of safety requirements and system design as well as extensive analysis and testing of the system. Hence, the verification activities build a major fraction of the total development costs of a system. The literature approximates this fraction at about 50% (Terwiesch & Loch, 1999) or even 75% of the costs for safety critical systems (Laprie, 1994). Huge effort can be saved, if a determination of the system parts that need to be re-verified is possible. While this can be achieved for some limited changes in software (refactoring) or upgrades of aircrafts that are especially designed to be modified during their lifetime, none of the currently existing impact analysis techniques is applicable for automotive safety concepts (see gap analysis in section 3.2).

In section 3.3 we present a novel impact analysis technique that is providing accurate results, that enables a re-certification process and is based on a subset of verification and validation activities. Hence, our impact analysis technique ensures to identify all verification and validation activities that are affected by the change and not more. To achieve this, the approach is based on the semantics of system requirements rather than relying on interconnections of components only. Based on this impact analysis process we state a set of requirements on the concrete language, which is used to express the safety concept of the system, in section 3.4. We conclude the results gained in section 3.5.

3.1 Related Work on Impact Analysis Techniques

Extensive comparison of impact analysis approaches have been performed in the last few years.

Jarratt et al. (2011) summarize existing approaches to deal with engineering change in their literature survey. They define an engineering change as an “[...] alteration made to parts, drawings or software that have already been released during the product design process. The change can be of any size or type; the change can involve any number of people and take any length of time.” They categorize the existing literature in three categories of approaches. Techniques in the first category focus on the process, such as research related to change propagation. The second category is populated with approaches that focus on tools to support changes in products. Finally, product oriented approaches handle engineering change by developing systems that are easier to change. The last category is not in the scope of this thesis.

In the field of software impact analysis a survey has been written in 2011 by Lehnert and published as a technical report (Lehnert, 2011) summarizing more than 150 different approaches. He classified the approaches with respect to the number of supported system artifact types, like code, architectural components, documentation or requirements. He discovered that only 13% of the available impact analysis techniques are concerned with multiple artifact types (e.g., dealing with implementations and architectures at the same time). Of these 13% only 17% have been evaluated. Hence, very little information about the performance of impact analyses for complete systems is available.

Kilpinen (2008) analyzed in 2008 impact analysis techniques using the classification of Bohner (1996), namely traceability based impact analysis and dependency based analysis. While the first approaches are based on manually created tracelinks between system artifacts, the approaches in the second category use more detailed information within the system, like source code, to extract relations. Such relations can be established by, for example, usage of variables or functions. Kilpinen extends these categories with the class of experience-based impact analysis approaches such as reviews, walk-throughs or inspections. These can be applied even if no traceability information is available, but depend on skilled engineers.

Similar to Kilpinen and Bohner, Dick (2005) distinguished between explicit traceability (e.g., a satisfaction relation between requirements—in this thesis called refinement), which is especially created for documenting the relation between artifacts, and implicit traceability, given if there is another primary reason for establishing a relationship, such as the assignment of code artifacts to a task.

Nearly all of the existing impact analysis techniques focus on improving the detection of influences of changes on other system elements. But there remain two basic problems: missing a potential impact and highlighting too many possible candidates. Both problems are equally important and can result in significant development costs due to unnecessary verification activities or reduced quality of the system caused by missed ripple effects. Also, both factors influence each other. If the set of possible candidates is too large, the accuracy of detecting the real propagation candidates decreases and therefore the probability of missing a necessary change increases. For safety critical systems it is therefore important to prune candidates as much as possible while still avoiding missed propagations. Hence, in this thesis we use a classification of impact analysis approaches that is different from the already existing ones, oriented on the accuracy of the approaches. This classification is influenced to a great extent by the degree of available semantic knowledge used in the relations between system artifacts.

Hence, we distinguish between approaches not being dependent on any tracebility information (Section 3.1.1), approaches that use manually created traceability information (Section 3.1.2), approaches that extract traceability information out of behavioral models (Section 3.1.3), approaches that use probabilitic techniques to extract traceability information out of previous changes (Section 3.1.4) and finally approaches that ensure the identification of all candidates (Section3.1.5). Based on this classification we evaluate academic approaches as well as commercial tools.

3.1.1 Impact Analysis without Tracing

Kilpinen (2008) identified a field of impact analysis that she calls “Experiental Impact Analysis.” Approaches in this category do not use any form a traceability but rely completely on the skill of the involved engineers. She argues that reviews or discussions in the development team can reveal relations between artifacts that are not covered by traceability. Ambler (2002) goes a step further and warns readers of his book on agile developments techniques that the costs of traceability do not pay off. He argues that in his experience companies do invest too much time in updating and maintaining the traceability matrix. Hence, it is easier to ask one or two of the skilled engineers, who know the system well. Similarly, Graaf, Lormans, and Toetenel (2003) mention that the traceability is often not maintained because of insufficient tool support and untrained personnel.

This approach is currently being practiced especially in small companies. The numbers mentioned by Ambler—one or two very skilled developers who know the system well per project—indicate that Ambler is not addressing big projects, in which it is unlikely that a single person is able to estimate the changes in all existing components.

Nuseibeh, Easterbrook, and Russo (2000) describe the problem of maintaining consistency among different design artifacts in the development process, like specifications, test plans, source code or change requests. While doing so, they provide a justification for Ambler’s statement. The relations between the mentioned development artifacts are the typical targets for traceability management. They define inconsistency in this context to “denote any situation in which a set of descriptions does not obey some relationship that should hold between them.” As an example they state a consistency rule: “In a data flow diagram, if a process is decomposed in a separate diagram, the input flows to the parent process must be the same as the input flows to the child dataflow diagram.” This is a possible consistency rule for a decomposition traceability link in data flow diagrams. They argue that for large systems it is infeasible to maintain the consistency of this traceability information since different developers update or construct elements or relations. They reported from a use case in which it was not possible to perform formal analyses on a software because the system changed to such a great extent during the preparation of the analysis, that the results would have been worthless by the time they would be available. While local consistency criteria might be possible to establish, they do not guarantee global consistency. Since inconsistencies are unavoidable they suggest using it as a tool to identify areas of the product that require more attention during the design. They propose an iterative change process for handling traceability inconsistencies as depicted in Figure 3.2.

They do not intend to fix all inconsistencies, but intentionally foresee whether to ignore or tolerate them. They consider inconsistencies as a risk and try to determine how

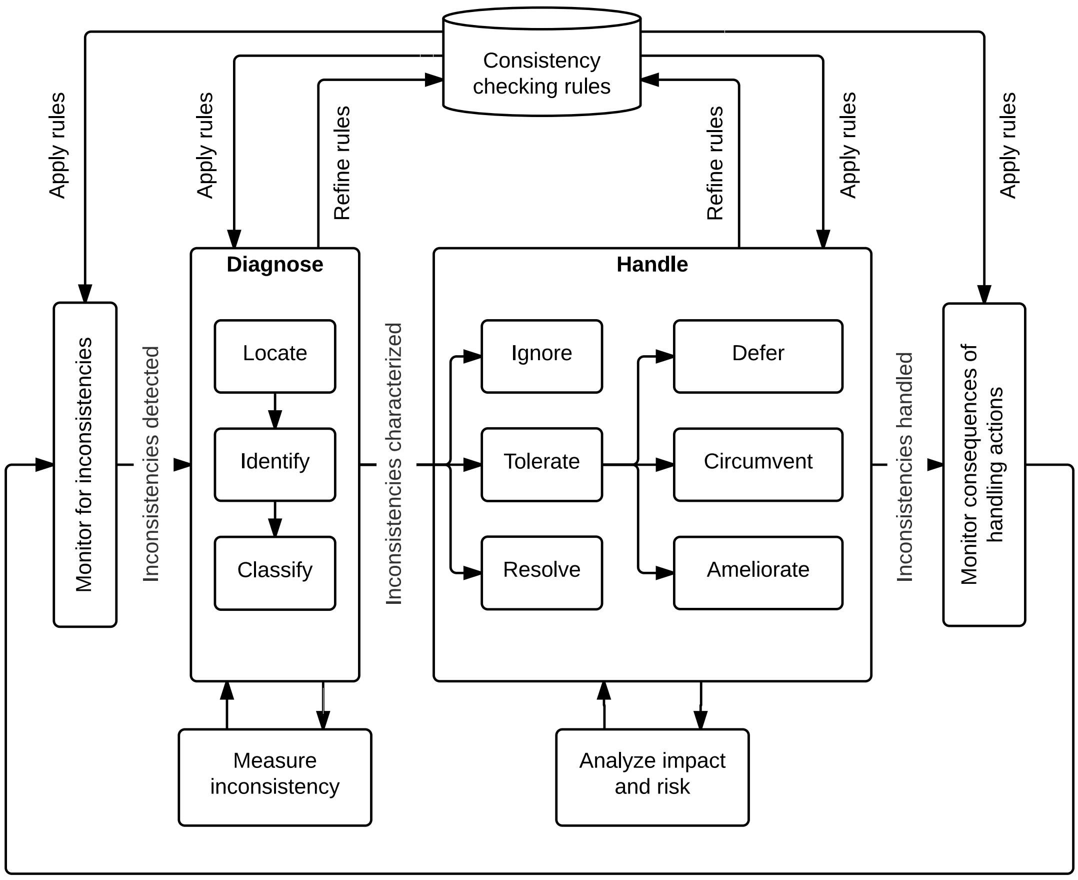


Figure 3.2 – Process how to handle inconsistencies according to (Nuseibeh, Easterbrook, & Russo, 2000)

expensive a problem this inconsistency could become. If the costs of such inconsistencies are less than the effort of fixing them, the inconsistency is not resolved. With the effort to maintain a consistent traceability and the therefore high associated costs, it is understandable that other, potentially cheaper, solutions are targeted.

Also Humphrey (2000) highlights the importance of reviews to identify faults. In contrast to Ambler, his argumentation is based on the quality of reviews compared with tests. He has counted the keystroke errors he made while entering data or typing code. He identified 28 keystroke errors on 1000 lines of code, from which 9.4% have not been flagged by the compiler, leaving 2 to 3 random errors per 1000 lines of code. He argues that the test coverage needs to be extremely high to identify all of these errors, since they affect the result in only a few situations. Hence, following traceability and re-running the test associated with the components might not discover these faults; reviews are a much better approach, in his opinion.

Still, especially for safety critical systems it is required by most of the safety standards to perform tracing between system artifacts and keeping this traceability consistent.

	R1	**R2**	**R3**	**R4**
R1			X	X
R2		X		
R3				X
R4	X			

Table 3.1 – Traceability matrix to represent the dependencies between requirements. Source: Sommerville and Sawyer (1997)

Hence, the large effort for maintaining traceability is not a valid argument for these approaches in this case.

3.1.2 Explicit Traceability Impact

While the opponents of traceability form a minority across the group of practitioners and researchers in the field of system design, it is a common recommendation to establish and maintain traceability between system artifacts (Gotel & Finkelstein, 1994; Robertson & Robertson, 1999; Ramesh, Powers, Stubbs, & Edwards, 1995). Also standards like CMMI-DEV (CMMI Product Team, 2010) highlight the importance of traceability as a discipline within requirements management. More specialized standards like the ISO 10007 (2003), which focuses on configuration management, states that traceability is one of the key requirements for an effective change control.

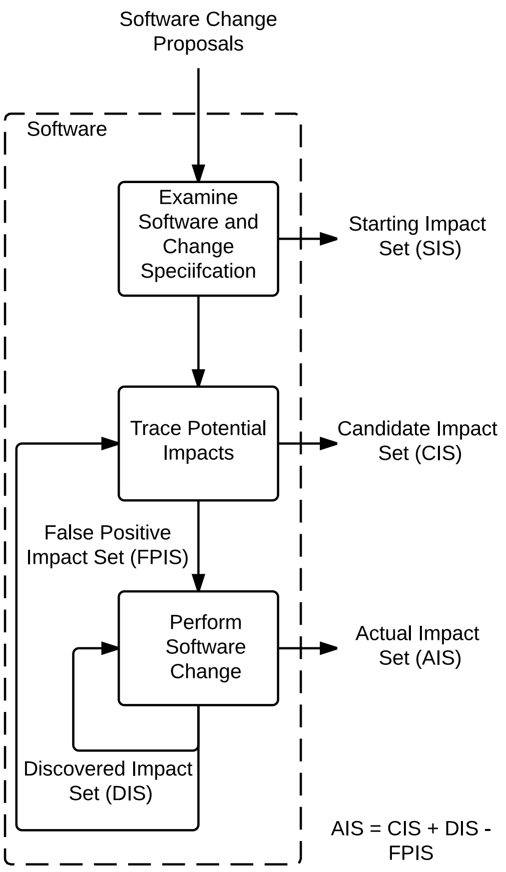
Sommerville and Sawyer (1997) distinguish between six types of traceability links. These links connect requirements on the one side to source code, rationals, other requirements, architecture components, design components and interfaces on the other side. While sources, rationals and other requirements are self-explanatory, They classify architectural components on logical level as architecture, while calling components that are already separated into hardware and software, as design components. Interface requirements are requirements on external elements to the system and considered as assumptions in this thesis. In literature, the traceability between verification activities and requirements or implementations, as well as documentation (Rajan & Wahl, 2013) is frequently missing. Sommerville highlights the simplest representation of a relationship between requirements in a traceability matrix (see Table 3.1).

Traceability matrices, representing directed dependencies, can be easily extended towards multiple dependency types such as refinement or satisfaction. This kind of traceability management is typically supported by tools such as IBM Rational DOORS (Software, 2015) or Dassaults Reqtify (Dassault Systems, 2012), which additionally provide a more integrated view on traceability in the system.

According to Leveson and Weiss (2004) traceability is considered as a fundamental technique to build safe systems. They do not restrict traceability to relations between requirements and software modules, but suggest extending the term to design feature and decisions. Especially if components are reused in another system this extended traceability allows the use of stated assumptions to decide if a component fits in a different context. In a contract-based design approach every requirement features an assumption for exactly that purpose (see Section 2.1).

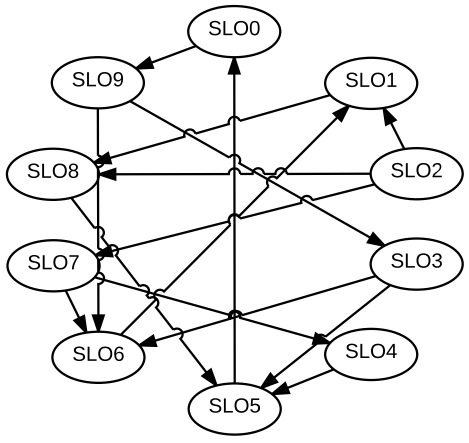
Also Dick (2005) and Robertson and Robertson (1999) highlight the importance of traceability between design artifacts and design layers for change management of complex systems such as aircrafts or weapon systems. Dick states two main benefits of traceability: first, the understanding of the system is improved. Questions like "What is the role of this component?" can easily answered. Second, and more important for this thesis, traceability enables a semi-automatic impact analysis. Assuming that a proper tool support is given, the relationships between requirements, software and hardware artifacts, as well as documentation elements help to identify potentially affected entities. In contrast to Ambler (2002), Dick is confident that the investment in traceability pays off. Dick suggests a process for using traceability to conduct changes in a system. After creating a traceability tree with all the linked elements connected to the changed one, pruning of branches has to be performed by engineers since the existence of a link does not necessary mean that a change propagates. Finally, changes are planned and applied in all affected system layers.

A similar process is presented by Bohner (1996) introducing the term *Software Change Impact Analysis* which he defines as "The determination of potential effects to a subject system resulting from a proposed software change." The process is depicted in Figure 3.3. The *Starting Impact Set* (SIS) includes the initially changed elements as requested by the change specification. The *Candidate Impact Set* (CIS) is the set of system elements that are potentially affected by the change. This information is based on the traceability links established in the system. Bohner (1996) distinguishes between directly impacted elements in the system and indirectly impacted ones. An element, he uses the term software life-cycle objects (SLOs), is potentially directly impacted if there is a direct traceability relation in the traceability matrix (see Figure 3.3). Bohner defines an SLO as any artifact such as variables or requirements. Because the traceability does not necessarily mean that the change propagates to that component, he introduces the set of *False Positive Impacts* (FPIS), and creates loops in the process causing the *Discovered Impact Set* (DIS) to grow over time. In addition, he mentions that even if a reachability graph is generated starting from the changed element, in most software projects the whole system is connected. To deal with this problem he describes two approaches, a *structural* one and a *semantical* one. The structural approach creates a distance matrix between one SLO and the other elements in the system. He suggests investing more effort in the analysis of closely related components, because, he argues, the probability of a change propagation is dependent upon the distance of two elements. To reduce the number of false positives any further, Bohner recommends including more semantic information in the impact analysis process. With this semantic information he refers to the type of relationship between SLOs (e.g., a variable defined within a class or a decomposition relationship between requirements). He does not provide any further details on how to use this information. Approaches using the semantics of software can be found in the



Software Change Proposals
Software
Examine Software and Change Speciifcation
Starting Impact Set (SIS)
Trace Potential Impacts
Candidate Impact Set (CIS)
False Positive Impact Set (FPIS)
Perform Software Change
Actual Impact Set (AIS)
Discovered Impact Set (DIS)
AIS = CIS + DIS - FPIS

(a) The Impact Analysis Process



(b) Traceability between Software Life-Cycle Objects (SLOs)

Figure 3.3 – Impact analysis process as suggested by Bohner (1996)

next section.

Sommerville and Sawyer (1997) mention a similar idea to improve the quality of explicit traceability impact analysis. They advise using a *Data Dictionary* that includes information about all names used in a system (such as a description of the element), where it is defined and used, who defined the name and when. This can apply to different types of entities, such as objects, process elements, attributes. Without explicitly naming it, he introduces design ontologies for software components. The process of maintaining this data dictionary is mainly manual as is the usage for conducting an impact analysis based on it. Still, he mentions that computer-aided software engineering (CASE) tools, can support the generation of parts of the data dictionary.

Current change management tools used in a systems engineering context like Reqtify (Dassault Systems, 2012), IBM Rational Change (IBM, 2012) or Atego Workbench (Atego, 2012) focus also on the traceability between requirements and/or system artifacts to perform impact analyses. If changes occur, these tools highlight the system artifacts directly or indirectly connected by tracelinks to a changed element. Figure 3.4 shows the analysis view of IBM Reqtify and Atego Workbench.

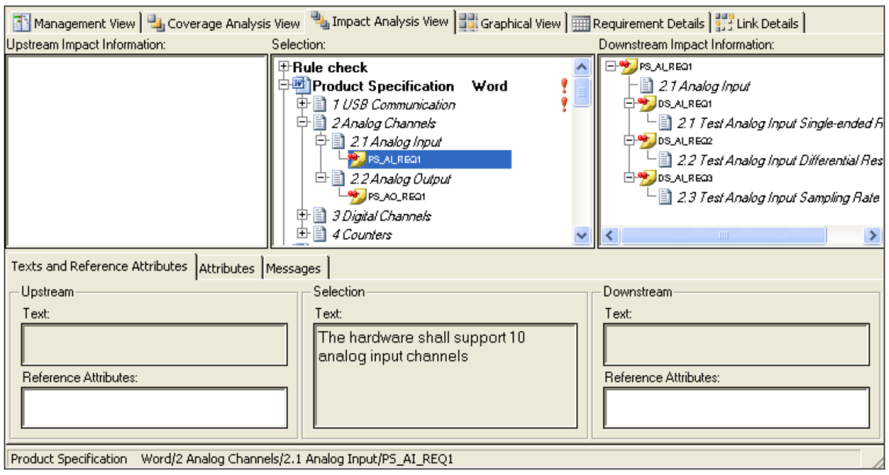
All the approaches presented in this section share the common problem that manually created traceability is not sufficient to clearly identify system elements that need to be changed from the system elements that are potentially affected by a change (Nuseibeh & Easterbrook, 2000). Therefore, it is still a manual and fault prone process to track the changes across the chain of traceability. The idea of using the distance in a traceability graph to prune results assumes that the change propagates in a breadth-first manner. Hence it would make sense to limit the investigation of elements this way. Still, from the experience of the author, the propagation behavior is nearly the opposite. If a requirement is changed, it is not necessary to change multiple or even all subrequirements to compensate for the created inconsistencies, but a single or very few requirements are selected. Taking the example of resource consumption, this behavior is based on the assumption that it is cheaper to save a greater amount of resources in a single subcomponent, than reducing the resource consumption of every subcomponent to a small degree. Furthermore, if there is no slack initially foreseen in the specification, the change needs to propagate either to the top level, to change the expected behavior of the system, or to propagate to an implementation, requiring modifications in hardware or software. That pure traceability based impact analyses do not perform well (e.g., in contrast to approaches based on changes histories) has also been identified by other researchers (Hassan & Holt, 2004). Nevertheless, it is a factor that improves quality if used together with other change propagation indicators (Lock & Kotonya, 1999).

However, approaches are needed that use more information than the manually created traceability to automatically and reliably prune the space of possible impact propagation candidates.

3.1.3 Implicit Traceability Impact Analysis

Implicit traceability impact analysis approaches are typically applied to software (Bohner, 1996). Podgurski and Clarke (1990) identified two types of relationships that can be extracted and then used for impact analysis: *control dependence* and *data-flow dependence*. The first type uses the call structure of programs. In this category we present *call-graph analysis* and *path based impact analysis* in this section. The later category analyzes the use of variables within the program, such as the *program slicing* technique.

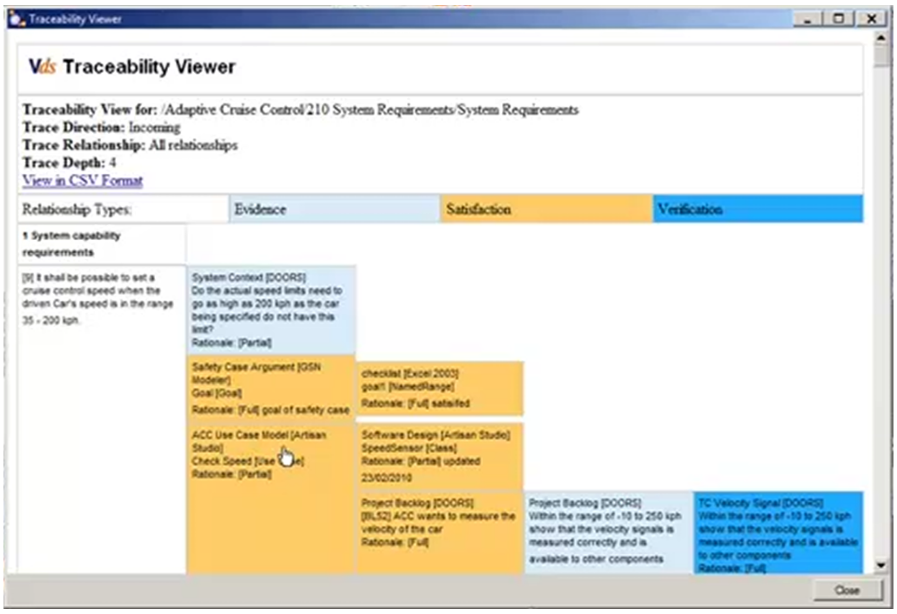
Ryder and Tip (2001) presented a change impact analysis for object oriented software based on call graphs. A call graph (Ryder, 1979) for Program P consists of the nodes that represent methods or procedures and edges that represent call relationships between them. Algorithms exist (Tip & Palsberg, 2000) that are able to compute a call graph for various programming languages. Their change impact analysis is based on eight atomic changes, that are listed in Table 3.2. While most of the changes are self-explanatory, the LC changes related to dynamic method binding (method dispatching) of object oriented programs. To tackle the problem of inheritance she extends classical call graphs with an additional lookup table to store how calls to objects resolve, depending on their runtime type. The triples in the table are of the form ⟨*runtimeReceiverType,*



The hardware shall support 10 analog input channels

Product Specification Word/2 Analog Channels/2.1 Analog Input/PS_AI_REQ1

(a) Impact analysis view of IBM Reqtify, for each element upstream and downstream impact is highlighted



Traceability View for: /Adaptive Cruise Control/210 System Requirements/System Requirements
Trace Direction: Incoming
Trace Relationship: All relationships
Trace Depth: 4
View in CSV Format

Relationship Types:	Evidence	Satisfaction	Verification

1 System capability requirements

[9] It shall be possible to set a cruise control speed when the driven Car's speed is in the range 35 - 200 kph.

System Context [DOORS] Do the actual speed limits need to go as high as 200 kph as the car being specified do not have this limit? Rationale: [Partial]

Safety Case Argument [GSN Modeler] Goal [Goal] Rationale: [Full] goal of safety case

checklist [Excel 2003] goal1 [NamedRange] Rationale: [Full] satisifed

ACC Use Case Model [Artisan Studio] Check Speed [Use Case] Rationale: [Partial]

Software Design [Artisan Studio] SpeedSensor [Class] Rationale: [Partial] updated 23/02/2010

Project Backlog [DOORS] [BLS2] ACC wants to measure the velocity of the car Rationale: [Full]

Project Backlog [DOORS] Within the range of -10 to 250 kph show that the velocity signals is measured correctly and is available to other components

TC Velocity Signal [DOORS] Within the range of -10 to 250 kph show that the velocity signals is measured correctly and is available to other components Rationale: [Full]

(b) The impact analysis within Artego Workbench provides a traceability view of elements to a specified trace depth, 4 in this picture.

Figure 3.4 – Impact analysis as provided by current industrial traceability tools

Abbrev.	Description
AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of method
LC	Change virtual method lookup
AF	Add a field
DF	Delete a field

Table 3.2 – The atomic changes for object oriented call graph based impact analysis of software. Source: Ryder and Tip (2001)

Operator	Description
$A < B$	A is a direct descendent of B
$A \leq B$	A is a direct descendent of B, or $A = B$
$A \leq^* B$	B is an ancestor of A, or $A = B$
$A <^* B$	B is an ancestor of A, but $B \neq A$

Table 3.3 – Notation for inheritance relations. Source: Ryder and Tip (2001)

staticMethodSignature, actualMethodBound⟩. The formal definition is given in Figure 3.5 based on the inheritance notation depicted in Table 3.3.

Based on these change types the impact analysis is performed to determine which test drivers are impacted by a change. Ryder defines test drivers as classes that exercise parts of the program and are used for regression testing. The first operation of the impact analysis is to determine the changes in the code (e.g., by using a tool like *diff*) and split them up into partial ordered atomic changes. Ryder claims that all changes can be split up using their selection of atomic changes.

Then, for a given set of test drivers $T = \{t_1, \ldots t_n\}$ the methods tested by t_i are denoted $N(P, t_i)$ (nodes in the call graph) and the relations respectively $E(P, t_i)$ (edges) and, hence, form the call graph for t_i. The affected tests (T_A) by a set of changes A are the tests that refer to a changed or deleted method ($CM \cup DM$), or are affected by a change in the dynamic binding of methods (virtual dispatch). The formal definition of T_A is given in figure 3.6.

Compared with the manually created traceability approaches, the call graph based approach is much more detailed, and the relations (i.e., the calls between methods) are extracted automatically, limiting the probability of failures in the traceability matrix.

$$\begin{aligned}
\mathit{Lookup} &= \{\langle C, A.m, B.m\rangle | \text{class A contains virtual method m, } C \leq^* B \leq^* A, \\
&\quad \text{class B contains virtual nethod } m, \\
&\quad \text{there is no class } B' \text{ that contains method } m \text{ such that } C \leq^* B' <^* B\} \\
\mathit{LC} &= \{\langle a, b\rangle\} | \langle a, b, c\rangle \in \{(\mathit{Lookup}_{old} - \mathit{Lookup}_{new}) \cup (\mathit{Lookup}_{new} - \mathit{Lookup}_{old})\}
\end{aligned}$$

Figure 3.5 – Definition of lookup and LC. Source: Ryder and Tip (2001)

$$\begin{aligned}
T_A &= \{t_i | t_i \in T, N(P, t_i) \cap (CM \cup DM) \neq \emptyset\} \cup \\
&\quad \{t_i | t_i \in T, n \in N(P, t_i), b \rightarrow_B A.m \in E(P, t_i), \langle B, X.m\rangle \in LC, B <^* A \leq^* X\}
\end{aligned}$$

Figure 3.6 – Definition of the affected test drivers T_A given a set of change A

The presented approach is only usable for a syntactic impact analysis. The test that are not part of T_A are sure to compile, but it cannot be claimed that the behavior of the system is still correct or not. Changes in the behavior can possibly propagate further through the system, which is not covered by the analysis. Still, compared with classical traceability based impact analyses, some changes can be identified as nondestructive, such as adding an empty class. The most frequent use of an impact analysis is the change of a methods body. All tests that are called from within this body are considered to be potentially impacted (see Figure 3.6). This overestimation is similar to the analyses presented in Section 3.1.2.

Another technique to conduct change impact analysis on software is program slicing. Program slicing has been introduced by Weiser (1981), when he defines this approach as "a method used by experienced computer programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form that still produces that behavior. The reduced program, called a 'slice,' is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior." Hence, a slice $S(v, n)$ of program P, with v being a set of variables and n a statement in P is called a *slicing criteria.* Slicing tools, both academic and commercial, are available; an overview of some of them can be found in (Korpi & Koskinen, 2007).

Gallagher and Lyle (1991) presented an approach to use these program slices for a software change impact analysis. They introduce a new type of slice, a so-called *decomposition slice* $S(v)$, which includes all the statements of program P which are in the slices of $S(v, n)$, where n are all statements where v is part of an output ($\mathit{output}(P, v)$) or the last statement (last) in P.

$$S(v) = \bigcup_{n \in N} S(v, n) | N = \{\mathit{output}(P, n) \cup \mathit{last}\}$$

Furthermore, they define *output-restricted* slices, as slices where all output operations are removed. Two output restricted decomposition slices $S(v)$ and $S(w)$ are considered independent if $S(v) \cap S(w) = \emptyset$. In addition, if $v \neq w$ and $S(v) \subset S(w)$ the slice $S(v)$ is defined as *strongly dependent* on $S(w)$. An output restricted slice that is not strongly dependent on any other slice is called a *maximal* slice. If $S(v)$ and $S(w)$ are output-restricted decomposition slices of Program P, statement in $S(v) \cap S(w)$ are considered as *slice dependent statement*.

Based on these definitions they built a framework that considers changes in a decomposition slice. They consider only two types of changes in programs, deletions and additions. A change of an existing line is handled as both, first a deletion, then an addition. They provide four rules of how to deal with changes:

- Independent Statements may be deleted from a decomposition slide, because they do not affect the computation of the complement.
- Assignment statements that target independent variables may be added anywhere in a decomposition slice, since they are unknown to the complement.
- Logical expressions and output statements may be added anywhere in the decomposition slice, since they do not even affect the computation of the changed slice.
- New Control statements that surround any dependent statement will cause the complement to change.

Furthermore, if the complement is affected, Gallagher and Lyle also provide an algorithm for how to support the developer in incorporating the change.

They elaborate extensively on the properties of decomposition slices, but could be more precise with respect to the changes. They provide specific rules like adding an assignment statement or changing the control structure. It is hard to judge if their chosen classification is complete. Although there exist extensions to slicing, like the integration of macros in C/C++ (Vidács, Beszédes, & Ferenc, 2007), it is an assumption of all slicing approaches that the developer is maintaining the complete code base. The decomposition slices introduced by Gallagher and Lyle separate the code on a per-line basis, which is an unsuitable separation for allowing shared work across multiple developers. Hence, the approach is not applicable for compositional impact analysis. Still, it simplifies the handling of changes for the individual developers, since they can implement the changes on a much smaller program, which increases the overview and hence reduces faults.

The problem of both approaches, call graph analysis and static program slicing, is that the result presented to the developer is much bigger than is actually needed. To reduce the unnecessarily identified elements Law and Rothermel (2003) introduced a new approach called *path based impact analysis*. The idea of that technique is to consider only those procedures that are on the execution path of a changed procedure P. That is, the affected procedures are either still on the program call-stack after P returns or are called after P and potentially use values returned by it.

They instrument binaries to get call traces based on real executions of the program. Since it is possible to instrument binary code it is not necessary for their approach to have the source code of the program available. A trace they consider might look like this:

$$MBrACDrErrrrx$$

Capital letters represent calls to procedures, lower case r represents a return of a procedure and the x represents the end of the trace. The analysis is based on searches on these traces. They distinguish between *forward* and *backward* searches in the trace. With forward searches they can determine which functions are called directly or indirectly by P. These are important since they might use values that have a different meaning after the changes in P and consequently produce a wrong output. The same applies for procedures that are called after P has returned. With backward searches they can determine the procedures in which P returns into and, hence, use data that has been altered and requires further adaptations.

This technique is a dynamic technique, therefore highly dependent upon the test-cases used to generate the traces. Even though they reduce the set of possible affected procedures, they introduce the risk of missing important propagations, because there was no test-case that has the needed behavior. Therefore the approach is not suitable for safety critical systems because the probability of missing relations is currently not precisely explored.

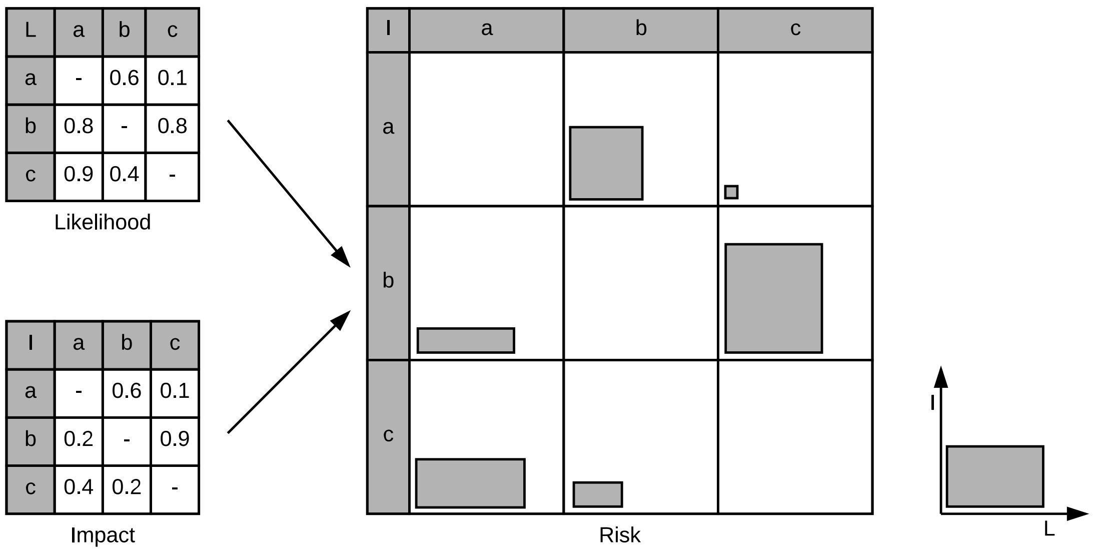
3.1.4 Impact Analyses using Change Histories

Most of the implicit traceability impact analysis techniques are only applicable for software and it is currently not possible to judge the accuracy of the approaches. Hence, multiple different probabilistic techniques have been proposed that rely on previous changes on the current or similar systems and try to quantify the reliability of the impact determination.

Clarkson et al. (2004) have proposed a technique based on a Design Structure Matrix (Steward, 1981). This Matrix is very similar to a traceability matrix as presented by Sommerville and Sawyer (1997) (see Table 3.1). In addition, the design structure matrix may also be used to represent relations between design tasks.

Before the impact of a change can be determined, it is necessary to perform an initial analysis of the current and similar older systems. This initial analysis is separated into three stages. In the first stage a product model is created. This activity is basically concerned with creating a decomposition of the system into smaller subsystems, encapsulating parts of the functionality. Clarkson et al. recommend using not more than 50 subcomponents for a system.

In the second stage the likelihood of a change propagation and the degree of impact is estimated and entered in a *design structure matrix* (DSM) (see Figure 3.1). The likelihood of change propagation is defined as the average probability that a change in a component leads to a change in another component. The impact is the average proportion of work that needs to be re-done if the change occurs. These matrices are



L	a	b	c
a	-	0.6	0.1
b	0.8	-	0.8
c	0.9	0.4	-

I	a	b	c
a	-	0.6	0.1
b	0.2	-	0.9
c	0.4	0.2	-

Figure 3.7 – A graphical product risk matrix. Source: Clarkson, Simons, and Eckert (2004)

filled based on data that has been collected during previous builds of similar systems or expert knowledge.

Changes can propagate directly to another component or indirectly across multiple other components. Hence, in the third stage of the initial phase the different paths are evaluated on which a change may propagate between components. As a result, the risk matrix is updated according to the possible propagation paths. The combined risk matrix may be represented in a graphical form (see Figure 3.7) so that the influences of likelihood and impact can still be identified.

After this initial phase, the effect of incoming changes can be estimated. After identifying the component that needs to be changed, the possible impacts can be seen in the product risk matrix. To better display the different affected components Clarkson et al. suggest a logarithmic case plot, which also represents the affected systems, but has the benefit that components with an equal risk are located on a straight line (see Figure 3.8)

Clarkson et al. evaluated the approach on various changes applied to a helicopter from GKN Westland Helicopters and had predicted the changes in a reasonably good way. Still, the main contribution of this approach is the visualization of risks and impacts. The critical input are the design structure matrix and the estimates for likelihood and impact that are extracted from already performed changes. Hence, the quality of the approach does depend mostly on the quality of the collected data and the experts involved in the process.

Another probabilistic approach has been developed by Lock and Kotonya (1999), who address the problem of identifying the relevant propagation path from the huge set of possible paths, which are identified using various traceability techniques. In the first

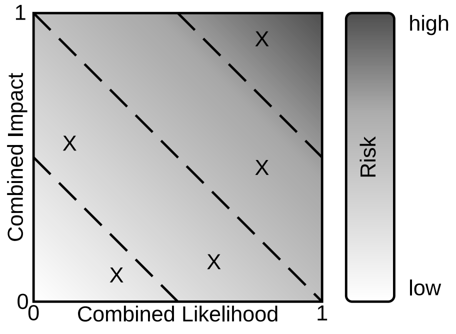


Figure 3.8 – A case risk plot. Source: Clarkson, Simons, and Eckert (2004)

step of their approach, they collect traceability information of the system from multiple sources.

The first source is *pre-recorded traceability* information. This source refers to the explicit traceability presented in Section 3.1.2 of this thesis, which is the class of manually created traceability links by the developers. The second source is a dependency extraction from behavioral models (see Section 3.1.3), which use forward and backward search of calling relationships to extract possible propagation paths. The third source is called *plain experience analysis*. This source uses the recorded data of previous changes to the system. Each change and its recorded impacts provides a small fraction of the traceability relation. The source is called "plain," since the identified impact paths are used as is; similarities between the paths are not evaluated. In contrast to the first two sources, the results of the experience analysis may already contain probabilistic information. The fourth source of traceability information is a new technique developed by the authors called *extrapolation analysis*. This analysis uses a partial set of direct impacts, as collected by the plain experience analysis, and tries to calculate the direct and indirect impacts for the whole system. The main idea is to identify previous changes that are similar to the current change based on the directly impacted elements. Hence, it is likely that the indirect impacts will also be similar. The last source of information is a so-called *certainty analysis*. The certainty analysis captures the belief of the developers as to how accurate a specification element is. This completeness is expressed within two values. First, the *degree of definition* (DoD), which is a value between 0 and 1 indicating how complete the element is defined. Second, the *certainty of definition*(CoD), is defined as the confidence that the entered information is correct. The resulting total certainty is defined as $C_{total} = DoD \cdot CoD$. This information, representing the reliability of an possibly impacted element, is an indicator which impact prediction is more likely to be correct.

The proposed integrated framework uses all the above mentioned information to select the most probable impacts from the set of candidates. Hence, all traceability information is combined into a traceability structure. A probability P_{path} is assigned to each of the propagation paths. If multiple relationships exist between entities (e.g., pre-coded and

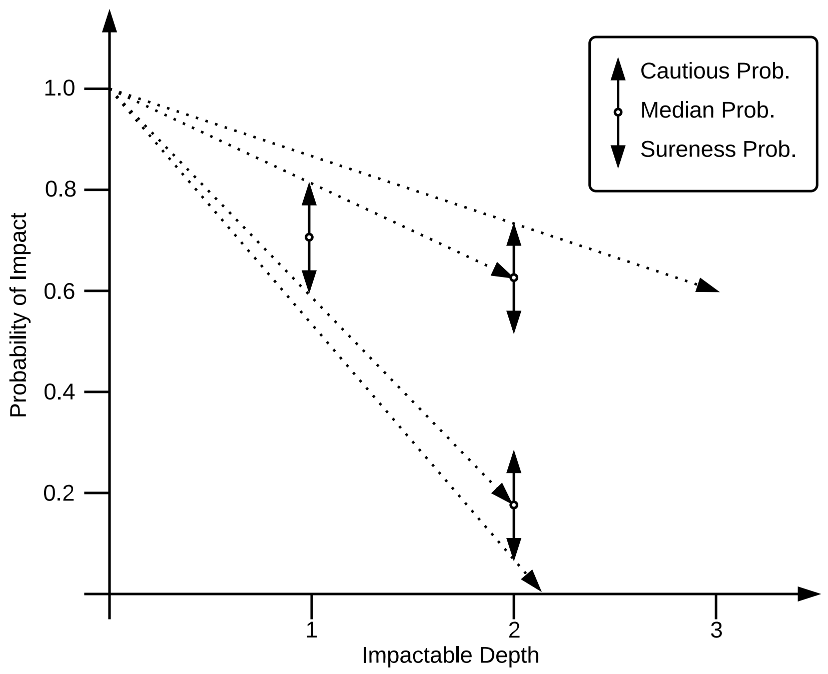


Figure 3.9 – Influence of sureness and cautious probabilities on total propagation probability

extracted from behavioral models) the probability will increase. The probabilities gained from previous experience can be used identically. To the values gained from this approach an adjustment is proposed using the collected certainty values. Two probability values are derived using the certainty:

- The *sureness probability* is the likelihood for a change to propagate based on the information available. The sureness probability P_{sure} is hence defined as $P_{path} \cdot C_{path}$. That is, if the information of an element is not reliable, this will reduce the path probability.
- The *cautious probability* is representing the risk of impact based on the information available. The cautious probability is defined as $\neg P_{cautious} = \neg P_{path} \cdot C_{path}$.

Figure 3.9 highlights how these values can be used to characterize the probabilities of impacted elements. In addition the authors describe in detail how to combine the results from the various sources, how to resolve duplicates and how to represent the cyclic nature of change propagation. Also, they present a variety of techniques for pruning the space of possibly affected components (e.g., by imposing a probability cut-off or focusing on a special behavioral aspect of the system).

The approach integrates many of the available data in a common model to assess the probability of a change impact propagation. Unfortunately they have not provided any evaluation results. Nevertheless, many of the used data are still dependent upon the judgment of experienced engineers.

Hassan and Holt (2004) analyzed different change propagation heuristics for software. They distinguished between the *data source heuristics* of how to identify change relations between entities and *pruning techniques* that help to select the most appropriate candidates from a set of potentially impacted elements. They analyzed the typical sources, such as call dependencies between methods, but also had a closer look at uncommon techniques such as *historical co-change* (elements that are changed frequently at the same time), *developer data* (code that is frequently edited by the same developer) or *code layout* (entities that are encapsulated together, e.g., in a directory in the file-system or a class). They discovered based on an analysis of large open source software projects, that the historical co-relation between changes, and the structural grouping of elements in the same file-system directory or file are much more accurate than call dependencies between software methods. However, the idea of using the developer as an indicator for changes, that is, that files edited by the same person are related and therefore it is likely that change propagates, has been proven false. They built a hybrid heuristic using the gained results that performed well in comparison with the single parameter heuristics.

More probabilistic approaches exist, such as an extension to program slicing developed by Santelices and Harrold (2010). Nevertheless, the drawback of all of these approaches is the remaining probability, that a propagated change is missed. If the impact analysis shall be used in a certification process to determine which verification and validation activities need to be re-run, the probability for remaining effects needs to be at least as high as if all tests of the system have been re-run. Nevertheless, to achieve a similar probability for the overall system, the impact detection probability needs to be significantly larger than the probability for each individual test in the system. (On this topic see the evaluation results in Section 5.) Still, the best hybrid approach evaluated by Hassan and Holt (2004) featured a recall of ≈ 0.5 and a precision of ≈ 0.5, that is, they suggest half of the actual impacts and that half of their suggestions are correct. This accuracy is similar to testing and review methods, which ranges between 40% and 60% detection accuracy (Runeson, Andersson, Thelin, Andrews, & Berling, 2006; Boehm & Basili, 2005) and can be significantly increased by automated testing techniques such as requirements based testing. Hence, the accuracy of the impact detection approaches is not sufficient for pruning verification and validation activities in a change management process.

3.1.5 Impact Analyses Avoiding Recertification

There are only very few approaches available to avoid a complete re-certification of a system after changes. Buckley, Mens, Zenger, Rashid, and Kniesel (2005) as well as Kilpinen (2008) mention *refactoring* (Mens & Tourwe, 2004; Fowler, Beck, Brant, Opdyke, & Roberts, 1999) as an impact analysis technique to fully automatically deal with changes in software. Opdyke (1992) defines refactoring as "reorganization plans that support change [...] [and] do not change the behavior of a program; that is, if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same." With refactoring some changes can be implemented automatically (Casais, 1994), like replacing library calls to optimize the memory usage (Buckley et al., 2005).

Still, the types of operations are limited. Opdyke (1992) lists eight types of refactorings that have been identified while analyzing the evolution of a complex software over two years:

- defining an abstract superclass of one or more existing classes
- specializing a class by defining subclasses, and using subclassing to eliminate conditional tests
- changing how the whole/part association between classes is modeled, from using inheritance to using an instance hierarchy of aggregates and their components
- moving a class within and among inheritance hierarchies
- moving member variables and functions
- replacing a code segment with a function call
- changing the names of classes, variables and functions
- replacing unrestricted access to member variables with a more abstract interface

This list, using C++ terminology, indicates that the possibilities applying refactoring are limited. Still, to implement more complex changes the application of refactoring techniques might be part of the whole process and reduce the introduction of faults in the software.

Another approach to avoid full re-certification has been presented by Nicholson et al. (2000) addressing explicitly safety critical systems. They distinguish between two types of systems. First, *Federated Systems*, where each function or application is hosted on separate hardware units, that communicate with each other. Still, these units are considered independent from the point of view of certification. Second, *Integrated Modular Systems* (IMS) are especially built to allow an easy update of the system and reduce the costs of re-certification later on. Their re-certification approach is based on the use of an IMS. Nicholson represented the safety case in the *goal structuring notation* (see Section 4.1.2). They presented eight different types of changes (e.g., a change of an application where the effects are contained within a single partition or changes to the underlying hardware platform). For each of these change classes a process and change recovery actions need to be defined, which are part of the initial certification of the product. The concept underlying this approach is the use of equivalence, that is, the certification of "families" of variants. The recovery actions shall prove that the modified systems argumentation is equivalent to the old argument. This process is depicted in Figure 3.10.

To identify to which category a particular change belongs, they propose a recursive, three phase impact analysis (see Figure 3.11). In this very generic process the potentially impacted elements are identified according to rules. However, these rules are not further detailed. In a second step the actual impact needs to be determined, which is also not described in any more detail. Still, the basis for the impact analysis is the availability

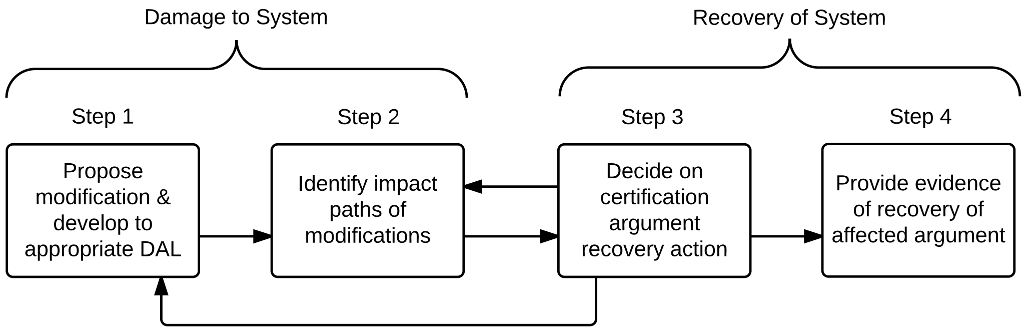

Damage to System
Recovery of System
Step 1
Step 2
Step 3
Step 4
Propose modification & develop to appropriate DAL
Identify impact paths of modifications
Decide on certification argument recovery action
Provide evidence of recovery of affected argument

Figure 3.10 – Process of using recovery actions to show equivalence of the new safety argumentation compared with the one prior to the change, according to Nicholson, Conmy, Bate, and McDermid (2000)

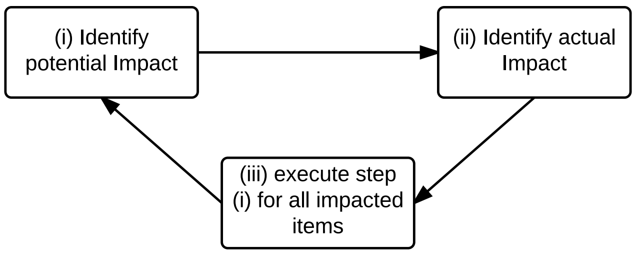

(i) Identify potential Impact
(ii) Identify actual Impact
(iii) execute step (i) for all impacted items

Figure 3.11 – A very generic, recursive process of conduction of the impact analysis within an IMS, according to Nicholson, Conmy, Bate, and McDermid (2000)

of slack in the system. This stems from the fact that IMS are built to be extended and modified after production, hence the available resources are not used until a critical limit has been reached. This means that, for example, the memory or the processing power of the underlying hardware, is over-dimensioned at the point of initial release. By the consequent use of service oriented interfaces and hardware abstraction layers it is possible to contain the change within a partition without complicated techniques. Still, for example, in the case of the change of timing requirements, it needs to be proved that the slack is sufficient to cover the needs of the change.

Although the approach of Nicholson is applicable for avionic systems, it cannot be transferred to the automotive domain. While the presented approach deals with the problem of altering a released system, that is especially built for later updates, in the automotive domain later changes to a released system are rare and not part of the re-qualification. The problem in the automotive domain stems from the evolution of one series to another, hence slack is non-existent in a very competitive market. Furthermore, the supporting techniques to separate functions are not always available.

3.2 Gap Identification and Goals for Impact Analysis

From the various impact analysis techniques presented in the previous section none is suited to exclude unaffected verification activities in a functional automotive safety concept from being re-executed. Pure traceability based approaches are not able to determine a stop criterion and overestimate the change in a magnitude comparable to the complete system. Although being able to restrict the changes with a stop criterion, software based impact analyses are not suited for two reasons. First, logical system descriptions are not necessarily fully implemented in software, and second, the approaches target compilability and are not able to reason on the semantically correct behavior. Probabilistic impact analysis approaches are able to give estimates of how likely a propagation will occur, but the precision of these approaches is not sufficient to establish the same level of confidence for the whole system as if all verification activities were executed again. To achieve this, the results need to be significantly more precise than the average probability of a system or integration test to miss a fault. Since these approaches are based on the existing changes performed to the same or similar system, the size of the available data history is a critical factor for the accuracy of the impact prediction. It is very unlikely that the available data will drastically increase, since most systems will not be produced long enough, before they are be replaced by a new development. The impact analysis presented for integrated modular systems is not restricted by the already mentioned flaws. It is applicable for systems, does provide a stop criterion and targets correctness of the system. Still, it is based on existing slack and isolation mechanisms between components in the system. For avionic systems it is likely that extensive modification will be performed to an aircraft, the lifetime of which is typically more than 30 years (Jiang, 2013). Hence, the manufacturers anticipate the need for additional space for later added upgrades in the system that can be used. For automotive systems this is currently not the case. Although, software updates are becoming more and more important since bugs in the recent past have led to accidents (CBSNEWS, 2010), only rarely will new features be added after production. Hence, the new impact analysis approach needs to be able to contain changes without the existence of extensive slack.

As a result, four main technical requirements need to be fulfilled:

- The new impact analysis shall provide a stop criterion for change propagation in the system.
- The new impact analysis shall be able to detect impacts on logical architectures.
- The new impact analysis shall use the semantical correctness of the system as the target for impact detection.
- The new impact analysis shall be able to contain changes with very little slack available in the system.

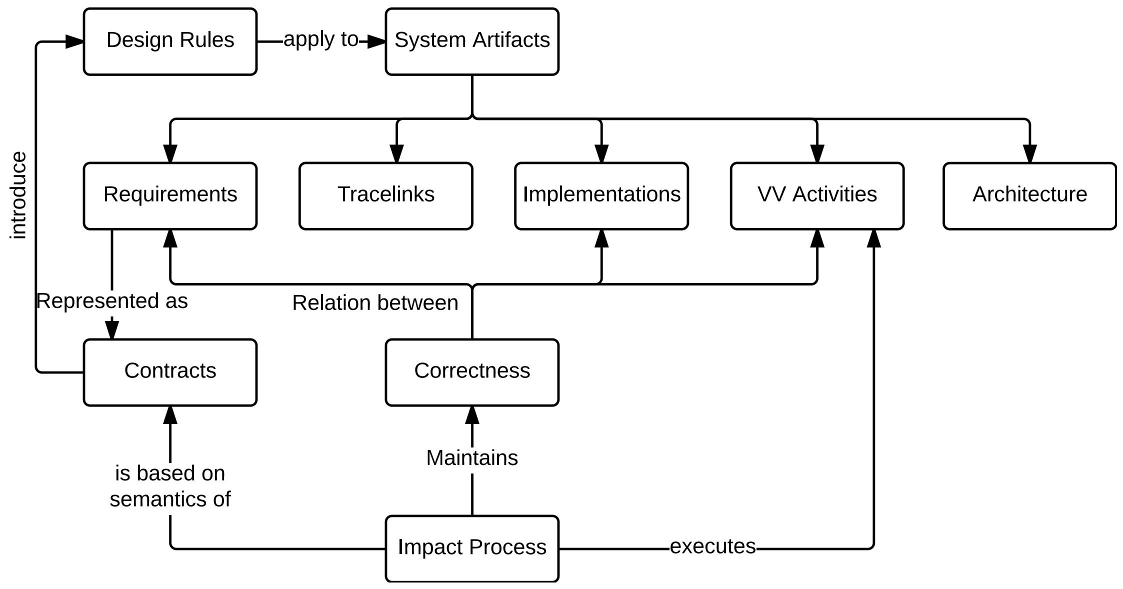


Figure 3.12 – Relations between entities involved in the impact analysis

3.3 Impact Analysis on Contract-based System

In this section a novel impact analysis approach is presented, to implement the requirements stated in section 3.2. Here, the process and necessary verification activities are detailed, while in chapter 4 the concrete language to express safety concepts is elaborated. Figure 3.12 depicts the relations between the elements involved in the process. The impact analysis process is based on an abstract system representation detailed in section 3.3.1 encompassing e.g. the requirements, expressed as contracts, the implementations and the architecture. Also the design rules of how to apply contract based design are described. The correctness criterion, which is the property that shall be maintained by the impact analysis is described in section 3.3.2.

3.3.1 System Representation

Lehnert (2011) used the set of supported design artifacts as a classification of impact analysis approaches. According to him only 13% of the approaches support multiple types of artifacts, like code, structural models, requirements or documentation. In our approach we want to support all relevant artifacts necessary to describe and analyze functional safety concepts. Hence we integrate behavioral, structural and process artifacts.

An overview of the artifacts is shown in figure 3.13, which are explained in detail in the following two sections.

The analysis process itself is not limited to specific languages or representation formats, therefore we base our approach on a abstract system model. The selection of the elements within this model is based on research performed to identify the common system engineering artifacts used for safety critical systems (Rajan & Wahl, 2013; Baumgart et

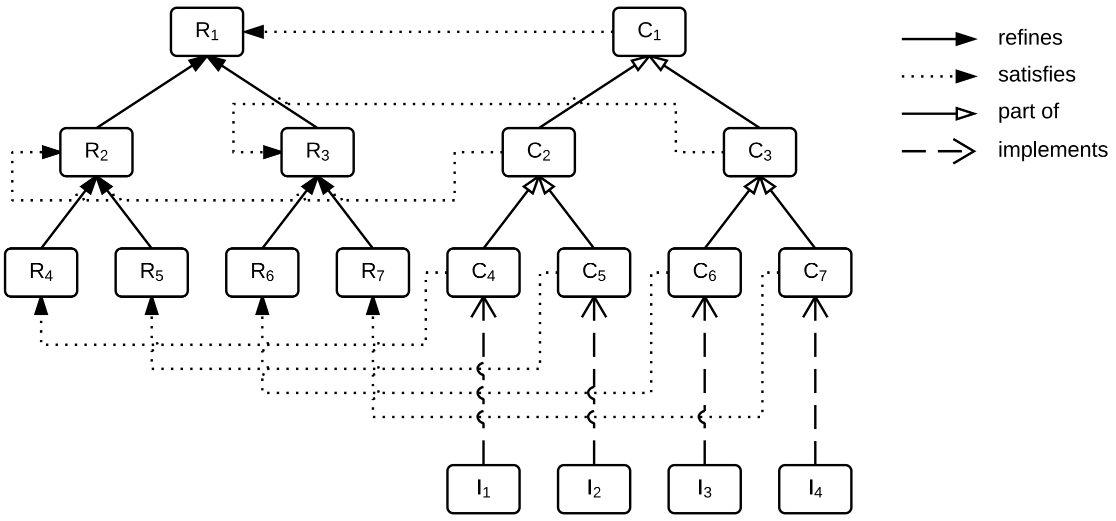


Figure 3.13 – Overview of the System Artifacts and Tracelinks repected by the Impact Analysis

al., 2011). In addition, the needs of the ISO 26262 (see Section 2.3) had to be considered. We chose to make the system representation as generic as possible, to allow an application in many scenarios that are potentially more specific.

System Entities

We identified three main types of artifacts to describe functional safety concepts (see Table 3.4). A structural model, sometimes also called architectural model (Baumgart et al., 2011). Typical structural models in the automotive domain are AUTOSAR (AUTOSAR GbR, 2014), EAST-ADL (ATESST2 Consortium, 2010) or SysML (OMG SysML, 2012). The most generic common element in these models is a *component* identifying a system part (or even the system in total). These components are de-composable, that is, they can be divided into subcomponents. Components have associated ports that represent the input and output values. The components are therefore used to describe the interface of the elements of a system. This system description is called the *preliminary architecture* in the ISO 26262 (see Section 2.3). The second main type is the description of the behavior of components. This description may have two different representations. On higher abstraction levels the behavior is represented as *requirements*, describing the intended behavior of the components. If the system is broken down to a sufficient degree the behavior can *implemented.* This implementation itself can have various forms, like source code, an executable model, or even a mechanical prototype. The concrete characteristics are not important for our approach. While the ISO 26262 explicitly forces the use of a requirements breakdown structure, it does not directly require the existence of behavioral models for the functional safety concept. Nevertheless, the ability to avoid hazardous events needs to be shown, and the standard provides an example note stating that “the

Entity	Description
Component	Decomposable, model-based representation of the interface of a system or system part. A component consists of ports that represent the inputs and outputs
Requirement	Requirements represent functionality or properties that the component it is attached to shall fulfill
Implementation	Implementations represent the concrete, executable behavior of a component. This might be software (code or functional model) or a hardware implementation.

Table 3.4 – Abstract entities used in the impact analysis process

ability to mitigate or to avoid a hazardous event can be evaluated by tests, trials or expert judgment; with prototypes, studies, subject tests, or simulations" (ISO 26262, 2011). Hence, executable models are needed to perform tests or simulations.

System Relations

As indicated by various authors (Ramesh et al., 1995; Ambler, 2002; Nuseibeh et al., 2000) the traceability in a complex system is very expensive to maintain. Hence, it is one of our goals to reduce the needed links between the system artifacts to a minimum. We base the impact analysis on three traceability links, as depicted in Table 3.5. These links exist in most of the analyzed system models (ATESST2 Consortium, 2010; OMG SysML, 2012; Baumgart et al., 2011) and represent the most basic relations in a system. Furthermore, the chosen tracelinks cover the relations defined between contracts (see Section 2.1.3). In addition, all the presented links are required by the ISO 26262 and therefore do not provide any additional effort.

The *satisfy* link establishes a connection between a requirement and a component. It indicates that a component shall have the behavior described by a requirement. It does not indicate that the component is actually fulfilling that requirement. This is indicated by a test case that proves the correct relationship between a component, the requirement and the implementation of the component. The *refine* link is created between requirements on different decomposition levels. It indicates that a requirement is broken down into finer granular requirements that cover in total the source of the link. Again, this relation indicates only that this relation shall exist; a V&V case needs to prove the correctness of this link. The *implementation* link connects a component with its implementation. In the current process we assume that there is a single implementation for a component. This is a realistic assumption, since the interface of the implementation

Tracelink	Description	Source	Target	Cardinalities
Satisfy	Connects a requirement with a model component.	Comp.	Req.	1,1..*
Refine	A requirement is decomposed into multiple more concrete requirements.	Req.	Req.	1,1..*
Implementation	Connects an implementation to a component.	Impl.	Comp.	1,0..1

Table 3.5 – Tracelinks necessary for our impact analysis

needs to fit to the interface of the component. Internally, the implementation might consist out of many submodels, or source code files, which is in this representation abstracted towards a single implementation. Each implementation needs to have one associated component, while not every component is expected to have an implementation.

The source and target types are introduced in Table 3.5 to define the semantics of the link, in the actual impact analysis process the links are expected to be traversed in both directions.

We use the following notation for traversing links in this work. Each link can be traversed in both directions, each direction is indicated by an arrow above the function name. We define the function $\overrightarrow{S}(c)$ to return all requirements connected to a component c. The function $\overleftarrow{S}(r)$ returns the component a requirement is attached to. A requirement can only be attached to a single component. The refinement relation can be traversed using $\overrightarrow{R}(r)$ and $\overleftarrow{R}(r)$. $\overrightarrow{R}()$ returns the refined requirement, while $\overleftarrow{R}()$ returns the refinees. Similarly the function $\overrightarrow{I}(i)$ returns the component that is connected to implementation i and $\overleftarrow{I}(c)$ returns the implementation that is connected to component c.

Using Contracts

Contracts provide a semantic framework to reason on relations between requirements and implementations (see Section 2.1 for an introduction into contract semantics). One essential relation is refinement ($\preceq$), which is a binary relation between requirements that indicates that one requirement is a correct specialization of another. Together with the parallel composition ($\otimes$) virtual integration testing (VIT) can be performed. While using VIT, real integration testing can be completely omitted in favor of a set of refinement and satisfaction analyses. For the system represented in Figure 3.13, this means, that if is has been shown that implementation I_1 to I_4 are correctly representing the behavior expressed by the requirements R_1 to R_4, and all requirements refinement relations are correct, then it follows that the system composed from the implementations fulfills all stated requirements.

Hence, contracts are the favorable form to represent the systems requirements R. In order to use contracts in the development process a few constraints need to be complied to. Most of all, a structured handling of requirements is needed. All requirements need to be organized in a requirements breakdown structure established by refine links:

Structural Constraint 1. *Each Requirement r needs to be either a top-level requirement of the system or it refines exactly one requirement q:* $r \in \overleftarrow{R}(q) \rightarrow \forall_{s \in R | s \neq q} : r \notin \overleftarrow{R}(s)$

Although the requirements break-down is often not obeyed completely in practice (Gotel & Finkelstein, 1994), the ISO 26262 requires this kind of tracing anyway by proposing a "hierarchical approach by which the safety goals are determined as a result of the hazard analysis and risk assessment. The functional safety requirements are then refined from the safety goals" (ISO 26262, 2011). Nevertheless, as an additional constraint, this refinement needs to be represented in the architecture of the system as well:

Structural Constraint 2. *For two Requirement r and r′ for which holds that* $r' \in \overleftarrow{R}(r)$ *it needs to follow that there is a direct part relation between the associated components:* $\overleftarrow{S}(r') \in \overleftarrow{P}(\overleftarrow{S}(r))$*. This property also holds vice versa.*

There are also constraints on how the requirements need to be formulated. As stated in section 2.1, contracts are defined over the interface of a component. Based on a assumption on the input, a guarantee of the value of the outputs is given. This restricts the requirements to mention only values that are provided by the ports, directly connected to a component.

Structural Constraint 3. *The mentioned values* $\mathbb{S}(r)$ *in a requirement r are available on the components interface* $\square(\overleftarrow{S}(r))$*:*

$$s \in \mathbb{S}(r) \rightarrow \exists p \in \square(\overleftarrow{S}(r)) : p \overset{p}{=} s$$

This kind of specification is advised by many practitioners (see Sommerville and Sawyer (1997) or Oertel et al. (2013)) as well as requirements and system engineering standards (ISO/IEC, 2011; ISO/IEC 12207, 2008). Black box specifications ensure the writing of implementation free requirements, since it is not possible to name or describe internal elements of the component. The ISO/IEC 29148 (ISO/IEC, 2011) states the absence of implementation information as one of the main properties: "The requirement, while addressing what is necessary and sufficient in the system, avoids placing unnecessary constraints on the architectural design. The objective is to be implementation-independent. The requirement states what is required, not how the requirement should be met." Another benefit of the interface based specification is the avoidance of overlaps in the scopes of requirements. Hence, it is not possible that a requirement assumes properties in a distinct part of the system. Such requirements bear the danger of being not correctly assigned to the architecture.

In addition to the constraints on the requirements, there are also constraints on the architecture. We assume all components to be *well connected*. Although this is a typical

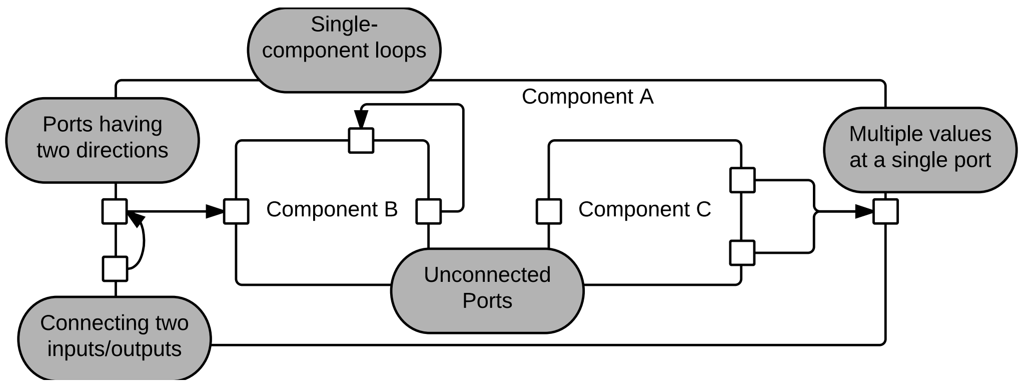

Single-component loops
Component A
Ports having two directions
Multiple values at a single port
Component B
Component C
Unconnected Ports
Connecting two inputs/outputs

Figure 3.14 – Faulty connectors between ports

constraint for all port based specification languages, there are some special restrictions that apply only in the context of contracts.

Figure 3.14 depicts incorrectly connected ports. Connections between two input ports, using a port simultaneously as an input and an output as well as not connected ports are generally considered as faults in the design. Loops on a single component and multiple connections to a single port are contract specific problems. A loop on a single component does not provide any benefit, since time can only evolve inside a component and the connections are transferring the data immediately, the information at the output and the input would always be identical. Hence, this way of modeling is not applicable to design control loops. If time shall evolve while sending a signal between components (e.g., by using a bus), the channel needs to be explicitly modeled using a dedicated component. This component may then introduce a delay. The argumentation for prohibiting multiple connections to an output port is very similar. Since the values are expected to be identical if they are connected, there are no semantics defined to “over-ride” a value if multiple ports are connected to one output. If such a behavior is needed, the overruling strategy needs to be specified for a dedicated component that has multiple input and one output port. On the other hand it is not a problem to connect an input port to multiple inputs.

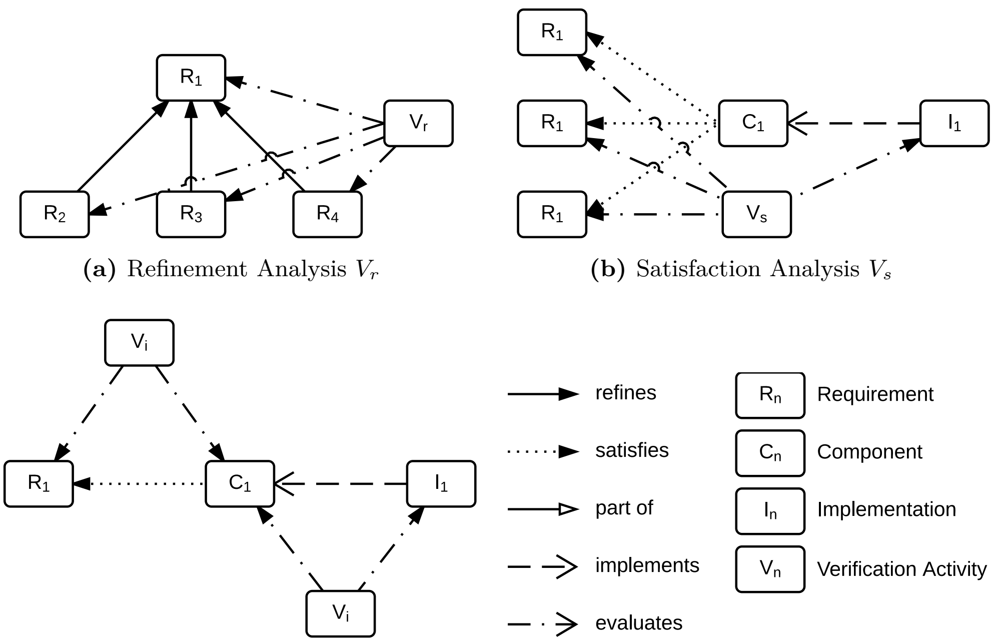
To avoid the necessity for a separate check of the connections, we define the values in the system only over its name. A direction is therefore also only assigned once, for both connected ports. Hence, we assume identity between ports having the same value.

Structural Constraint 4. *Two ports p_1 and p_2 on components c_1 and c_2 that are connected by a delegation or assembly connector, need to be named identical and have the same direction.*

Using this design rule, we avoid having to state further rules for the connectors between ports and can simply rely on the port names.

Verification Activities

Resulting from the set of tracelinks and the relations used between contracts, a set of verification activities can be selected, that allows analysis of the existing relations, so that



(a) Refinement Analysis V_r **(b)** Satisfaction Analysis V_s

(c) Interface Analysis V_i **(d)** Link and Component Type Overview

Figure 3.15 – The verification activities used in the impact analysis process

the impact analysis process specified in section 3.3.4 is able to make use of them. These are the necessary verification activities for the process to ensure system consistency. While in this section the verification activities are presented, the systems consistency criterion is described in section 3.3.2. Additionally to the here shown verification activities, there might be additional ones, like tests or reviews, that are linked to the elements. However, only the here mentioned V&V cases are used for the change propagation detection.

The *refinement analysis* proves the correctness of the split up of requirements. This analysis refers directly to the refinement relation on contracts described in section 2.1.3. Since refinement is defined on two requirements only, the refinement analysis is building the parallel composition R_p of the subrequirements $R_1 \dots R_n$ with:

$$R_p(R) = \bigotimes_{x \in R} x$$

The parallel composition needs to specify at least one possible solution, otherwise refinement is considered as failed, too. The result of $V_r(r_{top}, R)$ is defined as:

$$V_r(r_{top}, R) = \begin{cases} 1 & \text{if } R_p(R) \preceq r_{top} \wedge [\![R_p(R)]\!] \neq \emptyset \\ 0 & \text{if } R_p(R) \npreceq r_{top} \end{cases}$$

The satisfaction analysis indicates if an implementation `M` implements the behavior required by requirement r. This analysis directly refers to the satisfaction operator defined in Section 2.1. Hence, the analysis is defined as:

$$V_s(\mathtt{M}, r) = \begin{cases} 1 & \text{if } \mathtt{M} \models r \\ 0 & \text{if } \mathtt{M} \not\models r \end{cases}$$

Since the satisfaction analysis is based on requirements and implementations, it does not rely on components. The relation between the component c and its ports $\square(c)$ and the signals $\mathbb{S}(r)$ mentioned in the requirement r as well as the relation between the components ports and the implementation ports $\square(i)$ need to be checked separately. To avoid a re-mapping of names, we define two ports p_a, p_b or signals as port-equivalent ($\overset{p}{=}$) if the name and type (if available) match. This interface check is performed by the Interface analysis V_i(r, c):

$$V_i(r, c) = \begin{cases} 1 & \text{if } \forall s \in \mathbb{S}(r) : \exists p \in \square(c) : s \overset{p}{=} p \\ 0 & \text{if } \exists s \in \mathbb{S}(r) : \nexists p \in \square(c) : s \overset{p}{=} p \end{cases}$$

Furthermore, the interface check can be applied to components and implementations as well:

$$V_i(c, i) = \begin{cases} 1 & \text{if } \forall s \in P(i) : \exists p \in P(c) : s \overset{p}{=} p \wedge \forall p \in P(c) : \exists s \in P(i) : s \overset{p}{=} p \\ 0 & \text{if } \exists s \in P(i) : \nexists p \in P(c) : s \overset{p}{=} p \vee \exists p \in P(c) : \nexists s \in P(i) : s \overset{p}{=} p \end{cases}$$

3.3.2 Correctness as a Target for Impact Analysis

Impact analyses have different targets as seen in the related work section. For example, Ryder and Tip (2001) used a call-graph analysis to detect impacts that could compromise the compilability of source code. Nicholson et al. (2000) wants to detect impacts on resource usage like memory consumption, execution time or bus occupation. In contrast to these single property oriented impact analyses, we want to guarantee the correct semantic behavior of the total system with respect to all stated requirements. Correctness of requirements was extensively analyzed in the CESAR Project (Rajan & Wahl, 2013). A set of requirements A' is correct if "requirements in A' allow us to reach goal $g(A)$, i.e.: in a given context Γ if the requirements are all satisfied the goal $g(A)$ will also be satisfied. On the contrary, if one cannot prove that $g(A)$ is satisfied when A' is complete, then we can conclude that there are errors in A'. Consequently, if A' is not correct with respect to A then at least one requirement raises a problem in A'. In other words, this requirement cannot be satisfied in the context Γ for one of several reasons: it is false or its realization is unfeasible [...]." (Benveniste et al., 2011). Correctness requires completeness, which is given if there are sufficient requirements in A' to prove that the goal $g(A)$ can be deduced from A'.

In the context of this work, we can translate this definition to:

Definition 5 (System Correctness)**.** A structural consistent system consisting of a set of requirements R, a set of components C and a set of Implementations I is correct if:

$$\begin{aligned}
\forall_{r \in R} : V_i(r, \overleftarrow{S}(r)) = 1 \quad \wedge \\
\forall_{i \in I} : V_i(\overrightarrow{I}(i), i) = 1 \quad \wedge \\
\forall_{r \in R | \overleftarrow{R}(r) \neq \emptyset} : V_r(r, \overleftarrow{R}(r)) = 1 \quad \wedge \\
\forall_{i \in I} : V_s(i, \overrightarrow{I}(i)) = 1
\end{aligned}$$

Hence, the three verification activities (Refinement Analysis, Satisfaction Analysis and Interface Analysis) build the base for the correctness criterion of the system. We consider each tracelink an analysis target. The correctness criterion of the CESAR project can be established using our definition and the virtual integration corollary (see Section 2.1.4). For all subrequirements R_i of R, with their implementations $\mathtt{M}_i$, composed to $\mathtt{M}$, it holds:

$$\left[\bigwedge_{i=1}^{n} V_s(\mathtt{M}_i, R_i) \wedge V_r \left(\bigotimes_{i=1}^{n} R_i, R \right) \wedge \bigwedge_{i=1}^{n} V_i(R_i, \mathtt{M}_i) \right] \rightarrow V_s(\mathtt{M}, R) = 1 \wedge V_i(R, \mathtt{M})$$

If the implementations correctly implement the behavior of the subrequirements and the subrequirements are a correct refinement of the top-level requirement it directly follows that the composition of the implementations ($\mathtt{M}$) satisfies the behavior required by R. Note that the definition of satisfaction (see Section 2.1) requires the contracts and the components inputs and outputs to coincide. Hence, V_i has to hold for all subcomponents $\mathtt{M}_i$ and their connected Requirements R_i. To conclude that V_i also holds for R and $\mathtt{M}$, the subelements $\mathtt{M}_i$ need to be properly connected to each other and to $\mathtt{M}$. These structural constraints have been detailed in section 3.3.1. We consider components to be the implementation resulting from the composition of the subcomponents or implementations.

This relation can be applied recursively within the requirements breakdown structure to show the correct implementation of all requirements.

It needs to be mentioned, that virtual integration checking of the relations can only be performed if requirements are stated in a precise way. Hence, the semantics of the requirements need to be unambiguous. The formal language needed to represent requirements in a functional safety concept, is described in chapter 4. Also the concrete realization of the verification activities V_s and V_r are described in that chapter in the sections 4.5.2 and 4.5.1.

3.3.3 Change Operations

Eckert (Eckert et al., 2004) distinguishes between initiated changes and emergent changes. While changes in the first category have a cause outside of the system, such as a customer requirement change, emergent changes are necessary because of problems that have been identified during the development process. In this thesis we start with initiated changes and detect the emergent changes, which cause the change propagation. This implies, that we are starting from a consistent system, modify parts and want to establish a consistent system again.

The atomic elements that can be changed are requirements, implementations and components. Verification activities are represented as own entities in the implementation (see Section 6.1) but are in this section considered as the formal representation of system properties. Hence, they are part of the argumentation but cannot be modified by the user.

To maintain the system in a structural consistent state, we define the change operations in a preserving way. That is, that the structural consistency cannot be violated by performing changes to the system. Hence, many typical change operations, like deleting a requirement or adding a component, cannot be executed alone, but require further modifications, either on other system elements or on the traceability structure. The change operations and the additionally necessary actions on elements and traceability are listed in Table 3.6. The selected change operations are a subset of more possible operations, but are sufficient to create all structural consistent systems.

The modification of existing elements is a very common change in the evolution of existing systems. These types of changes do not alter the structural consistency of the system, hence no additional modifications to system elements and traceability links are necessary.

In contrast, adding a single element would taint the structural consistency of a system. If a requirement is added to the system, two different scenarios are possible. Either the requirement is added to an atomic component, or the requirement is added higher in the hierarchy of the requirements break-down structure. In the first case only the traceability needs to be corrected, by creating the needed refinement link to the corresponding top-level requirement and a link to the component that shall fulfill the new requirement. If a requirement is added higher in the hierarchy of the system, additional requirements need to be added along the refinement chain and linked to the corresponding component, until the lowest level of components is reached and an implementation is directly connected to that component. This is necessary to avoid requirements continuing to exist in the system that are not refined.

There are also two cases of how to add a component to the system. If a component is added to one of the atomic components in the architecture, the implementation of the extended component needs to be removed and an implementation needs to be added to the newly added component. Also a requirement needs to be created and linked to the added component. If a component is added on higher levels of the system hierarchy, a new requirement shall be created and linked to the new component. Also the refinement link to a requirement one level higher in the refinement structure needs to added. Furthermore, an implementation shall be added and linked to the new component. In both cases it needs to be noted, that a problem in identifying a proper top-level requirement for the newly added requirement indicates that the feature is added at the wrong position in the system hierarchy.

Adding an implementation is not necessary. Since there is only one implementation allowed for each atomic component, the structure of the system is modified by modifying the components. This is an operation that is needed in any practical scenario, but would

Operation	**Target**	**Element Updates**	**Traceability Updates**
Modify	Requirement	none	none
	Implementation	none	none
	Component	none	none
Add	Requirement (atomic)	none	add link to existing component, add link to existing top-level requirement
	Requirement (intermediate)	add at least one requirement for each level of components below the initially assigned component	link all requirement to components, update refinement links between requirements
	Component (atomic)	delete implementation of extended component, add implementation, add at least one requirement	create part link, link implementation to component, link requirement to component, create refine link for requirement(s).
	Component (intermediate)	add implementation, add at least one requirement	link implementation, link requirement(s), link component
Delete	Requirement	delete all subrequirements, delete all components and implementation that do not have associated requirements	delete all tracelinks connected to deleted elements
	Component (with siblings)	delete subcomponents, delete connected implementation, delete connected requirements,	delete all tracelinks to deleted elements
	Component (w/o siblings)	delete all subcomponents, delete all requirements, delete connected implementation(s), add new implementation	delete all tracelinks to deleted elements, add tracelink to newly created implementation

Table 3.6 – Change operations overview. For some operation and target combinations multiple integration scenarios exist

cause redundant description in this thesis.

Deleting elements also requires corrective measures to preserve the structural consistency of the system. If a requirement is deleted all refined requirements need to be deleted as well. In addition, the components and implementations that do not have associated requirements need to be removed. The tracelinks connected to deleted elements need to be deleted as well. To delete a component, also all subcomponents and their connected requirements need to be deleted. If there are sibling components to the deleted component a new implementation needs to be created to avoid atomic components existing without an implementation.

The selection of change operations is complete in a sense that all possible correct systems can be built using this operation, starting from an already existing correct system. Modifications to the existing component structure are performed by adding or removing requirements and the structure is modified by adding or deleting components.

Although changes in tracelinks are a possible change scenario, we do not explicitly address these changes. Each re-link of a tracelink can be represented as a removal and an addition of the connected system artifacts. In the following we use the term *change* to encompass modification, addition and deletion.

3.3.4 Impact Analysis Process

To establish a linear relation between the size of the change and the effort to re-verify the system, it is necessary to identify precisely which verification activities are affected by a change. The process of identifying these verification activities and establishing a consistent state of the system after changes is depicted in Figure 3.16.

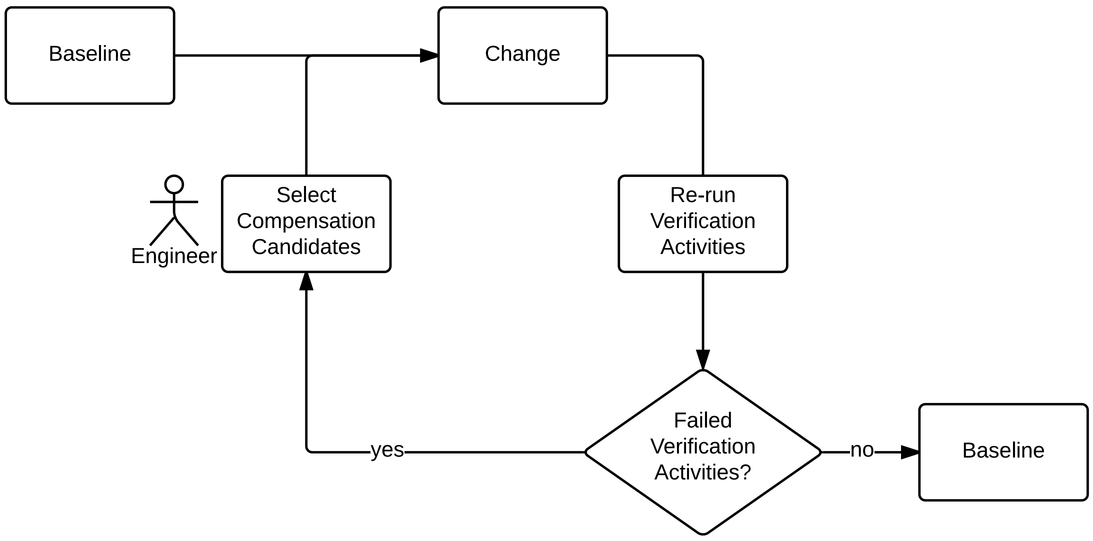

Baseline
Change
Engineer
Select Compensation Candidates
Re-run Verification Activities
Failed Verification Activities?
yes
no
Baseline

Figure 3.16 – Basic process for handling change requests

Starting from a correct system an initial modification of the system is performed. As extensively discussed in the related work section (see Section 3.1) this change might propagate and cause further changes to become necessary. To identify the affected elements by this change, the verification activities related to the changed element need to be re-executed. Table 3.7 lists the verification activities to be re-executed based on the element type that has been changed and the change operation. The verification activities are selected under the principle that all tracelinks connected to a changed element need to be re-evaluated. The supported types of changes, and their additional modifications of the system to maintain the structural consistency are listed in Table 3.6. These additional changes for an operation are marked in Table 3.7 with a plus-sign, indicating the added elements, and a minus-sign indicating removed elements.

Operation	**Element**	**Activities to Execute**
Modify	Requirement r	$V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$ $V_r(r, \overleftarrow{R}(r))$ $V_i(r, \overleftarrow{S}(r))$ $V_s(r, \overleftarrow{I}(\overleftarrow{S}(r)))$
	Implementation i	$V_i(\overrightarrow{I}(i), i)$ $\forall_{r \in \overrightarrow{S}(\overrightarrow{I}(i))} : V_s(r, i)$
	Component c	$V_i(c, \overleftarrow{I}(c))$ $\forall_{r \in \overrightarrow{S}(\overleftarrow{I}(c))} V_i(r, c)$
Add	Requirement r (atomic)	$V_s(r, \overleftarrow{I}(\overleftarrow{S}(r)))$ $V_i(r, \overleftarrow{S}(r))$ $V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$
	Requirement r_i (intermediate) + requirements R_c	$\forall_{r \in R_c} : V_s(r, \overleftarrow{I}(\overleftarrow{S}(r)))$ $\forall_{r \in R_c} : V_i(r, \overleftarrow{S}(r))$ $\forall_{r \in R_c} : V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$
	Component c (atomic) - implementation i_d + implementation i_a + requirement r	$V_i(c, i_a)$ $V_i(r, c)$ $V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$ $V_s(r, i_a)$
	Component c (intermediate) + requirement r + implementation i	$V_i(c, i_a)$ $V_i(r, c)$ $V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$ $V_s(r, i_a)$

Operation	**Element**	**Activities to Execute**
Delete	Requirement -requirement R_r	$V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$ [1]
	Component c (with siblings) - Requirement r - subcomponents C_s - implementations I_s - subrequirements R_s	$V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$
	Component c (w/o siblings) - requirement r - subrequirements R - subcomponents C - implementation I + implementation (i_d)	$V_s(\overrightarrow{R}(r), i_d)$ $V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$

Table 3.7 – Verification activities to be restarted based on the type of change. Additional changes according to Table 3.6 are marked with +/- signs

If requirements are changed, it is essential to check the effects on the existing refinement relations towards the top level $(V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r))))$ and towards the implementation level $(V_r(r, \overleftarrow{R}(r)))$. This allows semantical identification if there is a change propagation towards the top-level or the implementation-level in the requirements breakdown structure of the system. Hence, based on the semantics of the requirements a criterion exists to stop the change propagation in the system. This is not possible in pure traceability based impact analyses. Even impact analyses based on source code are not capable of analyzing effects on the semantic behavior.

In addition to the refinement check the compatibility with the component needs to be analyzed $(V_i(r, \overleftarrow{S}(r)))$. And, for the lowest level of requirements it also needs to be checked if they are still compliant to the implementation $(V_s(r, \overleftarrow{I}(\overleftarrow{S}(r))))$.

If implementations are changed, the relations are simpler than in the case of requirements. It needs to be ensured that there is no impact on the linked requirements. In case of performance optimization in the implementation it can be expected that the requirements are still met. Nevertheless, for each requirement connected a proof is

[1]If no requirements remain for a component the component needs to be deleted. By this operation additional verification need to be performed.

needed, that is: $\forall_{r\in\overrightarrow{S}(\overrightarrow{I}(i))} : V_s(r,i)$. Furthermore, the compliance of the interface of the implementation to the interface of the components needs to be analyzed again $(V_i(\overrightarrow{I}(i), i))$.

Modifications to components result only in interface checks. Nevertheless, the change might propagate further to requirements and implementations as well. The interface checks whether the implementation $V_i(c, \overleftarrow{I}(c))$ and all connected requirements $\forall_{r\in\overrightarrow{S}(\overleftarrow{I}(c))} V_i(r,c)$ need to be re-evaluated.

Requirements added to atomic components need to be checked against the implementation $(V_s(r, \overleftarrow{I}(\overleftarrow{S}(r))))$ and the interface of the component $(V_i(r, \overleftarrow{S}(r)))$. Also the refinement relation to the higher-level requirement needs to be evaluated: $V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$. If the requirement is added higher in the hierarchy of the system, additional requirements need to be created that refine the added requirement until the implementation level is reached. Hence, all interface, refine and satisfy analyses need to be executed for all the requirements.

Components added at the lowest level of refinement requires the current implementation to be removed, and an implementation needs to be added on the new component. Hence, also a requirement needs to be created for the newly added component. The verification activities linked to all of these elements need to be re-executed. While adding an intermediate component there is no need to remove an implementation, but the set of verification activities is identical to the case of an atomic component.

Deleting a requirement will also force the removal of all subrequirements. Nevertheless, only the refinement relation that had the changed requirement in its set of refining requirements needs to be re-verified: $V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r)))$. Since removing a requirement from a component will not affect any satisfy or interface relations, no further analyses need to be performed. It needs to be noted that components that do not have associated requirements need to be deleted. We do not include this deletion of components in the change operation of the requirement. These deletions start a new operation, that requires the re-validation of further system components.

There are two delete operations for components defined. Either there exist no sibling components on the same level, in which case an implementation needs to be created, or there exists at least one sibling, in which case it is not necessary to create any additional implementations. Therefore, since the requirements of the components are deleted, the refinement relation needs to be re-verified $(V_r(\overrightarrow{R}(r), \overleftarrow{R}(\overrightarrow{R}(r))))$ as does the satisfaction relation between the newly introduced implementation and its requirements for components with no siblings $(V_s(\overrightarrow{R}(r), i_d))$.

As the next step in the process, for all failed verification activities, further actions are required. If all performed tests have been run successfully, a new consistent state of the system has been reached. We show this property later in this section. For each failed verification activity, at least one modification in the set of compensation candidates is necessary. The potential candidates for additional changes directly follow from the involved elements in the individual verification activities. A comprehensive list of the compensation candidates is depicted in Table 3.8. It is possible that a single change

Verification Activity	**Compensation Candidates**
$V_r(r, R)$	modify r or $p \in R$
	add requirement(s) to R
	delete requirement(s) from R
$V_i(r, c)$	modify r or c
$V_i(c, i)$	modify c or i
$V_s(r, c)$	modify r or c

Table 3.8 – Which elements are considered as compensation candidates

resolves multiple failed verification activities.

The modification on the system, again, needs to be evaluated. This cycle of verification activities and modification is repeated, until all verification activities are successful.

To show that this approach contains a change in the identified region, and there is no change effect outside the identified system elements, we need to show that:

- There is no impact on any system element on abstraction levels higher than the last changed, but successful refinement activity.
- There is no impact on lower abstraction levels in the system if the refinement activity on requirements has a positive outcome.
- Implementation and interface checks are already contained in the requirements refinement structure.

Requirements build the backbone for change propagation in the presented impact analysis approach. Figure 3.17 indicates the possible paths of change propagation in a system. The connections refer directly to the verification activities that are executed after changed as described in Table 3.7. It needs to be noted that the part relations between components are not used for change propagation. Instead, if the interface of components c is changed, first the relation to the requirements $R = \overrightarrow{S}(c)$ is checked. If there are changes in the requirements needed, also changes in the set of refining requirements $R_r = \bigcup_{r \in \overrightarrow{S}(c)} \overleftarrow{R}(r)$ are needed. These will then influence the components $\overleftarrow{P}(c)$.

Following this observation it needs to be shown that the refinement relations of requirements are able to contain the changes: If the refinement relation that includes a changed requirement r' in the set of subrequirements $(V_r(\overrightarrow{R}(r'), \overleftarrow{R}(\overrightarrow{R}(r'))))$ delivers a positive test result the traces described by the parallel composition of the new subrequirements $[\![\bigotimes_{\overleftarrow{R}(\overrightarrow{R}(r'))}]\!]$ is a subset of the set of traces of requirement $[\![r']\!]$, as by the definition of refinement. Hence, both the new set and the old set of refined requirements correctly refine the higher-level requirement. As a consequence, all refine relations that make use of the higher-level requirement r are not affected by the change. The same principle applies for the refinement relation that uses the changed requirement as a top-level

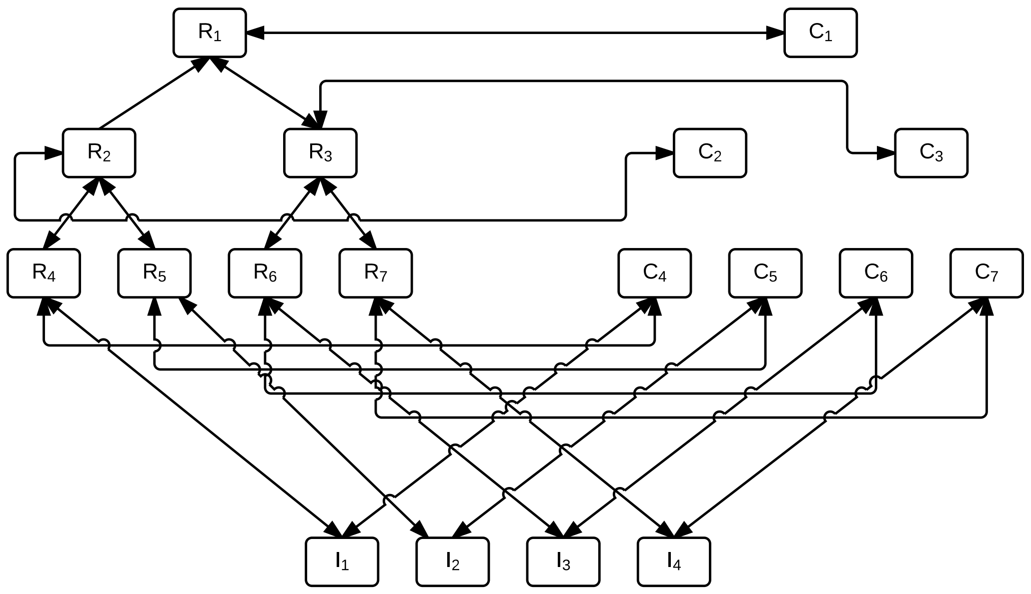


Figure 3.17 – Illustration of the possible change propagation paths, according to the used verification activities V_s, V_i, V_r

requirement. If this relation is still correct, the set of refining requirements is still a subset of the changed requirement $V_r(r', \overleftarrow{R}(r')) = 1 \Leftrightarrow [\![\bigotimes_{\overleftarrow{R}(r')}]\!] \subseteq [\![r']\!]$. Hence, the refining requirements do not need to be modified and all lower refinement relations are still valid.

Hence, we have shown that a semantic impact analysis is possible using the systems requirements. It is determined on the behavior of system elements if a change propagates to a different component, we can directly exclude elements from the set of possibly impacted elements even if they are connected by a tracelink. The engineers of the system are still in full control over their changes, since they select the component from the set of compensation candidates manually. This is the preferred way, since the experience of the engineers enables them to quickly select good compensation candidates. A time intensive design space exploration would require much time and is potentially even more faulty.

The here proposed change impact process focuses on the needed verification properties in the system. But other tests and or reviews are very likely to be performed to gain certification for the applicable engineering and safety standards. These tests can be connected to elements in the system they are evaluating. If it has been discovered that there is no impact on the elements they are connected to, the result of the additional test/verification activity is not affected. Examples for such activities could be reviews or field tests. Again, the same restriction applies here as for the whole process. The tests are not affected only if all relevant aspects of the system are modeled. Therefore, requirements need to exist that state the expected results of the test.

3.3.5 Supporting the Compensation Candidate Selection

In contrast to impact analysis approaches that fully automatically implement a change in the system, for example, by predefined rules (Lehnert, Farooq, & Riebisch, 2013), our experience with engineers conducting impact analysis indicates that approaches are favored in which the control of the applied changes is still in the hands of the engineer. Also Ambler (Ambler, 2002) and Humphrey (Humphrey, 2000) indicate that experienced developers are able to give good estimates of how a change needs to be implemented. Still, in addition to the identification of affected V&V activities, a guidance in the impact process is appreciated by the developers (de la Vara, Borg, Wnuk, & Moonen, 2014). Hence, we provide an approach that helps the developers in selecting suitable compensation candidates. Nevertheless, this is not possible and needed in every case, but there exist situations, in which a change can be implemented in a cost-efficient way which is hard to recognize by the developer. These cases are best suited for automated guidance.

The system situation in which we want to support the engineer is characterized by the availability of slack. Slack is defined by Nicholson et al. (2000) as the budget of time or space, that is not yet used by any system component. These margins exist often, either intentionally or unintentionally in systems. Nicholson mentions the intentionally introduced slack, which is used to improve the maintainability of a device for later upgrades. Unintentionally introduced slack results from uncertainty in the development process with respect to the later used hardware or software solution. That is, that at the beginning of the development process it might not be obvious how much resources the realization of a function might consume. Hence, the resource consumption needs to be conservatively over-approximated. If it turns out later, that a solution needs less resources, the slack is present in the refinement structure of the requirements. This slack is used to compensate changes. Eckert et al. (2004) is calling these elements absorbers in the design space.

Intentionally introduced slack (see Figure 3.18 `ECU1`) is typically larger, than unintentionally introduced slack, as seen on `ECU2`. This results from competitive markets, in which slack is an avoidable cost. Furthermore, unintentionally introduced slack is not present only on the lowest levels of decomposition in the systems architecture. It is possible that slack is available on higher levels of refinement/decomposition. This results from requirements that do not use all the available resources since it is already known that they are not necessary. This situation can occur if complete branches of requirements and implementations are transferred between projects.

In the automotive domain, slack in not introduced intentionally, hence, the buffers available for changes are extremely small. Nevertheless, even in this scenario we are able to *shift slack* to compensate changes. With respect to the scenario in Figure 3.18, a change in `Task1` could lead to a violation of the resources assigned to `ECU1`. In case of the existing impact techniques, this would require changes in the implementation of `Task2` to meet the resource limit of `Task1`. Still, if the limit of ECU1 is only slightly exceeded, we could increase also the resource limit of `ECU1`, by reducing the resource limit of `ECU2`. The limit of `ECU2` can be reduced to the sum of resources consumed by `Task3` and `Task4`.

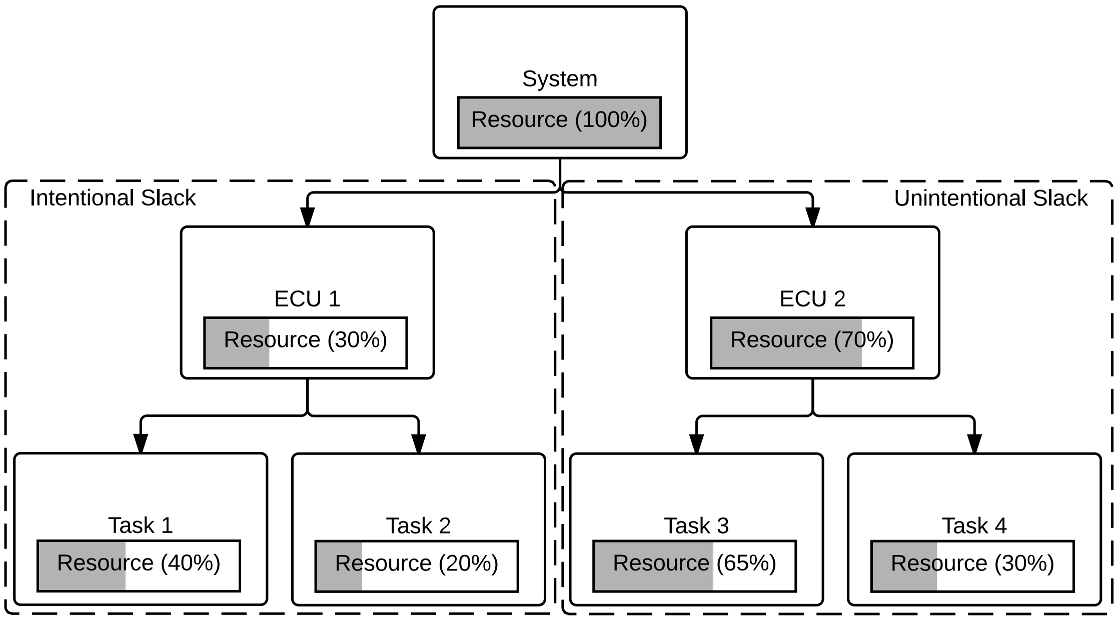


Figure 3.18 – Slack introduced intentionally (ECU1) and unintentionally (ECU2) as a resource. The values indicate the percentage used of the available ressources one level above.

Hence, a change in `Task1` can be compensated by a chain of resource shifts, without the need to change a single further implementation. In case of a complex system, it is nearly impossible even for a skilled developer to discover this compensation possibility, hence an automated technique to identify the compensation strategy is needed.

In terms of trace semantics and contracts the slack between requirement r_{top} and a set of subrequirements R is given by:

$$\Delta T = [\![r_{top}]\!] \setminus [\![\bigotimes_{r \in R}]\!]$$

If the refinement relation between r_{top} and $\bigotimes_{r\in R}$ is given, r_{top} can be safely replaced by $\bigotimes_{r\in R}$ since the set of traces of r_{top} is only reduced, and no additional traces outside the set of $[\![r_{top}]\!]$ are added. That is, none of the already verified properties is tainted. We describe the shifting process using two algorithms (see Algorithm 1 and Algorithm 2). We define the *drain* function that replaces a requirement with the composition of its subrequirements recursively in Algorithm 1. The worst-case running time of the `DRAIN` function depends on the number of requirements $|R|$ in the system and the running time of the parallel composition $T(\otimes)$ which might differ between different representations of requirements. Hence, an asymptotic upper bound of $\mathcal{O}(|R|) \cdot T(\otimes)$ exists. The algorithm obviously terminates.

Using the drain function we can try to prevent a change in a requirement r_c to propagate towards the implementation, avoiding costly changes on hard- or software.

Algorithm 1 recursive procedure to retrieve a set of traces of a subtree of the requirements break-down structure without slack

```
Require: $r$
 1: procedure DRAIN($r$)
 2:     if $\overleftarrow{R}(r) = \emptyset$ then
 3:         return $r$
 4:     else
 5:         $R = \overleftarrow{R}(r)$
 6:         $R_{new} = \emptyset$
 7:         for all $q \in R$ do
 8:             $R_{new} = R_{new} \cup$ DRAIN($q$)
 9:         end for
10:         return $\otimes_{r \in R_{new}}$
11:     end if
12: end procedure
```

Hence, the starting point is a failed verification activity $V_r(r_{top}, R)$, with $r_c \in R$, with r_c denoting the changed requirement. In terms of Figure 3.18 this could result from a task, that cannot meet its expected resource constraints, and therefore the corresponding requirement stating the guaranteed resources is changed. The algorithm to check if it is possible to compensate the change by relaxing the resource constraints higher in the requirements break-down structure is depicted in Algorithm 2. The algorithm proceeds iteratively to increase the included part of the system in the analysis. Hence, first only one refinement level above r_{top} will be considered, then two levels, and so on, until the top-level requirement of the system is reached. The check returns if such a compensation by draining slack is possible and indicates to which level changes in the requirements would be necessary. Based on this information the developer can decide if he wants to adapt the resource usage in the requirements, which is potentially much cheaper, or wants to modify implementations.

In the worst-case the Algorithm 2 needs to iterate over all levels of the requirements structure without finding a successful refinement. Hence, the `DRAIN` function is called in a magnitude of $|R|$ resulting in an overall asymptotic upper bound of $\mathcal{O}(|R|^2) * T(\otimes)$. The running time can be reduced to $\mathcal{O}(|R|) * T(\otimes)$ by caching the results of the `DRAIN` function, instead of calculating them in each level of iteration again. This has been skipped in favor of a better readability. The algorithm obviously terminates.

3.4 Requirements on a Modular Safety View

The process and algorithms in this chapter are based on an abstract system representation using trace semantics and contracts. Hence, to be able to use the technique on a real system, we need to provide a specification language, that is able to represent functional safety concepts and is still suited for the presented change impact analysis approach.

Algorithm 2 procedure to iteratively test the requirements of a system if compensation is possible by draining slack

```
Require: $r_c \in R, V_r(r_top, R) = 0$
procedure SlackAvailable($r_{top}, R, r_c$)
    ℹ Initializing variables
    level = 1
    start = $r_c$
    ℹ extend scope of requirements towards top level
    while $\overrightarrow{R}(start) \neq \emptyset$ do
        start = $\overrightarrow{R}$(start)
        ℹ now, perform draining at new start node
        $R_l = \overleftarrow{R}(start)$
        $R_{new} = \{\}$
        for all $r \in R_l$ do
            ℹ $R_{new}$ is the set of all drained requirements
            $R_{new} \cup$ drain($r$)
        end for
        ℹ Check if refinement is given for drained requirements
        if $V_r(start, R_{new}) = 1$ then
            return (TRUE, level)
        else
            level++
        end if
    end while
    ℹ Top level requirement has been reached, but refinement not given
    return (FALSE, level)
end procedure
```

Therefore, the following requirements on a system safety model exist:

- Requirements shall be represented as contracts
- The refinement property of the requirements shall be automatically analyzable.
- The compliance of the requirements to implementations shall be automatically analyzable.

Based on these requirements a formal specification language is developed in chapter 4.

3.5 Conclusion

Changes propagate in a system from one component to related components. These relations are typically expressed by tracelinks for HW/SW systems or also by call relations and variable usage for pure software systems. Different targets exist for the use of impact analyses ranging from an initial estimate of costs to the reduction of faults introduced in changed systems by giving hints which other elements are most likely to be affected. We want to use a change impact analysis in the context of safety critical embedded systems to reduce the re-certification effort. In fact, a linear relation between the size of the changes performed to the system and the needed verification activities shall be established, eliminating the need to re-certify the whole system. Our investigation of the existing techniques has shown that none of them is suited to be applicable for functional safety concepts of safety critical automotive systems. The gap analysis has revealed four requirements on the impact analysis:

- *The new impact analysis shall use the semantical correctness of the system as the target for impact detection.*
  A change impact analysis has been presented that is capable of identifying the affected elements within a system on a semantic base (see Section 3.3). It is unique to our approach to reason on the actual required behavior of a component, expressed as contracts, instead of the syntax of source code or predefined tracelinks. Since contracts provide separation between an assumption, describing the context, and a guarantee, describing the provided behavior of a component, the refinement relation indicates if a changed context is still suitable for a component. This ensures, that there are no further changes necessary in a part of the system if the change is contained by successful refinement checks.

- *The new impact analysis shall provide a stop criterion for change propagation in the system.*
  The impact analysis process (see Section 3.3.4) determines affected system elements using the refinement, satisfaction and interface relations. Although a changed component indicates, that all directly connected elements are potentially affected, all relations can be assessed using the semantics given by the specified contracts. Therefore, the change propagation can be stopped.

- *The new impact analysis shall be able to contain changes with very little slack available in the system.*
  No overestimation in resources or behavior is required by the process (see Section 3.3.5) as a barrier for change propagation. The system can be arbitrarily modified. However, if there is slack available in the system specification, the engineer performing the analysis can be notified that cost efficient modification alternatives exist. In particular, we offer a set of requirements to be changed instead of an implementation. Since automotive devices are not intended to be modified after production, the slack is typically very minimal between directly refined requirements and, hence, a combination of slack in different parts of the system is necessary. To compensate a failed refinement analysis, the scope of the identification of slack is consecutively extended until finally the total system is analyzed to guarantee the smallest possible chain of changes.

- *The new impact analysis shall be able to detect impacts on logical architectures.* In this chapter we did not introduced any restrictions on the language used to express the contracts assertions but relied on abstract trace semantics.

The concrete language, meeting the requirements stated in section 3.4 to perform an impact analysis on functional safety concepts, is presented in the following chapter. Furthermore, the claim that the verification effort scales linearly with the size of the change is evaluated in chapter 5.

CHAPTER 4

Development of a Compositional Safety View

Safety critical systems are characterized by a high amount of required verification and validation activities necessary for qualification or certification. Changes in the system often cause a tremendous re-verification effort. Fenn et al. (2007) described the experience of industrial partners in which the costs for re-certification are related to the size of the system and not to the size of the change. Also Espinoza et al. (2011) discovered that the monolithic and process oriented structure of the safety cases required by nearly all domain-specific safety standards may require an entire re-certification of the system after changes. This stems from the inability of the impact analysis techniques to encapsulate propagating changes.

To overcome the re-certification problem a contract-based change impact process has been developed in chapter 3. This process is based on an abstract contract-based specification. Since the impact analysis shall be applied for automotive functional safety concepts a concrete specification language needs to be developed. The specification language shall describe the fault propagation and mitigation properties of a component. This includes degradation of the system as well as fault tolerance mechanisms.

Existing approaches to specifying safety properties in a modular way (see Section 4.1 for an overview) lack either expressiveness or compositionality, limiting the ability to contain changes. Compositionality is defined by Hungar (2011b) as well as Peng and Tahar (1998) as: “there exists a separation between the specification of a component and its actual behavior. If a component is replaced by another component meeting its original specification, the correct functional behavior of the composed system is maintained.”

It is a design goal of our approach to change existing industrial design processes as little as possible. We annotate existing design models with a formal, textual specification that can be stored in typical requirements engineering tools like IBM Rational DOORS. Therefore, no changes in the existing design process are necessary. The safety specification can be used directly as a requirement or a property in the development process. The distinction between requirement and property is process oriented. The presented formalization approach remains identical.

4.1 Related Work

Modular safety cases are highly desired in systems engineering. Hence, many different modeling approaches exist to represent safety properties and systems. In this work we focus on approaches that can be used in combination with typical engineering models like SIMULINK, AUTOSAR or SysML. Therefore, we do not investigate approaches that require special analysis models like “function-structure-models” (Echtle, 1990), which are graph structures with components as nodes and functions as edges, or models based on Lamport’s process and channel notation (Lamport & Merz, 1994). In addition, we focus on approaches that claim to be re-useable and modular, that is, that components can be put in a different context and can be characterized by their interface.

The safety modeling technique presented in this thesis falls into the category of *failure logic modeling* approaches. We briefly introduce the topic and present the limitations of three popular specification notations to demonstrate the need for a contract-based approach. Furthermore, graph based structuring approaches for safety cases have gained popularity and are introduced in section 4.1.2. Since the goals are very similar to what we want to achieve we describe two common approaches for *safety case structuring* and relate them to the contribution in this thesis. Finally, we detail the already existing work on safety contracts that has been published and that we build upon.

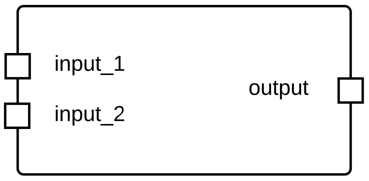
4.1.1 Failure Logic Modeling

Failure Logic Modeling (FLM) approaches (Lisagor, 2010; Lisagor, McDermid, & Pumfrey, 2006) describe the failure behavior on the output of a component in terms of input and internal faults. Fault trees or FMEA (Failure Mode and Effect) tables can then be derived from such specifications.

Hip-Hops

Hip-Hops (Pasquini, Papadopoulos, & McDermid, 1999) is a failure logic modeling-based approach that aims at automating the generation of safety analysis artifacts such as Fault Trees and FMEA tables. To this end, design models, for example, SIMULINK models (Papadopoulos & Maruhn, 2001), are annotated with a fault propagation specification that states for a component possible input and internal faults as well as how they influence the output. In Figure 4.1 an example component is depicted together with its local safety analysis in the form of a table. The local analysis results are the possible failure modes that might occur at the output of the component. These output deviations can be caused by internal malfunctions of the component or by incorrect inputs. It is also possible to state the probability of occurrence (failure rate) of the internal malfunctions.

The failure rates can be used along the structure of the generated fault tree to calculate the overall probability of a failure at top level.



Output Failure Mode	**Description**	**Input Deviation Logic**	**Component Malfunction Logic**	$\lambda(f/h)$
Omission-output	The component fails to generate the output	Omission-input_1 AND Omission-input_2	Jammed OR Short_circuited	$5 \cdot 10^{-7}$, $6 \cdot 10^{-6}$
Wrong-output	the component generates wrong output	Wrong-input_1 OR Wrong-input_2	Biased	$5 \cdot 10^{-8}$
Early-output	Early output	...	...	...

Figure 4.1 – Example component and failure logic table. Source: Papadopoulos and Maruhn (2001)

The deviations from the nominal behavior on the output considered by Hip-Hops follow HAZOP (Gould, Glossop, & Ioannides, 2000; McDermid & Pumfrey, 1994) guidelines and are structured in three categories. The first category is *service provision failure*, such as no service at request (omission) or unintended service (commission). The second category summarized failures related to the value domain (e.g., a value being out of range, stuck, biased or simply wrong. Timing failures, such as a too late or too early occurrence of an event, belong to the third category.

These safety analyses are performed for all components in the system hierarchy (see Figure 4.2). Therefore consistency checks can be performed if top level components analysis results fit to their implementation. Furthermore, fault trees can be generated for the functional failures on top level components. Together with the failure rates of the atomic faults it is possible to determine the probability of a system function failure. This fault tree generation can be performed iteratively during the development process to improve the systems design.

The basic approach has been extended by introducing negations (Sharvia & Papadopoulos, 2008) or temporal properties (Walker & Papadopoulos, 2008, 2009).

Hip-Hops is aiming for providing a hierarchical safety analysis approach. Still, our

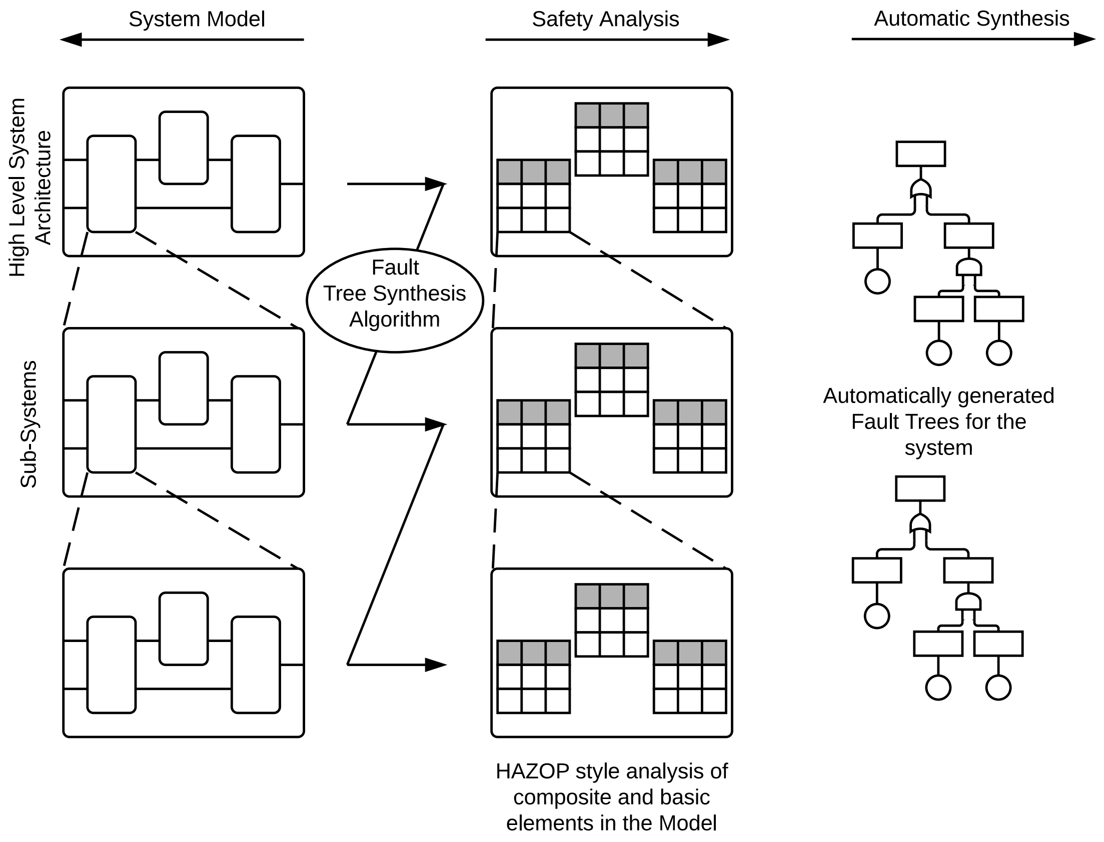

System Model
Safety Analysis
Automatic Synthesis
High Level System Architecture
Sub-Systems
Fault Tree Synthesis Algorithm
Automatically generated Fault Trees for the system
HAZOP style analysis of composite and basic elements in the Model

Figure 4.2 – Relation of hierarchical models and HAZOP analyses with generated fault trees. Source: Papadopoulos and Maruhn (2001)

approach is more requirement oriented. We directly formalize safety requirements and address the correctness of their refinement by using abstraction techniques. Hence, we are able to state top-level requirements like the absence of single point faults, and later refine the implementation, without compromising the validity of the previously gained results. This means, we fully support a top down oriented design approach, without the need to already know the atomic faults of the implementation components. Although the stated fault propagation relationships are similar, there is no need to manually interpret the generated fault trees while using safety contracts.

Fault Propagation and Transformation Calculus

The fault propagation and transformation calculus (FPTC) (Wallace, 2005) addresses the problem of a modular safety specification. The architectural design is expressed using Real-Time Networks (RTN) (Paynter, Armstrong, & Haveman, 2000) to model the communication channels. In an RTN the communication protocol can be specified in terms of destructive or nondestructive read and write operations. The calculus itself is based on direct propagation notations using failure modes in accordance to HAZOP guidewords. In Figure 4.3 the four basic types of propagations are depicted, using the asterisk sign (*) as a symbol for correct value. A component in an RTN is considered a *source* for a failure if a correct value is passed to the component and an internal fault causes the result to be wrong. In this example the deviation is a timing violation, late. Similarly, a component is called a *sink* if it can correct wrong input values. Most components that do not implement any special safety mechanisms will *propagate* a failure, that is, the deviation on the input will also be visible at the output. Many such rules can be defined for a component. Instead of stating all necessary rules, a default behavior is assumed, which is propagation. A *transformation* of failure occurs, if the failure at the input is different from the failure at the output, caused by the wrong input.

$*$	$\rightarrow$	late	$(source)$
early	$\rightarrow$	*	$(sink)$
ommision	$\rightarrow$	ommision	$(propagation)$
late	$\rightarrow$	value	$(transformation)$

Figure 4.3 – Basic fault transformation rules in FPTC. Source: Wallace (2005)

The examples in Figure 4.3 apply only for components with one input and one output port. For multiple ports a more complex notation needs to be chosen, as indicated in Figure 4.4. If multiple output connections are used the outputs can be referenced using tuples, where each position in the tuple relates to a defined connection. The tuple approach is also applied to inputs. Nevertheless, the number of potential combinations grows exponentially with the number of input connections. Therefore, two additional notations are introduced. The underscore symbol (_) indicates a "don't care." Hence, the second rule in Figure 4.3 indicates that as soon as there is a failure of type late on the first input port, the outputs will be affected by a value and also a timing failure, regardless of the status of the second input port. The second mechanism reducing the number of input combinations is the usage of *functions*. These functions have a similar meaning as the underscore, but bind a value to them, to be used on the right-hand side of the rule. Hence, the third rule defines a propagation of failures, where a late failure on the first input propagates to a late failure on the second output, and all possible other input failures on the second port will propagate identically to the first output port.

The approach provides a fixed-point algorithm to calculate the set of all possible occurring input and output faults for every component in the system. The algorithm starts with an in-set and an out-set for each component that includes only the normal behavior symbol, then the rules are applied until a fixed-point is reached. This algorithm is used to detect if changes in the architecture have an impact on the safety of the device. If the set of possible failures deviates, there is an impact from the change.

The FPTC approach tries to solve the same problem that is addressed by this thesis, namely the reduction of costs caused by changes in safety critical systems. Nevertheless,

late	→	(value,*,late)	(*multiple output*)
(late,_)	→	(value,late)	(*multiple in- and output*)
(late,*f*)	→	(*f*,late)	(*usage of variable*)

Figure 4.4 – Transformation rules in FPTC for multiple ports. Source: Wallace (2005)

the approach still has some flaws. In contrast to the solution presented in this thesis, the FPTC approach does not provide any support for multiple abstraction levels, i.e., specifying a component, then designing the subcomponents and proving the correctness of the refinement. Additionally, within FPTC it is not possible to state temporal properties in the propagation of faults (e.g., that a fault is going to be detected after some amount of time). Furthermore, in case of multiple input and output ports, the notation of *positive propagation* is more complicated than the *negative fault containment* properties stated within safety contracts. Also, the order of the evaluation of the FPTC requirements matters, which is not the case for the invariant safety patterns. The readability is further impacted by overlapping rules, where more concrete rules overwrite more general rules. The determination of the "concreteness" of a rule might itself be a difficult task. Furthermore, the used fixed-point algorithm calculates the set of possible failures, but the causality of the resulting failures is lost. Therefore the set of failures might be identical, but since the same failures might be caused by multiple inputs, the failure might occur much more frequently and the system is less safe, but according to the approach there is no change detected. A minor drawback of FPTC is also the necessity of an RTN model, while safety contracts can be attached to any port-based component model.

Compositional Fault Trees

Mäckel and Rothfelder (2001) identified two concrete flaws in classical fault trees. First, the difficult handling of events, that are used multiple times on different nodes, but are semantically identical. These *repeated events* are tagged in existing fault trees but still separate entities in the tree. Second, the limitation of fault trees to a single top level event. This limits the ability to represent relations between different top level failure modes. Hence, an extension of fault trees is proposed, called *cause effect graphs* (CEGs). Repeated events are represented as a single node having multiple edges. The readability is furthermore improved since CEGs can now also be read in a bottom-up fashion. In addition, multiple top level events can be represented that use a shared set of intermediate and basic events. That way faults can be identified that can lead to multiple top level events, which is a good starting point for architecture improvements.

Kaiser, Liggesmeyer, and Mäckel (2003) address the problem that the typical structure of a fault tree is related to the hierarchy of failures, but not to the hierarchy of the system. It was their goal to extend CEGs to be stated for architectural components that can be composed, aligning fault tree and architectural elements. To achieve this they

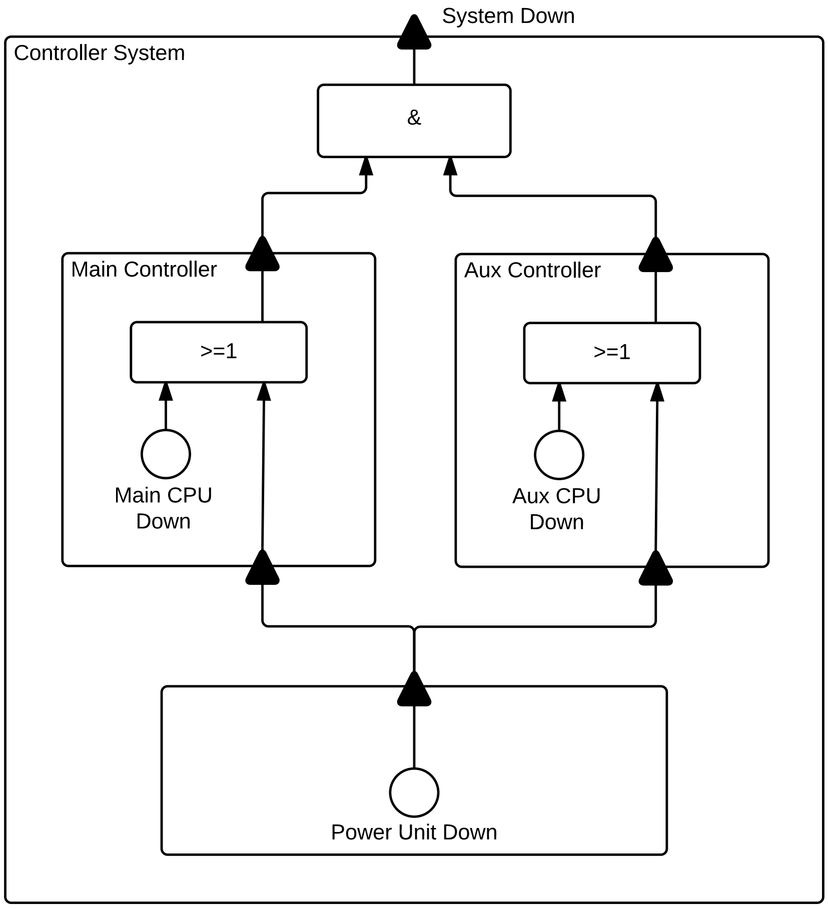


Figure 4.5 – Multiple componenet fault trees combined to a system

added directed ports to the fault tree, representing possible input- or output-failures of a component (see Figure 4.5). Hence, fault tree parts represent the failure propagation behavior of individual system components. As an additional benefit, the component fault trees are re-usable, if a component is instantiated multiple times in the architecture. Proposals exist to integrate compositional fault trees in UML (Adler et al., 2011).

With their extension of CEGs to compositional fault trees Kaiser et al. (2003) defined a failure logic modeling language, that resolves the architecture alignment problem of fault trees. Still, using FT-syntax the expressiveness of the failure propagation is limited to logical boolean operators and prohibits the use of timing information. Furthermore, the assumption, that component internal faults are independent from the rest of the system is in some cases to restrictive. If a component is instantiated twice, a systematic fault in the component is a common cause fault for both instances, but handled independently by default. The quantitative analysis is restricted to systems that do not rely on any inputs, which is uncommon for automotive systems. However, it seems that this restriction could be bypassed if the input failures are represented as internal faults of the first subcomponent processing them, or introducing a new input subcomponent for this fault. Compositionality and re-usability is addressed in the approach, but means for abstraction are still missing. That is, at the time of the analysis the model still needs to be complete and cannot be refined later without the need of a full re-evaluation.

4.1.2 Safety Case Structuring

A safety case is defined as "a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment" (Bishop & Bloomfield, 1997). Most safety standards require a safety case to be delivered to the assessing authorities to evaluate the developed device. Hence, many approaches have been developed lately to structure the safety case in a readable and easy to understand way. In this chapter of the thesis a specification for safety properties shall be developed that is easy to apply and allows reasoning about the fulfillment of safety requirements. Looking at these shared goals, we describe two popular approaches for safety case structuring and relate them to the contribution of this thesis.

Goal Structure Notation

The *Goal Structure Notation* (GSN) is a graphical notation to document arguments in terms of, for example, claims, evidences and contextual information. Although GSN can be applied to various argumentations for systems, services or organizations (GSN Community Standard, 2011), it is most frequently applied to represent safety cases for safety critical systems (Kelly & Weaver, 2004). The approach in its current form has been developed by Kelly in 1999 (Kelly, 1999) and is currently being adopted to a community standard (GSN Community Standard, 2011) published by Origin Consulting in York. We refer here to the notation of the community standard.

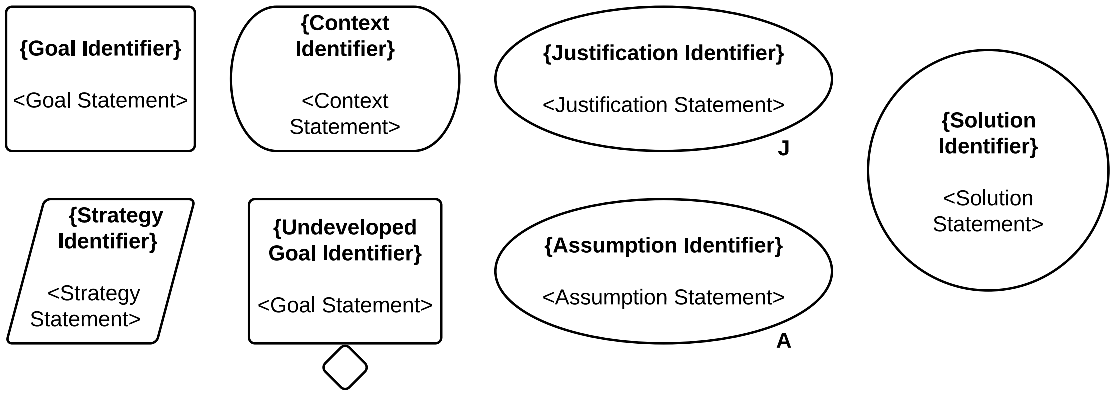


Figure 4.6 – Principal element of the goal structure notation according to GSN Community Standard (2011)

The main elements are represented in Figure 4.6. The *goal* represents the claim, which is a part of the argument. Typical goals are the absence of catastrophic single point

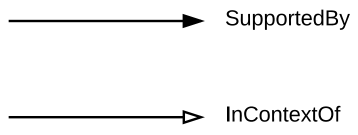


Figure 4.7 – Links in the goal structure notation according to GSN Community Standard (2011)

faults or that the system is reasonably safe to operate in a defined environment. Goals can be decomposed. The top level goal is fulfilled if all its subgoals are fulfilled. If more complex relations between the goals and the corresponding subgoals exist, *strategies* can be used to describe the inference mechanism in more detail. Such strategy could be the omission of all identified failures, which are listed as subgoals. Finally the decomposition of goals should be terminated by a *solution* that represents the reference to evidence items. Such evidence can be test or analysis results, as well as performed reviews. If there is no solution available yet for a goal or subgoal, this branch is terminated by an *undeveloped goal*, indicating that a refinement or a solution is still in development to finish the argument. This undeveloped symbol can be applied with identical semantics to strategies. Goals, strategies and solutions can be valid only in a particular *context*, that is, one which describes, for example, a specific safety standard. As another example, a strategy that argues over all hazards is valid in the context of a particular hazard and risk analysis. A *justifaction* is a statement of rationale for choosing a strategy or the presentation of a goal. If statements are used that are intentionally unproved, an *assumption* can be connected to a goal or strategy.

GSN provides two types of links between elements, *SupportedBy* and *InContextOf* (see Figure 4.7). The SupportedBy relation is used to indicate inferential or evidential relationship, and is therefore used to point from top-level goals to subgoals or strategies. Also Solutions are connected to goals via this link. The InContextOf link indicates the use of elements that limit the scope of an argument, for example, the use of contexts, assumptions and justifications. Therefore, these elements are connected to goals and strategies via this relation.

In addition to the graphical notation a method has been developed describing how to use the notation in a documentation scenario (Kelly, 1997). The method consists of six steps:

- **Step 1** - Identify goals to be supported
- **Step 2** - Define basis on which goals stated
- **Step 3** - Identify strategy to support goals
- **Step 4** - Define basis on which strategy stated
- **Step 5** - Elaborate strategy (and proceed to identify new goals – back to Step 1)

  OR

- **Step 6** - Identify basic solution

This recursive process can be followed to create a safety documentation in a top-down manner.

GSN is a notation to structure the information in the safety case. It is the main idea to represent goals and their breakdown structure until a refinement level is reached, where the subgoals can be proven by a given solution. This idea of decompositionality is very similar to the principles behind contracts. A specification is broken down into finer parts, and the refined specifications are checked against the implementations. Nevertheless, GSN does not provide any support for analyzing or formalizing statements. Still, results from contract analyses can be used as subgoals or strategies in a GSN argument structure and can therefore contribute to the top-level goal of providing a reasonable safe system.

SHIP Safety Case

SHIP was a EU project with the objective to "assure plant safety in the presence of design faults." To argue on the safety of the system a structure for safety cases is proposed (Bishop & Bloomfield, 1997). This hierarchical structure is represented in Figure 4.8 consisting of four main elements. A *claim* is a systems property that may directly or indirectly state safe operation. *Evidences* are used as the base for a safety argumentation, they typically represent special qualities of the development process or actions to deal with erroneous situations. This kind of evidence is called a *fact* (see Figure 4.8). Other types of evidence are *assumptions*, which are defined as conditions, which are necessary for the safety argumentation but not always given, or *subclaims*. The link between the evidence and the claim is called *argument* which uses *inference rules* to "relate" the different evidences.

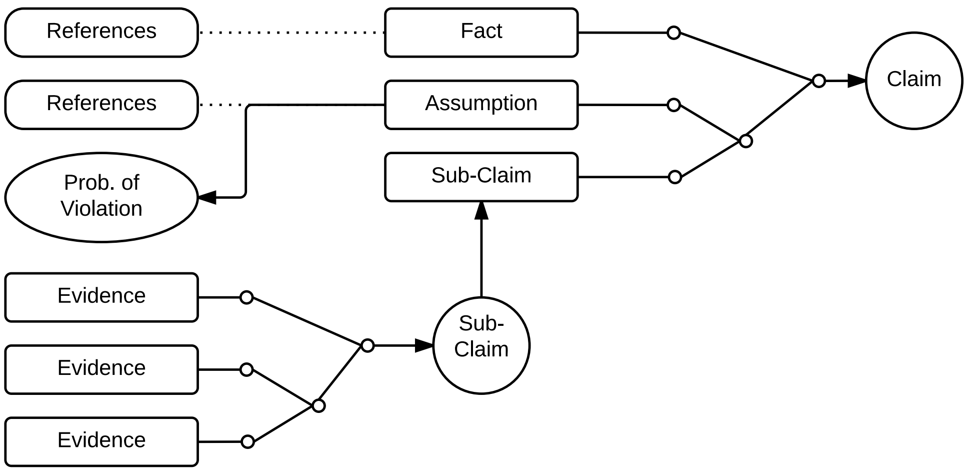


Figure 4.8 – Overview of hierarchical claim structure (Bishop & Bloomfield, 1997)

Depending on the type of the arguments the inference rules look different. For *deterministic arguments* the inference rules might be predicate logic to relate statements.

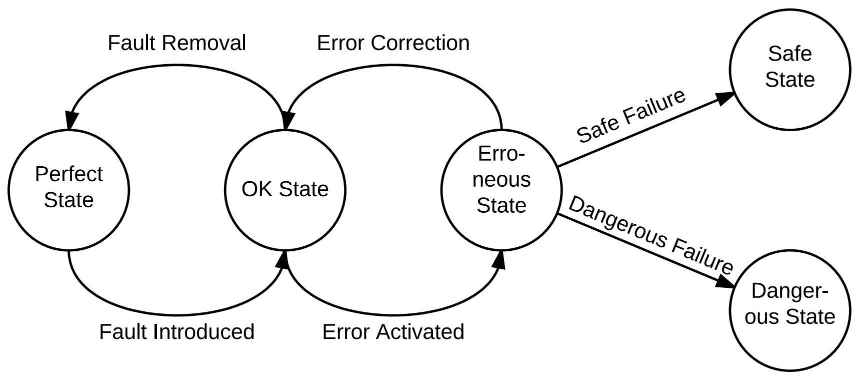


Figure 4.9 – Model of system failure behavior (Bishop & Bloomfield, 1997)

For *probabilistic arguments* the inference structure can be a markov process. There is a third type of argument mentioned, the *qualitative argument*, which, for example, can be the compliance to standards or guidelines.

To structure the arguments, a simple transition system is presented, that relates five different behavioral states of a system (see Figure 4.9). If the system is running under nominal conditions without faults, the *perfect state* is active. As soon as a fault occurs the system switches to the *OK state*, meaning that the fault is present but is not yet triggered. If an input is sent to the system that uses the faulty elements, the system is in the *erroneous state*. Whether there is immediately a dangerous situation depends upon the error. It is also possible to detect the failure and go to a *safe state* or correct the error and go back to the OK state. Arguments can reference transitions of this model, to argue why a special evidence helps to build a safe system.

Bishop and Bloomfield (1997) provide a table with possible evidences that help avoid or explicitly take transitions in the model. These evidences have three main sources, the *development process*, the *system design* or *field experience*. The arguments are proposed to be represented in a tabular form to state the causes (faults) and safeguards for the different transitions.

The SHIP approach has been taken up and extended in the Adelard (1998). The manual proposes a process of how to develop a valid safety case and provides checklists and examples for various stages in the process.

The presented approach can be extended with formal specifications to describe evidences and claims and, for example, trace semantics as inference rules. The SHIP approach does not provide any analysis capabilities itself, therefore the contributions of this thesis are on a much lower abstraction level, describing the fault propagation together with the possibility of automatic analyses, that can be used as an argument in a SHIP structure.

4.1.3 Previous work on safety contracts

Contract-based specification approaches have recently gained a lot of attention. Hence, the term safety contracts or formalized safety requirements is used differently by many

Ex. 1	`"if the braking command signal is not provided within 9 ms from the receipt of the pedal signals, then activate emergency brake within 1 ms"` (Sljivo, Jaradat, Bate, & Graydon, 2015)
Ex. 2	`"always (((NewDataAvailable and ValidCRC and 1 <= DeltaCounter and DeltaCounter <= MaxDeltaCounter) and previously in the past (NewDataAvailable and ValidCRC)) implies (then status_ok(Status)));"` (Arts, Dorigatti, & Tonetta, 2014)
Ex. 3	`"If the derivative of actualFuelVolume[%] is less than 0, indicatedFuelVolume[%], shown by the fuel gauge, shall be less than actualFuelVolume[%]; or indicatedLowFuelLevelWarning[Bool] shall be active (true) when the actualFuelVolume[%] is below 10%; or indicatedFuelVolume[%], shown by the fuel gauge, shall show a value below 0%."` (Westman, Nyberg, & Törngren, 2013)

Table 4.1 – Examples for safety requirements in the functional aspect

researchers. We have identified two basic meanings of safety contracts, which can be separated by the aspects (views) introduced in Section 2.2.2. There exist approaches which use contracts for requirements that are associated with the functional aspect according to the design space structure defined by Pohl et al. (2012) as well as Rajan and Wahl (2013). In terms of ISO 26262 such requirements are considered safety requirements. In that sense the term safety contract is not wrong, but we stick to a stricter separation. Others try to specify pure conceptual safety properties as suggested by the various failure logic modeling approaches (see Section 4.1.1).

Three examples for safety requirements in the functional aspect are depicted in Table 4.1. The first example states a relation between functional ports. It requires a dependency between the braking command, the pedal signals and the emergency braking. Clearly, emergency braking is a safety relevant operation, but this requirement states the technical realization of a detection of a fault and a reaction to it. Nevertheless, from this requirement the safety concept can only be guessed, there seem to be faults that cause the braking command not to occur in a defined time frame after the pedal signal and a stoppage seems to be the safe state. In this thesis we propose a strict separation between the safety concept with respect to failures and their propagation and the implementation of the safety concept in terms of detection techniques and reactions. The goal is to support the engineers in achieving a more complete specification. The second example describes also a requirement for safe communication using check sums and message counter. Although being on a technically lower level, this is not considered a safety contract in our understanding, but a functional refinement of a conceptual safety requirement. Although the third requirement still describes functional relationships, it includes an interesting artifact missing in the first two examples. In this requirement the measured values for fuel volume are compared to the actual fuel volume. With this comparison there are failure modes implicitly introduced that would typically indicate that a value is wrong (i.e., different to the real value). Nevertheless, this kind of specification is not totally compositional, since failure propagation is difficult to be expressed. This can be simply explained by the reference to ports way outside the scope of the component, in this case the real environmental value. Hence, Westman, Nyberg, and Törngren (2013) modified the contract approach defined in the SPEEDS project (Enzmann et al., 2008) to deviate from strict interface oriented black box specifications. This deviation is not necessary using the safety contracts presented in this work.

The second identified category of safety contracts are specifying safety properties in a failure logic modeling way. Damm, Josko, and Peikenkamp (2009) proposed that contracts and *Heterogeneous Rich Components* (HRC), the component model developed in SPEEDS, can be used in the context of the ISO 26262 to state safety requirements. The usage of contracts to express failure propagation is mentioned as a possible application scenario, without detailing a concrete notation. Damm et al. (2009) suggest dominance checking (in this thesis called refinement, see Section 2.1) as a suitable verification technique to ensure a correct reaction to faults in the system. Furthermore, different techniques are mentioned to verify the implementation with respect to stated contracts. These techniques encompass test case generation based on contracts as well as fault injection. Nevertheless, none of these techniques is detailed. However, the work of Damm et al. (2009) confirms that further research in this direction, in particular in languages that express safety properties in contracts, is necessary.

Based on these ideas, Böde et al. (2010) presented an approach for hierarchical safety specifications using safety contracts. They introduced patterns to describe the hierarchy of faults, the hierarchy of functions, and the relation between faults and functions, as well as the assignment of a criticality to failures. For example:

- `function <function-name> can be impacted by <failure-list>.`
- `failure <failure-name> is realized by <failure-expr>`
  `[except when <failure-expr>]`
- `failure <failure-name> is <criticality-level>`
  `[during <phase>] [in <mode>] [under <condition>]`

Although the approach is based on contracts, the stated patterns do not fully exploit the contract notation as assumptions are neglected, a major principle of contracts. Furthermore the semantics of the patterns are not defined in a formal way. In addition, a suitable mapping of the approach to an interface of the component is missing (e.g., to ports representing faults).

In 2011 a pattern based requirements specification language (RSL) was presented (Reinkemeier et al., 2011; Baumgart et al., 2011) that encompasses formalization means for functional, architectural, real-time and safety requirements. The safety patterns are influenced by the formalization approach presented by Böde et al., but they include additional patterns, building a set of 12 requirement templates. For example:

- `*Function* shall only fail if *failure_list*`
- `*Failure* shall be detected with probability *p*`
- `*Failure* shall not be caused by *n* independent failures`
- `hazard *Hazard* shall not occur with density higher than *n* per *reference*`

The first pattern is clearly related to previous work (Böde et al., 2010). The introduction of probabilities, independence constraints and hazards remains unique for the RSL. Still, precise semantics of the patterns have not been defined. Also guidance for applying the patterns is missing.

Oertel et al. (2014) presented an approach for expressing safety requirements optimizing and condensing previously gained results. The amount of patterns was significantly reduced from 12 patterns to only four. Furthermore, a linear temporal logic (LTL) based semantic definition was provided (Pnueli, 1977) (see Table 4.2 for overview of presented pattern). The patterns use a cut-set semantic for the expression set. That is, the occurrences of the faults or failures are not timed. Hence, it does not matter if the faults occur at the same time or in a special order, a cut-set defines all the traces in which the defined faults/failures occurred. Table 4.3 highlights how the expression sets can be specified inside a pattern. Each expression set consists of modes and fault/failures or sets of them. The operator *perm* can be applied to individual expressions, stating that some fault, failure or mode is permanently valid. Since the pattern itself negates the expression sets, that is, requires that these combinations of modes and fault/failures do not occur, the perm operator defines, used within the negation, that an expression shall be invalidated at some point in the future. The presented approach, however, does not consider abstraction techniques to refine safety concepts suitable for a top-down design. Also guidance of how to apply the patterns is missing. Nevertheless, in this thesis we build upon the notion and semantics presented by Oertel et al. (2014).

	Pattern 1	**Pattern 2**	**Pattern 3**	**Pattern 4**
Structure	**none of** {expr-set1, expr-set2,...} **occurs**	expr-set **does not occur**.	expr1 **only followed by** expr2	expr1 **only after** expr2
Intuition	The pattern described all traces of a system that do not contain any complete expr-set element.	Derived from pattern 1, where only one expr-set is used. Semantics are identical. Introduced for convenience	The traces of the system are accepted, iff either expr1 holds forever or each expr1 is followed by expr2 or expr1 does not occur.	The evolution of a system is accepted, iff any uninterrupted sequence of expr1 always occurs as a direct follower of expr2.
Example	**none of** $\{\{e_1, e_2\}, \{e_3, e_4\}\}$ **occurs**. Any evolution of the system containing e_1 and e_2 (or containing e_3 and e_4) violates the pattern. Any evolution of the system containing e_1 and e_3, but neither e_2 nor e_4 (or containing e_1 and e_4, but neither e_2 nor e_3, or containing ...), is accepted by the pattern.	see pattern 1	Correct_mode **only followed by** Incorrect_mode. An evolution of the system where Correct_mode always holds, is **accepted** by the pattern. Whenever the Correct_mode switches to be false, then the next state of the system has to be Incorrect_mode.	Detected_Mode **only after** Undetected_mode. A trace of the system where Detected_Mode never occurs is accepted by the pattern. A trace of the system where Undetected_mode always holds is accepted by the pattern. Whenever Detected_mode holds, it means that the direct previous state is Undetected_mode.
LTL expression	$(\mathbf{G}\neg e_1 \vee \mathbf{G}\neg e_2) \wedge (\mathbf{G}\neg e_3 \vee \mathbf{G}\neg e_4)$.	see pattern 1	$\mathbf{G}(e_1 \rightarrow (e_1 \mathbf{W} e_2))$.	$\neg e_1 \wedge \mathbf{G}((\mathbf{X} e_1) \rightarrow (e_1 \vee e_2))$

Table 4.2 – Overview of the four safety patterns as presented in Oertel, Mahdi, Böde, and Rettberg (2014)

Attribute/Operator	Description
Fault/Failure	Is the event of a *internal malfunction* or an *signal-malfunction* occurrence in a certain part of the system.
Mode	Both, *degradation modes* and *detection modes*, can be used to refer to a certain state of the system (e.g., status of detection or limp-home mode).
Expression (expr)	An expression can be a malfunction or a mode.
Expression Set (expr-set)	Is a set of expressions that do not necessarily occur at the same time. Similar to the concept of a cut-set.
Perm	It is an unary predicate that is applied to an expression. When we apply it to such an expression, e.g., `perm(expr1)`, it means that this expression holds for all coming states of the system. In LTL terms `perm(expr)` is equivalent to `G(expr)`

Table 4.3 – Safety patterns attributes (Oertel, Mahdi, Böde, & Rettberg, 2014)

4.2 Gap Analysis and Requirements

There are three main sources for the requirements on the modular safety view, which is to be developed in this chapter. First, the requirements directly resulting from the impact analysis process, which are summarized in section 3.4. On the other hand, the analysis of the existing safety modeling approaches revealed some missing features or combinations of features that are necessary in order to be usable in an impact analysis. In addition to these technological aspects, the ISO 26262 (2011) has requirements on the content of the functional safety concept, which is the target of the specification. The last two aspects are detailed in this section.

4.2.1 Gap Analysis

During the analysis of the existing safety modeling approaches it has been observed, that none of the techniques suited the needs of a change impact analysis. Some approaches like GSN (Kelly & Weaver, 2004) or SHIP safety cases (Bishop & Bloomfield, 1997), do not provide the needed degree of formalization. Hence they are suited to state a safety argumentation, but automated analyses, as required by the change impact analysis (see Section 3.4), are not possible to realize. Approaches using formal languages as presented by Sljivo, Jaradat, Bate, and Graydon (2015) or Arts, Dorigatti, and Tonetta (2014) provide the possibility for automated processing of requirements, but do not allow for specifying requirements in the safety aspect (see Section 2.2.2). Safety patterns that allow a formal specification of properties assigned to the safety aspect have been introduced by Damm et al. (2009) or Böde et al. (2010) as well as Oertel et al. (2014). Although the approaches differ from each other, none of them is able to support abstraction techniques in their specification language. Other approaches like Hip-Hops (Pasquini et al., 1999) or FPTC (Wallace, 2005) at least claim to support modularization. However, modularization does not imply that an abstraction mechanism is available. While modularization describes a logical partitioning, abstraction describes the same element in less detail, over-approximating the behavior.

The safety concepts of the ISO 26262 are described in a top-down process, starting with very abstract requirements that are refined together with the structural architecture. Thus, refinement of requirements is an essential aspect of the change impact analysis and none of the analyzed safety modeling approaches can be applied without. Nevertheless, safety patterns provide the largest set of needed properties. Hence, they have been chosen to be extended with the missing features.

4.2.2 Specification Needs from ISO 26262

To be able to express functional safety concepts according to the ISO 26262 a detailed analysis of the stated requirements is necessary. The ontology depicted in Figure 4.10 represents the summary of the identified concepts and relations, that are relevant for expressing functional safety concepts.

The functional safety concept consists of the preliminary architectural assumptions and the functional safety requirements, which describe the intended functionality of the components inside of the architecture.

The preliminary architectural assumptions consist of hierarchical composed *components*. The intended behavior of the components is expressed in terms of *safety requirements*. In contrast to the strict separation between functional and safety requirements in the SPES and CESAR design methodology (see Section 2.2.2). the ISO 26262 considers all requirements as safety relevant as long as they describe a function that is considered at least ASIL A. The main objective of the functional safety concept is to describe how *faults* are detected and *failure* mitigated. Hence, safety requirements may be violated if too many faults or failures are present in the system. This can be only one fault (which is after the analysis considered as a single point fault) or cut-sets with multiple faults. In this context we do not need to further distinguish between faults and failures (see definitions in Section 1.5 and an additional explanation in section 4.3.1). Nevertheless, the differences in the structural location of the occurrence of the unintended deviation, which is often considered as a main difference between fault and failure, needs to be stated. Hence, deviations on signals and deviations on the internals of a component are separated. As a result, a component can be specified according to the input and output signal deviations, as well as internal deviations, which is compliant to the failure logic modeling approaches (see Section 4.1.1). In addition to the structural occurence of faults and failures, the ISO 26262 distinguished between *permanent faults* and *transient faults*. Permanent faults are always existent in the system once they occurred, while transient faults may disappear and occur again, even with very high frequencies. The ISO 26262 provides multiple other fault classifications that are not considered here. For

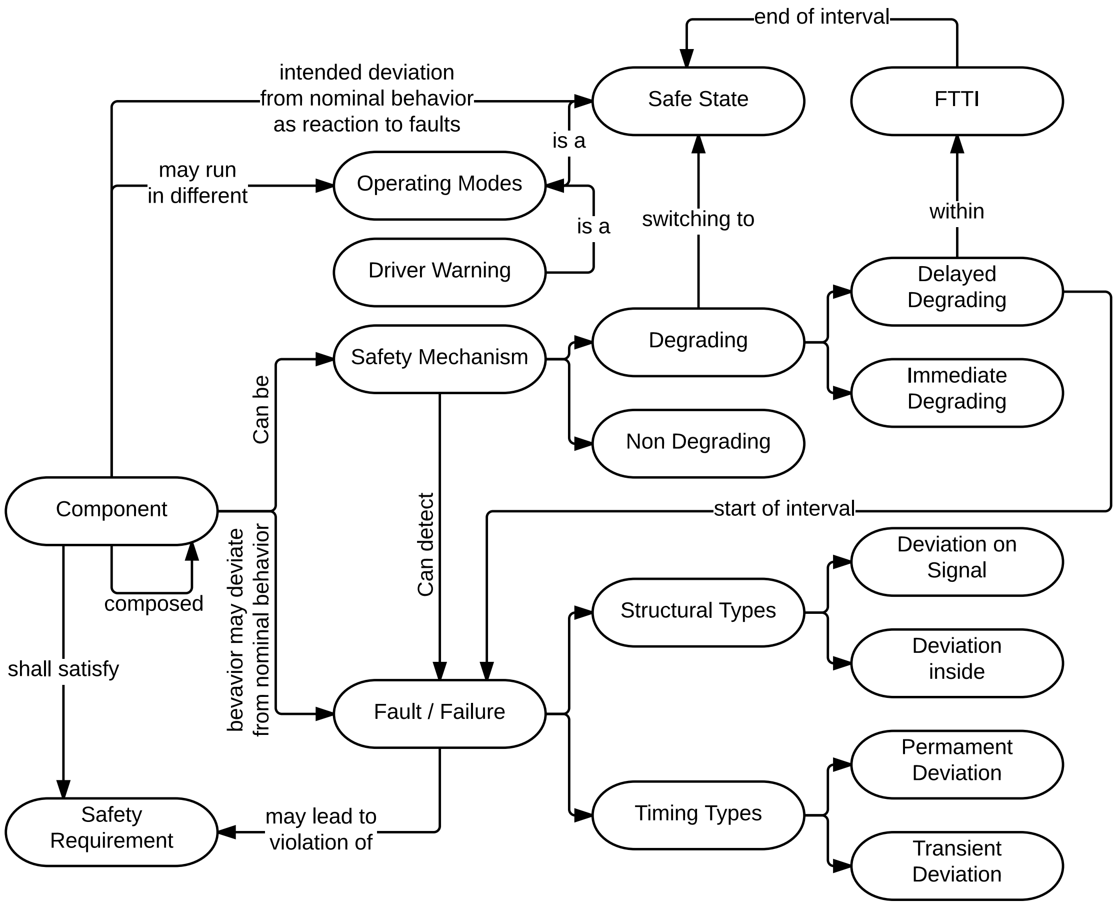


Figure 4.10 – Ontology representing the chosen abstraction for describing safety concepts.

example, the distinction between single-point or multi-point faults is not represented in the overview, since these classifications are analysis results with respect to the effect of the malfunction on the system. The same applies to the subclassifications of them, such as safe faults, residual faults or latent faults. In contrast, the properties *permanent* and *transient* describe the nature of a fault and a static property, respectively.

The ISO 26262 foresees multiple possibilities for how to deal with detected faults and failures in the system. These activities are executed by components that implement *safety mechanisms*, and hence, are directly influencing the fault propagation behavior. The first technique mentioned by the ISO 26262 is the degradation of a system, by switching it to a so-called *safe state*. This safe state provides a reduced functionality of the device, but is safe with respect to the risk identified with the detected fault. As an example, a light sensor of a car can switch the light of the vehicle permanently on if there is a fault or failure in the system detected. Hence, the function is degraded, since the automatic light feature is disabled, but the risk of a sudden loss of main light during driving in the dark is reduced. As a second possibility to deal with occurring faults, *fault tolerance* *mechanisms* are suggested. Safety mechanisms using fault tolerance techniques, such as multi-channel architectures, keeping the system functional, even in the presence of a defined set of faults. For these elements the arbitration logic shall also be specified, for example, how to select a signal from simultaneously generated ones.

The functional safety requirements shall further consider the possibly existing *operating modes* of the system. These could be special driving modes like "sport" or "eco" or also country specific modes for light and speed management. We also consider the safe-state or a "limp-home" mode as operating modes of the components. In addition the *fault tolerance time interval* (FTTI) needs to be specified for components that shall perform error detection or mitigation. The FTTI is the time from the occurence of the fault until it is mitigated or a safe state has been reached. Hence, the FTTI is only relevant for safety mechanisms that do not immediately perform a correcting measure and have a delay between the fault occurance and the detection or mitigation. A typical example for such delayed safety mechanisms are watchdogs, which can detect a fault only after their monitoring period has been exceeded. Conceptually, the *emergency operation interval*, which defines the time until an emergency operation shall be executed if a safe state cannot be reached, can be identically handled as the FTTI.

In addition to correcting faults or degrading the system, the ISO 26262 also foresees the possibility of warning the driver with, for example, a malfunction indication lamp, if the type of the fault allows such a strategy. Similarly to the safe-state the driver warnings can be represented using operating modes. Furthermore, if assumptions on the behavior of the driver or external measures are made, they need to be stated.

4.2.3 Requirement Summary

The requirements on the modular safety view from Section 3.4, Section 4.2.1 and Section 4.2.2 are:

- Requirements shall be represented as contracts.
- The refinement property of the requirements shall be automatically analyzable.
- The compliance of the requirements to implementations shall be automatically analyzable.
- The concepts used in the development of functional safety concepts according to ISO 26262 shall be represented, as described by Figure 4.10.
- The language to express the requirements shall provide means for abstraction.

4.3 A Specification Language Supporting Impact Analysis

Having specified the requirements on the compositional safety model, we are developing the concrete specification language in this section. First, we are adapting the language for the individual assertions in Section 4.3.1 and integrate them later on in contracts (see Section 4.3.2). We then present an abstraction technique in section 4.5.1 and demonstrate how to automatically analyze refinement and satisfaction properties in Section 4.5. Furthermore, guidelines on how to build an initial architecture are given in section 4.4.

4.3.1 Expressing Assertions using Safety Patterns

From the analyzed specification mechanisms in section 4.1 the safety patterns (Oertel et al., 2014) suit the needs for a semantic impact analysis best. They provide defined semantics in LTL and can therefore be easily integrated in a contract-based approach. They already provide means to represent cut-sets and argue about faults and failures. Furthermore, the defined modifiers, such as *perm*, distinguish between delayed and nondelaying safety mechanisms as well as permanent and intermittent faults. Still, two modifications are needed to use safety patterns to perform impact analysis in automotive functional safety concepts. In hierarchical systems the notion of fault and failures need to be adapted (see Section 4.3.1), and multiple failure modes need to be specifiable to be able to represent safe states (see Section 4.3.1).

Making Safety Patterns Hierarchy Ready

The existing safety patterns (Mitschke et al., 2010; Oertel et al., 2014) distinguish between faults and failures in their expression sets. The faults represent the causes that result in defined combinations in a failure on an output port of the respective component. This distinction is suited only for nonhierarchical systems. A failure of one component might be a fault to a connected component (see ISO 26262 (2011) Part 10, p.6, Figure 5). Therefore, this classification is not suited for a static model and the abstract concept *malfunction* has been chosen. This design decision is similar to the "*fault/failure*" element in EAST-ADL. The ISO 26262 uses the term malfunctioning behavior. The main difference to classical fault and failure notions is the focus on the location of deviation. Hence, the malfunction within one component is called *internal malfunction* and can potentially propagate to another component in the form of *signal malfunctions*. The signal malfunctions can be used to "connect" the fault propagation between different components, while the internal malfunctions can be used for abstraction and refinement techniques (see Section 4.3.3). Still, it is not necessary to refine every internal malfunction. At the lowest level of refinement they can be used as atomic malfunctions.

The malfunctions need to be described in detail. This concrete deviation of the signal to its intended value (sometimes called *failure mode*) can be described in various ways. McDermid (McDermid & Pumfrey, 1994) suggests, for example, considering failure modes like "ommision" and "commision", which indicate that a signal has not been sent or was sent unintended. There are also classifications (Bondavalli & Simoncini, 1990) that basically concentrate on distinguishing between "correct" and "incorrect" values. The choice of a fitting level of abstraction depends upon the use-case and the state of development. In the early phases of the development lifecycle the detailed malfunctions might be unknown, and more abstract models are more practical.

The concept of malfunctions does not cover the concept of an *error* that does not appear as a separate concept in the ontology (see Figure 4.10) either. The error describes the component's internal incorrect state, which might manifest in a signal malfunction when the component is used. Since we are concerned about the safety concept, the description of the interface is in the focus. The implementation, which is typically not available at that point in time of the development cycle, is not important for the concept. Nevertheless, if the implementation is available it is essential to ensure that the requirements resulting from the safety concepts are correctly implemented. A technique to ensure this relation is presented in section 4.5.2. Since this analysis is based on existing fault injection techniques, the separation of fault and failures is relevant again. Nevertheless, to support analyses that are using this classification, the information is still extractable from the model. Signal malfunctions on input ports and internal malfunctions can be considered faults, and signal malfunctions on output ports can be considered as the failure.

Introducing Multiple Deviations per Function

An essential capability of the functional safety concept is to express the degradation behavior of the system. In this so-called degradation concept, the safe-state represents a deviation from the nominal intended behavior, but keeps the system in an operating mode, that is still safe for driver and environment. Hence, the safe-state can technically be considered a malfunction of the system as well. In contrast to malfunctions that describe, for example, a *too high* value or a *too late* execution, the value of a safe-state is precisely defined. It is therefore not a representation of a "safe" deviation range, but the establishment of a well-defined state.

It is the goal to state a safety requirement that the system shall be working correctly or at least safe, if only a limited amount of malfunctions are present in the system. This means, that a relation between two failure modes for the same functional port need to be specified.

The currently existing safety patterns (Mitschke et al., 2010; Oertel et al., 2014) use a cut-set representation for faults and failures. An example pattern stating that the combination of malfunctions m_1, m_2 and m_1, m_3 are not permitted, looks like:

$$\textbf{none of}\{\{m_1, m_2\}, \{m_1, m_3\}\}\textbf{occurs.}$$

The cut-set indicates that these combinations of malfunctions are not permitted on any trace of the system, but there is no information about the time of the occurrence of the malfunctions included. The LTL representation of the patterns is:

$$(\mathbf{G}\neg m_1 \vee \mathbf{G}\neg m_2) \wedge (\mathbf{G}\neg m_1 \vee \mathbf{G}\neg m_3)$$

Hence, any trace on which the malfunctions m_1 and m_2 or m_1 and m_3 are present are not accepted. If requirements of a safety mechanisms shall be stated that degrades the system by switching to a safe-state, cut-sets are not sufficient anymore since a temporal

$\langle \mathit{malfunction} \rangle$ ::= $\langle \mathit{andExpr} \rangle$ | $\langle \mathit{singleMalfunction} \rangle$;

$\langle \mathit{not} \rangle$ ::= '!';

$\langle \mathit{singleMalfunction} \rangle$ ::= [$\langle \mathit{not} \rangle$], ALPHA, {(ALPHA | '_')};

$\langle \mathit{andExpr} \rangle$ ::= $\langle \mathit{singleMalfunction} \rangle$, '`and`' , $\langle \mathit{singleMalfunction} \rangle$;

Figure 4.11 – Extension of the existing safety pattern grammar with "combination" and "not" operator.

relation between malfunctions needs to be expressed. Only the traces are accepted where the system is either safe or correct. In LTL terms:

$$\mathbf{G}(signal_{safe} \lor !signal_{fail})$$

To integrate this concept in the existing grammar of the safety patterns the existing rules to write elements of the cut-set can be used if we bring the above statement in the form:

$$\mathbf{G}\neg(\neg signal_{safe} \land signal_{fail})$$

Hence, the new operators `and` and `not` can be added in the grammar that refines the concept malfunction (see Figure 4.11).

The `andExpr` m_1 *and* m_2 simply translates to:

$$m_1 \land m_2$$

The not expression simply translates to the negation symbol '!'

Hence, the contract:

$$\{\{signal_{fail}\ \texttt{and}\ !signal_{safe}\}\}\textbf{does not occur.}$$

translates to:

$$\mathbf{G}\neg(signal_{fail} \land \neg signal_{safe})$$

Using multiple malfunctions per functional port, we need to extend the structural constraints on valid architectures stated in section 3.3.1.

Structural Constraint 5. *If multiple malfunctions are assigned to a functional port, the components using the functional value need to be connected to all stated deviations of the signal.*

4.3.2 Expressing Safety Contracts

Contracts, as introduced in Section 2.1, provide a formal semantics to define refinement and composition operators, by separating a requirement into an assumption and a guarantee. The modified safety patterns presented in the previous section fulfill the requirements to state functional safety assertions as required by the ISO 26262. That is, they are able to address all relevant entities. Still, to express the relations between the elements, we need to embed the pattern in a contract. Hence, to be able to perform virtual integration testing later on, it is important to state the right assumptions and right promises. In current evaluations of contracts for automotive systems, the uncertainty in stating the right assumptions is a major cause for restrained use of contracts in that domain (Föster, 2012). As a result, strong emphasis needs to be put on the activities to support the engineer in specifying the safety contracts, in particular the assumptions. We provide four basic contract templates, which are directly related to the classification of safety mechanisms as presented in Figure 4.10. Hence, the degradation capabilities and the timing behavior of the detection are the main distinctions between the here provided templates.

Template for Nonsafety Mechanisms

Components that do not implement safety mechanisms propagate all incoming malfunctions to the output ports. For each output port, contract C_1 is considered as a template.

C_1		
	A:	**none of {** $M_a \subseteq \{\{m\} \mid m \in M_C\}$**} occurs.**
	G:	**{output_mf} does not occur.**

The set M_a is a subset of all malfunctions of the component C, either on input ports or internal ones (M_C). In M_a all the malfunctions shall be listed that can potentially cause the output-malfunction to occur. Since no safety mechanisms shall be implemented by component C, all input malfunctions assigned to ports that are needed to create the output, are part of M_a. In addition all internal malfunctions that can cause the output malfunction to occur. Not necessarily all internal malfunctions need to cause an output to fail. E.g., an internal malfunction that causes AD-conversion to be less accurate will not affect output ports, which results do not rely on any analog signals.

Such a contract needs to be stated for all possible malfunctions on the output ports of a component.

Template for Nondegrading Safety Mechanisms

A safety mechanism is responsible for detecting or mitigating an unwanted system state. In contrast to the previously stated contract template, a safety mechanism is able to limit the possible malfunction combinations that lead to unwanted situations. There is no special representation of a safety mechanism needed in the architecture, but it is

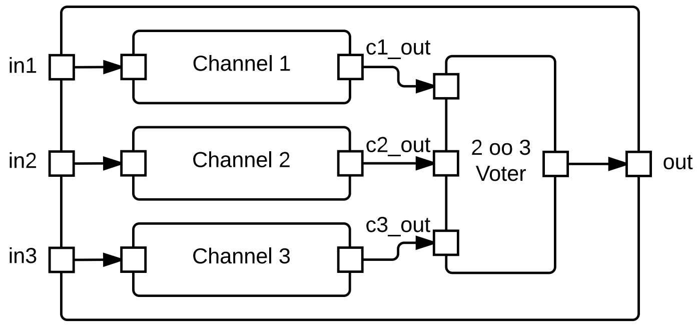


Figure 4.12 – Triple modular redundancy (TMR) as an example for a nondegrading safety mechanisms

advised to introduce one for clarity.

Figure 4.12 depicts a triple modular redundancy (Moore & Shannon, 1956) safety mechanism, which is one of the most common nondegrading safety mechanisms. Three channels implement redundantly the calculation of the output, and pass their results to a voting component. The voting component compares the results and outputs the value, which is identically calculated by at least two channels. Obviously, this approach can only guarantee a correct result if at most one channel fails. Hence, the contract template for nondegrading safety mechanisms looks like:

C_2	**A:**	**none of {** $M_a \subseteq \mathcal{P}(M_C)$**} occurs.**
	G:	**{output_mf} does not occur.**

The set of malfunction combinations M_a in the contract template is a subset of the powerset of all existing malfunctions M_C on component C. In M_a all combinations of malfunctions are defined, that may potentially cause the output malfunction `output_mf` to occur.

In contrast to components that do not implement safety mechanisms the size of the cut-sets can be larger than one. In the above stated example of the TMR module, one failed channel may not harm the correctness of the output. Hence, the cut-sets of this example safety mechanisms will contain only two elements. Assuming there is an input malfunction for each input and an internal malfunction for each channel, the M_a would be:

$$\begin{aligned}
&\{\{in1\_fail, channel2\_fail\},\\
&\{in1\_fail, channel3\_fail\},\\
&\{in2\_fail, channel1\_fail\},\\
&\{in2\_fail, channel3\_fail\},\\
&\{in3\_fail, channel1\_fail\},\\
&\{in3\_fail, channel2\_fail\},\\
&\{channel1\_fail, channel2\_fail\},\\
&\{channel2\_fail, channel3\_fail\},\\
&\{channel1\_fail, channel3\_fail\},\\
&\{in1\_fail, in2\_fail\},\\
&\{in2\_fail, in3\_fail\},\\
&\{in1\_fail, in3\_fail\}\}
\end{aligned}$$

Note, the combinations of input malfunction at the same failed channel are not considered here. In this architecture the combination of these faults would not lead to wrong output singal, since two correctly working channels exist. Furthermore, we have not assigned an internal malfunction to the voting component. This is a common principle in the design of safety critical systems (Bozzano & Villafiorita, 2011). The voting component is a single-point of failure. The only way to prevent this is an architecture that uses three voters and all functions of a system are implemented three times (Von Neumann, 1956; Moore & Shannon, 1956). This architecture is typically used in airplanes but not in automotive systems. Still, the ISO 26262 requires the freedom of single point faults "This requirement applies to ASIL (B), C, and D of the safety goal. Evidence of the effectiveness of safety mechanisms to avoid single-point faults shall be made available." (ISO 26262 (2011), Part 5, Req. 7.4.3.3). To be able to meet this requirement, the voting component is being designed in a more robust way than the other parts of the system. This includes techniques such as formal verification of the voting logic, which is feasable in short time because of the low complexity of the arbitration logic. Furthermore, the ISO 26262 does not consider faults resulting from defects in the execution platform as single-point of failures if at least one safety mechanism exists to monitor the platform: "If a hardware part has at least one safety mechanism (e.g. a watchdog for a microcontroller), then none of the faults of that part are classified as a single-point faults. The faults for which the safety mechanisms do not prevent the violation of the safety goal are classified as residual faults."

Template for Immediate Degrading Safety Mechanisms

In many cases it is not possible to implement nondegrading safety mechanisms, because either the necessary redundancy is too costly or other constraints, such as available space or computing resources, prevent such an approach. Hence, degrading safety mechanisms

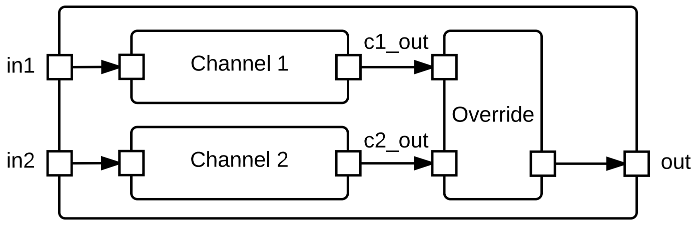


Figure 4.13 – Immediately degrading safety mechanisms implemented by two redundant channels

provide a cheaper possibility for switching to a safe-state after detecting malfunctions in the system. This degradation may happen immediately after the fault occurs or some time after.

Figure 4.13 depicts a two-channel voting architecture as an example for an immediately degrading safety mechanism. Both channels implement a functionality redundantly and the voter compares the signal. In contrast to the nondegrading safety mechanism the voter is not able to determine which is the correct signal, but is only able to detect a deviation in the signals. Hence, switching to a safe state is the only possible action to operate the system safely.

Contract C_3 depicts the template for the immediate degrading safety mechanism.

C_3		
	A:	**none of {** $M_a \subseteq \mathcal{P}(M_C)$**} occurs.**
	G:	**{output_mf and !output_safe} does not occur.**

The guarantee of this contract deviates from the promises in the previous two templates. The safety mechanism is able to guarantee that the system is *correct OR at least safe* even considering cut-sets of size > 1. This is expressed by two separate malfunctions of the output port. The malfunction *fail* simply indicates that the output is different than specified in case the system is working correctly and the malfunction *safe* indicates that the system is in its safe state. The safe state can be observed at the output easily, since the safe state is a known value (like light on, in case of the light sensor). With respect to the example in Figure 4.13 M_a is given as:

$$\begin{aligned}\{\{in1\_fail, channel2\_fail\},\\ \{in2\_fail, channel1\_fail\},\\ \{in1\_fail, in2\_fail\},\\ \{channel1\_fail, channel2\_fail\}\}\end{aligned}$$

The handling of the voting component needs to be identical as in case of the non-degrading safety mechanism.

In addition to the stated contract C_3 an additional contract needs to be stated that requires the correct signal in case of no malfunctions in the system (see Contract C_4).

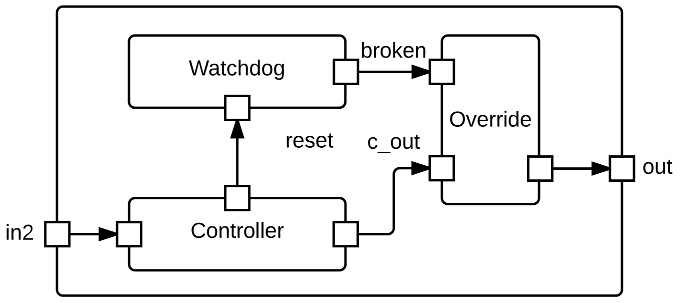


Figure 4.14 – Watchdog as an example for a delayed degrading safety mechanism

Therefore, the assumption lists all malfunctions, and the guarantee lists the output malfunction. We call this contract also a *performance safety contract*, since it is not necessary for the safety analysis. Without this contract it would be a valid behavior of the system to always remain in the safe state. This is safe, but obviously does not comply with the expectation of the customer to receive an operational item.

C_4		
	A:	**none of** $\{M_a \subseteq \mathcal{P}(M_C)\}$ **occurs.**
	G:	**{output_mf} does not occur.**

The safety mechanism with two redundant channels is only an example of many other techniques in this category. Still, the architecture will mostly look identical, though the degree of the redundancy differs. For example, a consistency check, which validates the result with respect to an estimate or defined bounds, does not need to fully implement the monitored function. A less precise calculation can be sufficient to identify wrong results. Similarly failed CRCs in transferred messages use information redundancy to be able to detect a well-defined type of signal malfunction.

Template for Delayed Degrading Safety Mechanisms

In some cases it is not possible to create any redundancy that is suitable for a safety mechanism. One typical example is the microcontroller itself. Although some more expensive devices provide lock-stepping of two redundant cores (Functional lockstep arrangement for redundant processors, 1993), the majority of systems is using watchdogs to detect faults in the hardware platform. As depicted in Figure 4.14, the controller needs to re-set the watchdog-timer on a regular time base. If this re-set is not performed, the watchdog assumes that the controller is malfunctioning and can switch the system to a safe-state. Hence, the time between the occurring malfunction and the detection of it is defined by the re-set interval of the watchdog. During the occurrence of the malfunction and the detection, the component is producing wrong results. Depending upon the function the system is implementing, this behavior is acceptable as long as the malfunction will be detected within some appropriate time period (the FTTI).

Contract C_5 can be used as a template to describe delayed degrading safety mechanisms.

C_5 **A**: **none of { $M_a \subseteq \mathcal{P}(M_C)$} occurs.**
G: `{perm(output_mf and !output_safe)}` **does not occur.**

While the handling of the assumption is identical to the immediately degrading safety mechanism, the guarantee is using the `perm` operator to relax the guarantee. Hence, the contract guarantees that the situation in which the output delivers a wrong result and no safe-state has been established, is not permanent. In terms of LTL, the `perm` operator translates to *globally*. Hence, the promise of contract C_5 translates to:

$$\begin{aligned} & G(\neg G(output\_fail \wedge \neg output\_safe)) \\ \Leftrightarrow \quad & G(F(\neg output\_fail \vee output\_safe)) \end{aligned}$$

Hence, at each time in the trace there must exist a point in the future where the system is either safe or correct.

Again, it is advised to state also a performance safety contract as depicted in Contract C_4.

4.3.3 Abstract Safety Specifications

Following the templates given in section 4.3.2 we are able to specify a functional safety concept on one level of abstraction. We have not yet considered how specification on multiple component levels (nested components) need to be formulated.

The approach, which is taken by the existing safety specification, is based on modularization rather on abstraction. Approaches like Hip-Hops or Pasquini et al. (1999) or Wallace (2005) compose the already existing properties of a set of subcomponents to a single larger component.

This means that fault propagations are described on the component level and can be modularized to larger components, but the composed specification is still using the constructs of the subcomponents. As an example in Oertel et al. (2014) the top level safety contract is using all non-desired atomic malfunction combinations in its assumption, even if the atomic malfunctions correspond to components of a more detailed abstraction level. Figure 4.15 illustrates that all faults in the most detailed (lowest abstraction level) components need to be made available to the containing components by use of fault ports. Hence, these specific malfunctions are “externalized” to the surrounding components. This approach is not compositional, since changes on internal components necessarily causes re-verification effort on the containing component. Furthermore, a top-down design is impossible, since it is assumed that all components and their faults are known in advance.

This problem is directly related to the identified gap on the current existing safety specifications (see Section 4.1 and 4.2.1), that is, the lack of abstraction possibilities. These techniques are necessary in order to support the top-down process of the ISO 26262. The main argumentation path of the ISO 26262 is to identify early the possible hazards that a system might produce and try to prevent them in the functional safety concept.

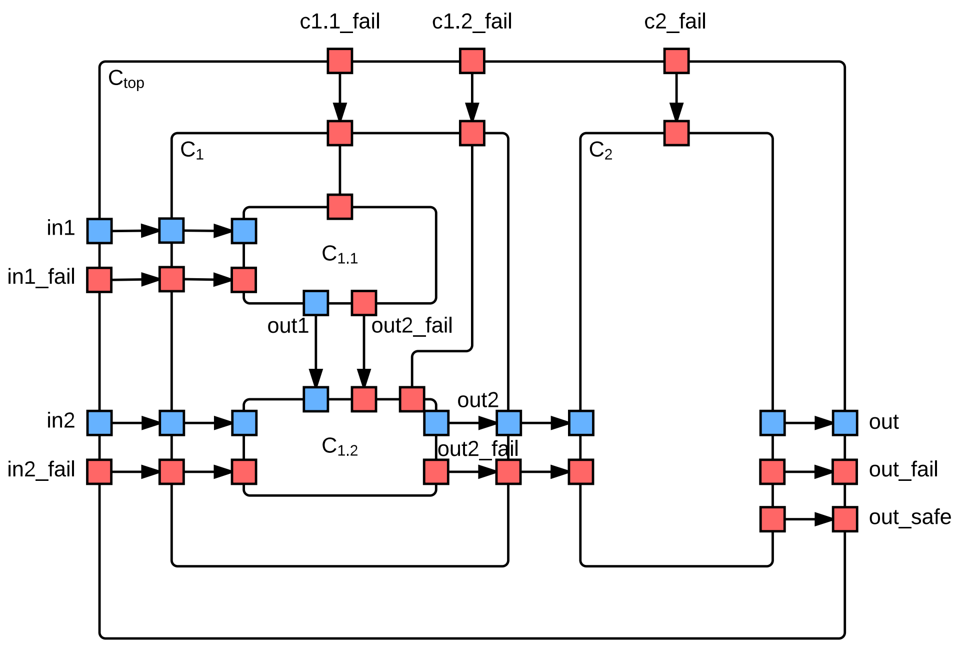


Figure 4.15 – All internal faults were "externalized" using fault activation ports

After the verification of the functional safety concept. The requirements are further refined towards technical safety requirements. Hence, the malfunctions at the lowest level of abstraction are not yet known. And even in the FSC, refinement is extensively used. Bottom-up techniques would invalidate all gained verification results, because they are not mappable to the existing verification results (see also Figure 4.15).

To enable a top-down design, the specification needs to be designed in a way, that the early gained analysis results still maintain their validity even if new subcomponents are introduced. The use of contracts does not provide this property as is. Hence, a fault abstraction concept needs to be introduced. Looking at figure 4.15 it can be observed, that the malfunctions on the input and output ports are handled correctly. This means, that the contract describing the top level component does not need any externalized malfunctions on input and output ports. Each component has its own input and output malfunction ports, which are local to the component. Nevertheless, to handle internal malfunctions, a new approach is needed.

A Fault Abstracting Specification

To identify a suitable solution the actual goals of the assumptions have to be identified. According to the templates provided in Section 4.3.2, the assumption states all combinations of malfunctions that can potentially cause an output to fail. Still, the important information is the minimal size of the cut-sets, which represents if the component does provide a certain level of fault tolerance. Hence, instead of specifying the concrete set of internal malfunctions it is sufficient to specify the minimal size of the cut-sets. This gives the engineers freedom in how to implement the component. Using this technique we are still able to express the absence of single-points of failure, as required by the ISO 26262, but do not need to state all particular faults individually. Thereby this generalization enables a top-down design approach.

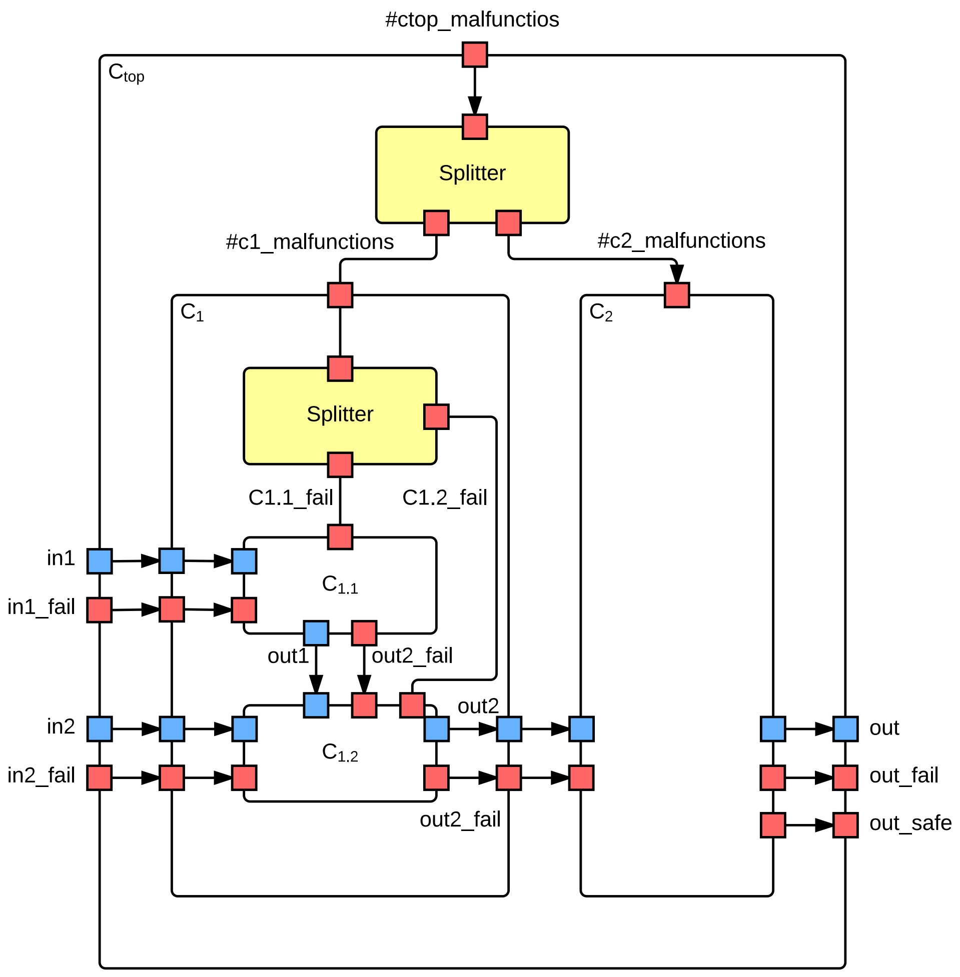


Figure 4.16 – Count ports are introduced and fault-splitter components ensure that no more faults are passed to internal components than specified

Still, an technical solution is needed, that integrates with the contract-based design principles and is still compatible with the defined contract templates and other identified requirements on the safety specification.

Therefore, since contracts are defined over traces on values of ports, we introduce *malfunction count* ports representing how many internal malfunctions are present in a component (see Figure 4.16). Only at the most detailed abstraction level (i.e., the implementation) do the particular malfunctions that might occur in a component need to be specified. The fault count ports are of type Integer and belong to the safety aspect of a system only. As for the fault ports, these ports are virtual and do not occur in any generated or manually produced implementation. The malfunction count ports can be used as placeholders in the cut-sets specified in the assumptions to represent any combination of internal malfunctions of the given size. Hence, the malfunction count ports cannot be simply connected to the subcomponents, but additional constraints need to be fulfilled. Since only components can be associated with a specification and not the signals by themselves, we need to introduce virtual *splitter components* to distribute the maximum number of malfunctions of the surrounding component to its subcomponents. The splitters are necessary to allow only traces that do respect the number of malfunctions, while still leaving open the choice of how to distribute them across the subcomponents (see Figure 4.16). For n subcomponents the contract associated to the splitter component can also be characterized by the following contract:

C_6
```
A: | true
G: | out1 + ...  + outn = #internal_mf
```

The guarantee creates a relation between the input port of the splitter (#internal_mf) and the output ports (out_n). If a cut-set in the assumption includes an internal malfunction count of x, this contract ensures that exactly x faults are activated on the subcomponents.

This approach is still compliant to the contract-based design principles, since all stated contracts only refer to its input and output ports. Hence, all defined relations and operators, like refinement and composition, are still applicable.

Intermediate and top level component's safety requirements can be specified using the malfunction count ports. The most common usage for the top level assumption is to state that only one or two faults occur in the system to express that there shall be no single or double faults in the system. For example, a specification of the component C_1 in Figure 4.16 could be stated as follows:

C_7
```
A: | none of {{in1_fail, in2_fail},
   | {C_1_#internal_faults=1, in1_fail},
   | {C_1_#internal_faults=1, in2_fail},
   | {C_1_#internal_faults=2} } occurs.
G: | {out2_fail AND !out2_safe} does not occur.
```

This specification defines, that the component shall be robust against one arbitrary malfunction, which may occur at one of the inputs or internally. Another useful scenario is that the inputs are assumed to be correct and only internal faults are considered. Such a contract can now be stated in a very short way:

C_8
```
A: | none of {{C_1_#internal_faults=2} } occurs.
G: | {out2_fail AND !out2_safe} does not occur.
```

The specification of an atomic component (like C_1.2 in Figure 4.16), for which the internal malfunctions are known, uses these atomic internal malfunctions, since no abstraction by using a fault count is necessary. The internal faults are represented as individual fault ports (see C_1.2_fault on component C_1.2 in Figure 4.16) and are directly connected to the splitter components. This ensures, that the specification of the atomic component can still be done accordingly to the guidelines presented in section 4.3.2 however one can additionally restrict the total number of different individual faults occurring simultaneously (in the same trace) on a more abstract and simpler level.

Semantics and Verification of Malfunction Abstraction

After having described the main idea and usage of the fault abstraction technique, the precise semantics need to be defined in order to be able develop an automatic analysis framework. The safety pattern semantics are defined using LTL (Pnueli, 1977). Since fast LTL model checkers are available, the development of a new semantic base should be avoided. Furthermore, the fault count ports should be defined using LTL. Nevertheless, since LTL does not provide a representation of integer values, a mapping to boolean logic needs to be created. Still, from a user's perspective the workflow does not change. The specification should be written using the integer fault ports, and the mapping performed completely automatically.

Since all malfunction count numbers are explicitly stated, and we do not consider unbounded variables, the integer ports can be expressed in a combinatorial fashion. Hence, a malfunction count port is split up in individual boolean ports (*explicit fault count ports*) representing each a valid value of this port (see Figure 4.17). If a contract assumes that n malfunctions do not occur in the system, all numbers of faults smaller than n are valid values. It needs to be stated that only one of these ports can be active at a time.

```
      A:  true
      G:  none of {
          {0_faults_Ctop, 1_faults_Ctop},
C_9       {1_faults_Ctop, 2_faults_Ctop},
          {0_faults_Ctop, 2_faults_Ctop}
          } occurs.
```

Furthermore, we need a separate splitter component for each explicit fault number port. These splitters allow all logical combinations of ports that sum up to the defined number of faults. For n internal faults and m subcomponents the splitter contract restricts the set of all possible occurring malfunctions $M = \mathcal{P}(\{X_0\_faults\_C0, \ldots, X_n\_faults\_Cm\})$ to M_f:

C_{10}
A: `true`
G: `none of {`M_f` } occurs.`

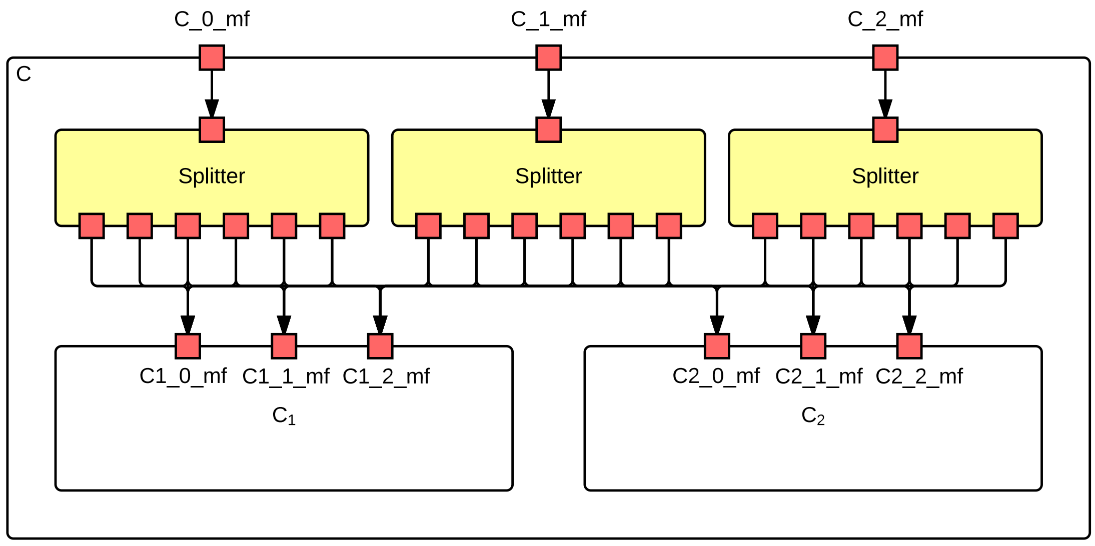


Figure 4.17 – Explicitly represent the counting with boolean logic for LTL implementation. In this example a fault port with the value 2 is represented.

with $M_f = \{\{x_i\_faults\_C_y\} \subseteq M : \sum_{i=1}^{(n+1)\cdot m} x_i \geq n\}$.

Nevertheless, for the purpose of a single refinement analysis we can simplify the translation, since at analysis time splitters are not needed and can be replaced by all combinations of faults of the subcomponents directly. For a given set of internal faults I the malfunction-count port `#internal_faults=n` in the assumption resolves to all sets of faults of length n of the powerset $\mathcal{P}(I)$. The refinement check is then implemented as a satisfaction check on the LTL formulas of the contracts (see Section 2.1.5). Rozier and Vardi (2007) as well as Li, Pu, Zhang, Vardi, and He (2014) suggested using a generic model, allowing all possible behavior, to check the property against reducing the problem to a classical model checking problem.

4.4 Process Guidance on Creating an Initial Architecture

In addition to the defined templates for contracts, the process of how to create a safety specification can be guided with a few practices. This is a guideline for describing the safety concept, not for developing the safety concept itself. These guidelines suggest a suitable process, however, other solutions are not excluded.

Design Practice 1. *The architecture and the specification, as well as the functional and the safety aspect shall be defined for one abstraction level in parallel.*

The parallelism of architecture and requirement design is necessary to comply with the structural consistency criterion of contact based design on the one hand (see Section 3.3.1), and to comply with the ISO 26262 process on the other hand. The functional and the safety aspect (see Section 2.2.2) are also typically designed in parallel. This stems from the simple fact that the safety concept shall ensure the safe execution of the behavior defined in the functional aspect. Hence, the number of components and their interconnections are mostly given by the functionality of the component. Even though the malfunctions are defined for the functional ports of the component, the activities of designing the functional and safety specification are interwoven. Although the functional architecture is the base for the first iteration of the safety concept, it is likely that additional components for voters or redundant channels need to be added. These components also fulfill a functionality, and hence, in addition to the safety requirements that state the malfunction propagation, functional requirements also need to be added. For example, the functional requirement for the voter in Figure 4.13 could be:

$$C_{11} \quad \begin{array}{l|l} \texttt{A:} & \texttt{true} \\ \texttt{G:} & \texttt{in1 = in2} \rightarrow \texttt{out = in1} \wedge \texttt{in1} \neq \texttt{in2} \rightarrow \texttt{out = 255} \end{array}$$

If the inputs deviate the output is set to 255, if the inputs are identical, this value is passed through. Hence, in the functional requirement the safe state is not mentioned, but the functional value, 255 in this case, is used. As a result, at least one iteration of the functional and the safety aspect needs to be performed to check the modification on the component structure, which is necessary to implement the safety mechanism and state the missing functional requirements, if possible. It could also be a result of the analysis of the functional aspect that the functions cannot be implemented correctly in the presence of the added safety mechanisms. Possible reasons could be the exceeding of resource constraints like space or timing. Hence, to avoid unnecessary iterations, both aspects should be designed commonly.

Design Practice 2. *Top level safety specifications represent the negated hazards. These may include already inhibition keywords that shall be used for the malfunction mode. Use safe-states if identified in HARA.*

Starting with the top-level component the main functional and safety requirements need to be specified. While the functional requirements typically results directly from the item definition (see introduction to ISO 26262 in Section 2.3), the safety goals are typically the negation of the identified hazards. Since the hazard identification also uses function modifier keywords for the identification of hazards, like “wrong function,” “too much function” or “too early function” these keywords can directly be used to define the avoided output malfunction. Identified safe-states need to be stated: “If a safety goal can be achieved by transitioning to, or by maintaining, one or more safe states, then the corresponding safe state(s) shall be specified” (ISO 26262, 2011).

Design Practice 3. *Describe malfunctions on signals and atomic malfunctions detailed in a table. The focus shall be on how and how much the value may deviate from the intended one if the malfunction is present.*

Although the identifier of the malfunction is sufficient to analyze the refinement of the safety specification, the satisfaction analysis (see Section 4.5.2) needs a detailed description of the effect of a malfunction on the corresponding signal. Still, even if it is not planned to run automated satisfaction checks, the description is useful for the designers of the lower levels in the component structure, since the choice of the proper safety mechanism heavily depends upon the concrete malfunction.

Design Practice 4. *If a fault tolerance time interval is defined for the hazard a delay shall be specified for the corresponding malfunction.*

The decision whether an immediate degrading or delayed degrading specification should be chosen for the top level safety contracts depends upon the existence of a FTTI for the respective hazard. For the safety specification we do not state the concrete time, but use the `perm` operator instead (as indicated in contract template C_5). The verification of the concrete timing properties is part of the timing aspect. For this aspect formal languaages and virtual integration techniques exist that can be used to validate the correct break down of timing requirements such as presented by Damm et al. (2011) as well as Gezgin et al. (2014).

Design Practice 5. *If no further requirements to the degree of fault tolerace exists, assume only one fault within the item.*

As has already been discussed, the freedom from malfunctions on a particular output port of the system can only be guaranteed under special circumstances. Any system, even the most carefully developed ones, may fail if the number of faults exceeds a certain threshold. Hence, the assumptions on top level are critical to the further development of the system. These assumptions of the top level safety contracts are dependent on the malfunctions on inputs that are needed to produce the considered output and the internal malfunctions. If not stated otherwise in the requirements on the item it is a good starting point to avoid single point failures in the system: "Evidence of the effectiveness of safety mechanisms to avoid single-point faults shall be made available [...] [to show] the ability of the safety mechanisms to maintain a safe state, or to switch safely into a safe state" (ISO 26262, 2011). Although this requirement is stated for the hardware design only, a dependent failure analysis to identify these faults is required for the whole system: "The analysis of dependent failures aims to identify the single events or single causes that could bypass or invalidate a required independence or freedom from interference between given elements and violate a safety requirement or a safety goal." Hence, the minimal size of the cut-sets shall be at least two in the assumption. Internal faults are only referenced by malfunction count ports. While developing a system in a distributed manner, cut-sets with single sized elements may occur. This is acceptable if the component shall be implemented by a different supplier, which shall not take care of this individual malfunction, because it might be handled by a different component.

Design Practice 6. *For refined components the higher level component is the context.*

If the top-level component is refined, the context for the specification of the subrequirements is the top-level component. Hence, the malfunction on the output is already defined and all input malfunctions are known. The actual engineering of the solution to the stated problem is starting at that point. It is not in the scope of this thesis to give guidance on how to build good safety concepts; this remains the responsibility of the involved developers. Still, their decisions are documented in the form of safety contracts. On the refined levels the templates can be used as well to describe the requirements on the various components in the architecture.

Design Practice 7. *Virtual integration checks shall be applied for any component refinement to detect design faults in the architecture.*

The automation of the refinement and satisfaction analysis is in particular necessary to provide an automatic impact analysis. Nevertheless, during the development of the initial architecture, the virtual integration check indicates at early design stages that the developed solution for the context defined by the top-level component or another higher-level component, does not cover the needs expressed by this context.

4.5 Analysis of Safety Contracts

To cover the requirements identified during the gap analysis (see Section 4.2.1) for change impact approaches, the specification language for functional safety concepts has to be automatically analyzable with respect to refinement and satisfaction.

4.5.1 Refinement Analysis

To perform a refinement analysis on safety contracts it is not necessary to develop a new technique. Since the properties are strictly defined in LTL the theorems presented in Section 2.1 are sufficient to prove virtual integration.

A technical solution based on these theorems is presented in Section 6.2.

4.5.2 Satisfaction Analysis

As indicated by Figure 3.13, only the lowest abstraction levels of components will be implemented, and the compliance of the system with the requirements will be proven by virtual integration checks once it has been shown that the implementations satisfy the stated requirements on the associated component.

This analysis is typically performed by reviews or testing (Ellims, Bridges, & Ince, 2006). Although advances have been made in the field of automatic testcase generation, testing techniques suffer from their incompleteness, since only a selection of all possible test vectors are applied to the system. As an alternative technique to test safety properties, fault injection (Arlat et al., 1990; Svenningsson, Vinter, Eriksson, & Törngren, 2010) can be used. Still, relying on simulative or experimental approaches, most fault injection approaches do not deliver complete results. To gain completeness, several fault injection analyses that rely on model checking such as (Bozzano & Villafiorita, 2003) and (Joshi & Heimdahl, 2005) have been presented. Another possibility consists in comparing existing fault tree analyses (FTA) or failure modes and effects analyses (FMEA) with the safety specifications (Schäfer, 2003). Again, this is a manual and error prone process. We built upon a formal, model checking-based fault injection technique (Peikenkamp et al., 2006), (Kacimi, Ellen, Oertel, & Sojka, 2014)

Fault Injection

A technique to verify fault tolerance mechanisms is fault-injection. It is defined by Arlat et al. (1990) as "the deliberate introduction of faults into a system." In order to execute a fault injection four inputs are needed. The set of faults F, which is typically described by Poisson processes, and the set of activation A, which can be either represented as stochastic processes or test data patterns. The results of one of the fault-injection experiments is extracted from the set of readouts R. The measures M can only be obtained experimentally and represent the findings of the fault-injection test sequence, for example, a MTBF or a boolean variable that indicated the removal of a fault.

While performing fault injection using a formal model (Peikenkamp et al., 2006) the set of activations is complete with respect to all input combinations and does not need to be specified.

Hence, we refer to a fault-injection as a function FI with three arguments: an Implementation I and a set of faults F and the functional requirement r, representing the measure, that the system shall implement. Each fault $f \in F$ has a formal description $\mathcal{F}(f)$ of the functional deviation if the fault is present. The result is a set C of fault combinations leading to the violation of r.

$$\mathrm{FI}(r, I, F) = C \subseteq \mathcal{P}(F),$$

with $\mathcal{P}$ denoting the powerset.

Analyzing Safety Contracts using Fault-Injection

The fault injection technique uses function properties as a target for verification. These functional requirements are concerned about the occurrence, order or value of signals. In contrast, we want to analyze safety contracts that specify the propagation of faults in the system, addressing the correctness of a signal. The safety ports, which represent the status of a signal are not available in the implementation. These ports are virtual and used for specification purposes only. Still, although safety and functional analyses are mostly performed separated from each other, there is a relation between them, which needs to be detailed, in order to provide the desired satisfaction analysis of safety contracts. Hence, the functional deviation described by the guarantee of the safety contract needs to be determined. This functional deviation is used in the fault injection analysis as a target. All combinations of malfunctions that lead to this deviation need to be identified.

To specify a functional requirement for the FI analysis the definitions of the malfunctions can be used. Nevertheless, the description of the malfunctions are not necessarily completely functional. That is, they may contain a reference to the *correct* value of a signal. Some examples of formal descriptions of malfunctions can be seen in Table 4.4. We will later detail why these references do not need to be resolved. Given the set of cut-sets of the guarantee $\mathbb{C}_p$ of the safety contract and the specification of the malfunction $m \in M$ as $\mathcal{F}(m)$, the functional requirement can be expressed as $\Phi(\mathbb{C}_p, \mathcal{F})$ with:

$$\Phi(\mathbb{C}_p, \mathcal{F}) = \bigwedge_{C \in \mathbb{C}_p} \bigvee_{c \in C} \neg \mathcal{F}(c)$$

To validate if an implementation correctly implements the fault propagation specification given by a safety contract (e.g., C_{13}), we need to identify which combinations of malfunctions cause the behavior given by $\Phi(\{\{c\_wrong\}\}, \mathcal{F})$, where $\mathcal{F}$ is defined by Table 4.4. Figure 4.18 describes the principal relation between a safety contract, the malfunctions of a system and the result of a fault injection analysis.

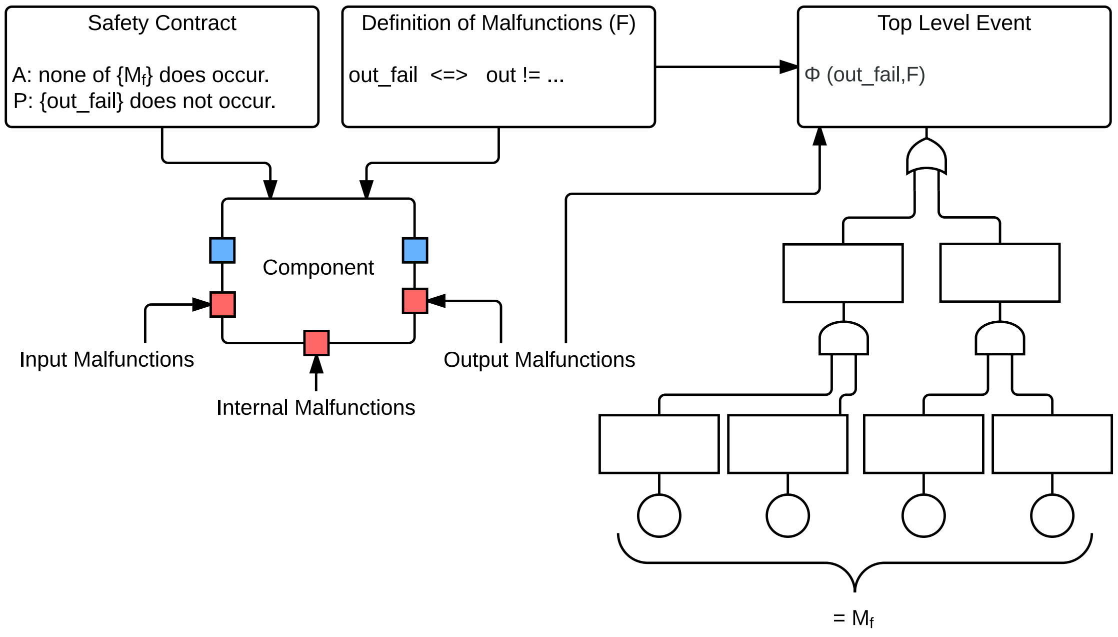


Figure 4.18 – Relation of a safety contract to the FTA results of the implementation of the component

The assumption of a safety contract specifies the combination of malfunctions that potentially leads to a wrong output, which is described in the guarantee. If the output of a component C is prone to a malfunction m, this means that the functional requirement r_f describing the expected value is violated, and instead $\mathcal{F}(m)$ is holding at the output. To verify the safety contract we need to know which malfunction combinations of internal and input malfunctions lead to this deviation on the output. Hence, $\neg\mathcal{F}(m)$ is passed to the fault-injection analysis, together with all definitions of the malfunctions $m \in M$ that might occur in a component. The created minimal cut-sets generated by the fault injection analysis represent all combinations of malfunctions that lead to the specified output deviation. Hence, to verify if the implementation does correctly implement the

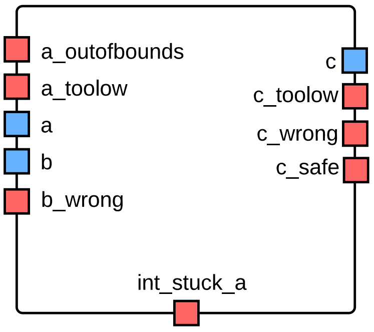


Figure 4.19 – A adder with limited fault tolerance capabilities

defined malfunction propagation behavior, the resulting cut-sets of the fault-injection analysis need to be a subset of the malfunction combinations M_f specified in the safety contract:

$$FI(r, I, F) \Subset M_f$$

where $X \Subset Y$ iff $\forall x \in X : \exists y \in Y : \forall y_i \in y : \exists x_i \in x$. This relation requires X and Y to be a set of sets.

We illustrate the approach on a small example. Figure 4.19 depicts a component, that calculates the sum of two input signals. Hence, the functional requirement r_f of C_{sum} is given as:

C_{12}	**A:**	`always(`$a \leq 10 \wedge b \leq 10$`)`
	G:	`always(`$c = a + b$`)`

One of the safety contracts is given as:

C_{13}	**A:**	**`none of {{a_toolow},{b_wrong},{int_stuck_a}} occurs.`**
	G:	**`{c_wrong} does not occur.`**

Looking at the safety contract C_{13}, the guarantee ensures that the output malfunction `c_wrong` may only occur under a limited set of internal and input malfunctions. The behavior of the components in case of this output malfunction is given as $\mathcal{F}(c\_wrong) = c \neq a + b$. Hence, the functional requirement for the fault-injection analysis is given as:

$$\Phi(\{\{c\_wrong\}\}, \mathcal{F}) =' c = a + b'$$

Table 4.4 depicts a full list of the malfunctions and the resulting formalization $\mathcal{F}$.

It can be observed that internal malfunctions and violations of assumptions on the input ports are easily described. Internal malfunctions refer only to input or output values and directly describe the deviation to the intended functional variables. Similarly

Malfunction m	**Description**	**Formalized**
a_outofbounds	the input exceeds the expected bound	$a > 10$
a_toolow	the input is within the bound, but larger than it should be	$a < a\_correct$
b_wrong	the input is wrong compared to correctly working system	$b \neq b\_correct$
c_toolow	the output is lower than expected	$c < a\_correct + b\_correct$
c_wrong	the output is wrong	$c \neq a\_correct + b\_correct$
c_safe	the output might be wrong but in a safe-sate	$c = 20$
int_a_stuck	hardware defect of the adder causing value a to be stuck at 0	$a = 0$

Table 4.4 – Description of the malfunction of the component depicted in Figure 4.19

the `a_outofbounds` malfunctions directly describe the values of the input that are not expected. The other output and input malfunctions refer to the correct values. For example, the malfunction `a_toolow` indicates that a is smaller than the correct value. The classifier `a_correct` is handled as any other malfunction in the system since it represents a nonexisting deviation of the value of the corresponding signal. It refers to the intended value of the system, which is operating without any present malfunction. Nevertheless, this kind of specification is necessary since the safety contracts describe change propagation and on the output of the top level component a judgment of the overall correctness of the signal needs to be made. Hence, the input and output malfunctions are the main transport element to propagate deviations of signals from one part of the system to another.

However, for analyzing the fault containment properties of an individual component, it is not necessary for the components themselves to judge about the correctness of the input signals. Any possible input could be correct or incorrect, depending upon the state of the system in which the component is embedded. Therefore, we extend the model with an additional component that simulates the input malfunction (see Figure 4.20). The correct value is provided at the input of the extended component and the output of the component is either identical to the input value if no malfunction is present, or, if the malfunction is present, the value is modified according to the description of the input malfunction. Hence, in this additional component, the input malfunction is represented as an internal malfunction. The functional requirement er_f for the extended component is given as:

C_{14}	**A:**	`true`
	G:	`always(`$in\_correct = in$`)`

In case of a malfunction, for example, `in_fail` the deviation $\Phi\{\{in\_correct\_toolow\}, \mathcal{F}\}$ results in "in $\neq$ in_correct." Furthermore, although the analysis is performed locally, we can still reason on the global correctness of output signals if the system is composed of multiple components.

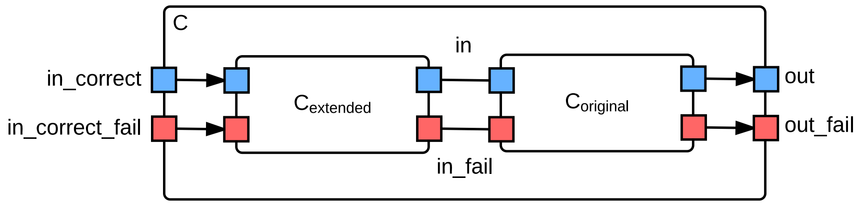


Figure 4.20 – Extending the system model to represent input malfunctions

Safety contracts describing safety mechanisms (see Contract C_{15}) and including a reference to the *safe-state* of the system can be handled identically. For the function $\mathcal{F}$ we assume that it handles composite cut-set elements such as `output_fail and !output_safe` as well. It holds that $\mathcal{F}(m_1$ and $m_2) \Rightarrow (\mathcal{F}(m_1) \wedge \mathcal{F}(m_2))$.

C_{15}	**A:**	**`none of`** `{{`M_f`}}` **`occurs.`**
	G:	`{out_fail` **`and`** `!out_safe}` **`does not occur.`**

Assuming that `out_safe` is given as $out = 20$ and `out_fail` as $out \neq in\_correct + 2$:

$$\Phi(\{\{out\_fail \text{and}!out\_safe\}\}, \mathcal{F}) = (out = in\_correct + 2) \vee (out = 20)$$

In contrast to the already presented creations of fault-injection tasks for safety contracts, the template for delayed degrading safety mechanisms needs to be handled differently. Contract C_{16} uses the `perm` operator to indicate that *finally* the wrong output will be detected and a safe-state will be established.

C_{16}	**A:**	**`none of`** `{{`M_f`}}` **`occurs.`**
	G:	`{perm(output_fail` **`and`** `!output_safe)}` **`does not occur.`**

It may be sufficient for an abstract safety concept to specify that the wrong result is not permanently in the system, however, for a concrete implementation it is necessary to specify a time bound. This time bound, the *Fault Tolerant Time Interval* specifies the maximum time between the occurrence of the fault and the attainability of the safe-state. Hence, the requirement statement to be checked by the fault-injection analysis is, informally:

whenever failure occurs then (safestate or !failure) *occurs within the FTTI*

Since this requirement cannot be expressed with propositional logic, we need to integrate it directly in the behavioral model of the implementation as an observer (see Figure 4.21). Instead of the requirement we run the fault-injection analysis with a reference to the `fail` state, which shall not be reached:

$$G(\neg fail)$$

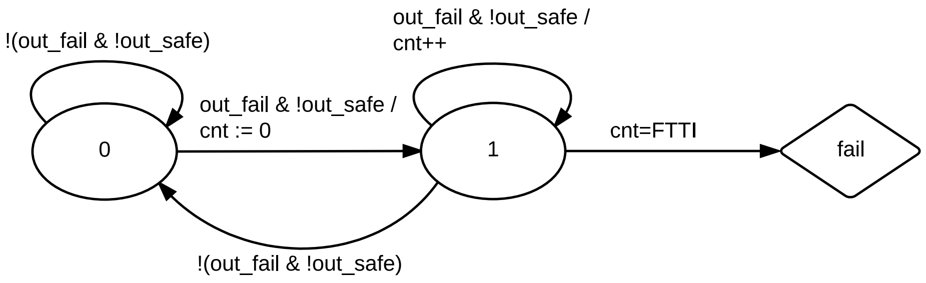


Figure 4.21 – Automaton for Perm with a given Bound

4.6 Conclusion

In this chapter a compositional safety view has been developed to reduce re-verification costs by enabling a determination of the area of the system affected by the incorporated changes. Furthermore, parts of a system can be re-used in other products. We have stated five requirements on the safety specification:

- *Requirements shall be represented as contracts.*
  We presented an approach to enable black-box safety specifications using safety contracts. Safety contracts use existing work, such as the formal safety pattern, but extend them e.g. with capabilities to specify multiple failure modes for one functional signal to represent also safe states of the system (see Section 4.3.1).

- *The concepts used in the development of functional safety concepts according to ISO 26262 shall be represented, as described by Figure 4.10.*
  We provided guidelines in the form of contract templates, which cover the needs of the functional safety concept of the ISO 26262. These contract templates can be used to specify the commonly used conceptual elements such as nondegrading safety mechanisms or delayed degrading safety mechanisms. Furthermore, we presented design guidelines how to develop a specification for a safety concept from scratch. This model can be used as a starting point for the impact analysis technique presented in Chapter 3.

- *The language to express the requirements shall provide means for abstraction.*
  In contrast to other safety specification approaches safety contracts provide a means of abstraction (see Section 4.3.3), thereby allowing the development of a system in a top-down manner, that is, to refine the specification by introducing the possible architecture of the subcomponents at a later point in time. We gained this property by developing a new fault abstraction technique for contracts using fault count ports and splitter components. In addition, this specification is closer to the requirements of current safety standards, and industrial needs often requiring the absence of single-point-failures without detailing the specific faults. Furthermore, it can be guaranteed that verification results gained in early stages of the development process do not get invalidated by refining the system.

- *The refinement property of the requirements shall be automatically analyzable.*
  The semantics of safety pattern are based on LTL, which allows to use the existing formulas for refinement as stated in literature (see Section 4.5.1)

- *The compliance of the requirements to implementations shall be automatically analyzable.*
  To be able to prove that an implementation correctly implements the fault propagation behavior specified by a safety contract, we based our approach on a fault injection technique. The relation between functional and safety specifications needed to be detailed to determine the functional deviation of a signal in the presence of a malfunction. Finally, we are able to locally analyze the fault propagation of a component while being able to use the result in the context of the whole system. This refers to the challenge that typically only malfunctions on the top level component are interesting for integrators or customers. Hence, the output malfunctions need to represent the "global truth" of the corresponding signals. Hence, malfunctions are described as deviations to a system without any present input or internal fault.

Using our specification and analysis technique it is now possible to judge the correctness of the output signals if all atomic components adhere to their specifications, and all refinement analyses of the components are successful. Hence, a virtual integration of the safety aspect is now possible. Impacts of changes in functional safety concepts are now identifiable. This means, if only parts of the safety specification have been changed, then only parts need to be re-verified. Other refinement and satisfaction analysis results are still valid without being re-calculated. Still, we need to evaluate if this approach is delivering the expected linear performance between the size of the change and the effort to re-verify the system.

CHAPTER 5

Evaluation

To evaluate the presented change impact analysis process and the developed formalism for safety concepts, two different techniques are used. To demonstrate the applicability of the proposed safety contracts and contract templates for safety concepts as well as the different analysis techniques, we provide an example in section 5.1, detailing the architecture, the safety requirements and also the functional requirements for a fail safe temperature sensor. The use-case is intentionally kept simple to still be able to provide a complete system specification. Afterwards the system is modified, and the proposed change management process instantiated and the required analyses and results are presented. Nevertheless, this example does not give any quantification of the benefit of this approach compared with the current state of the art. This would require many real-size systems, fully specified, and many different engineers involved, to get reliable data. Hence, we have chosen to develop a simulation environment (see Section 5.2) to generate architectures, perform changes and implement the engineers and automatic analyses as stochastic processes. This approach has even some advantages over the evaluation using real system data: It is possible to directly compare how the different parameters like size of the system, detection probabilities of the tests, the quality of the engineers or the number of requirements in a system influences the overall verification effort. Hence, the circumstances in which the approach performs best are precisely identifiable.

5.1 Example

In this example we step-by-step develop a functional and a safety specification for a temperature sensor. After this a change is introduced and the impact analysis process is used to determine in which region of the system the change is contained. This example shall apply the presented specification and analysis approaches. The example does not intend to quantify the approach or compare the development to other approaches. This

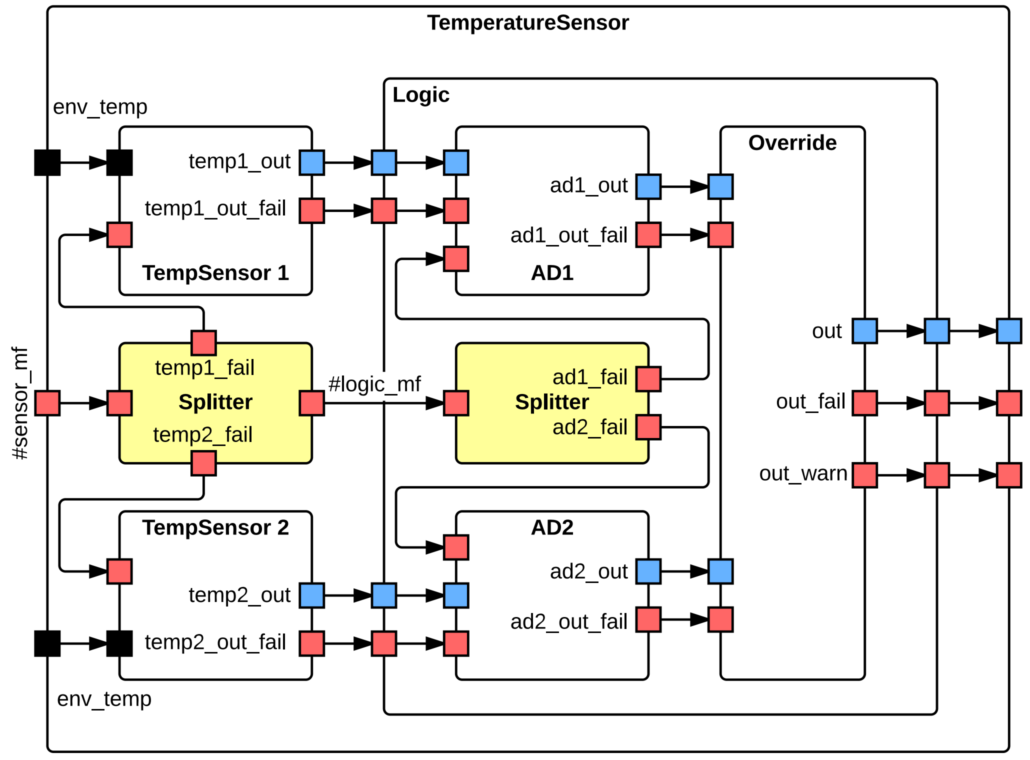


Figure 5.1 – Architecture of a temperature sensor required to be robust against single-points of failure

quantitative analysis is performed in section 5.2.

5.1.1 Specification and Design of the Initial System

We built a fail-safe temperature sensor in a top-down design process. The temperature sensor shall be used in the cooling system of the vehicle engine. The hazard analysis revealed that measuring a value which is lower than the actual value could cause a hazardous situation and hence a safety requirement exists, such that the sensor shall operate safely even in the presence of one fault. The safety contract describing that the system shall produce a correct, or at least safe, result in case of one fault, is:

C_{17}
```
A: | none of {{#internal_mf=2}} occurs.
G: | {temp_out_fail AND !temp_out_safe} does not occur.
```

The complete architecture of the sensor is depicted in Figure 5.1. The function of the temperature sensor shall be to deliver the value of the environmental temperature with an accuracy of +/- 1$°C$. The environmental temperature is expected to be between $-50°C$ and $200°C$:

C_{18}	**A:**	-50 $\leq env\_temp \leq$ 200
	G:	$((env\_temp - 1) \leq temp\_out \leq (env\_temp + 1))$

The safe-state of the temperature sensor is to output $200°C$, the maximum temperature. Note, that contract C_{17} and C_{18} alone do not specify the behavior correctly. Following these requirements, a behavior that constantly outputs $200°C$ would be correct and it would be safe, but it is not desirable to build a device that does not measure the temperature correctly. To exclude the unintended permanently safe-state behavior we need to state an additional safety contract, which requires a correct output if no malfunction is present in the component:

C_{19}	**A:**	**none of** {{#internal_mf=1}} **occurs.**
	G:	{temp_out_fail} **does not occur.**

These are the requirements that a company building temperature sensors may have received from an OEM that develops the whole cooling system for the engine. Still, if they need to develop a new system because no existing sensor fulfills these requirements, they need to refine the specification and the design.

A realization of these requirements on the next abstraction level can be a design with two simple but redundant analog temperature sensors in addition to a logic which provides the AD-conversion and a voting mechanism. The functional requirement of the temperature sensors is to output the environmental temperature in the range of -50 to 200 as a linear voltage between 0 and 5V with an accuracy of $1°C$:

C_{20}	**A:**	-50 $\leq$ env_temp $\leq$ 200
	G:	$\frac{(env\_temp-1)+51}{51} \leq$ temp1_out $\leq \frac{(env\_temp+1)+51}{51}$

The functional contract for `TempSensor2` is similarly:

C_{21}	**A:**	-50 $\leq$ env_temp $\leq$ 200
	G:	$\frac{(env\_temp-1)+51}{51} \leq$ temp2_out $\leq \frac{(env\_temp+1)+51}{51}$

The sensors themselves do not provide any safety mechanisms and hence can fail immediately as a result of a single internal failure. Therefore, the safety contract for `TempSensor1` is:

C_{22}	**A:**	{temp1_fail} **does not occur.**
	G:	{temp1_out_fail} **does not occur.**

`TempSensor2` is specified in an identical manner:

C_{23}	**A:**	{temp2_fail} **does not occur.**
	G:	{temp2_out_fail} **does not occur.**

The malfunction `temp1_fail` is defined as a generic internal failure of the component that leads to a random measurement and finally may cause a wrong value (`temp1_out_fail`) at the output (see Table 5.1). Note, that there are no input malfunctions defined for the environmental temperature, since this value is correct by definition. Hence, we can replace `env_temp_correct` with the environmental temperature `env_temp`. This behavior can be observed on all systems, that are specified in a way that the inputs of the system are expected to be correct. This allows to always find a functional representation for the correct signals at the input of components. Still, this is not the case for all systems and not necessary for the approach, but since it increases the comprehensibility the example has been chosen to provide this property.

Malfunction	**Description**	Formalization
temp1_fail	The sensing function does not work properly, no guarantee of the measures value within the temperature range can be given	$temp1\_out \in [-50, 200]$
temp2_fail	The sensing function does not work properly, no guarantee of the measures value within the temperature range can be given	$temp2\_out \in [-50, 200]$
temp1_out_fail	`temp1_out` does not reflect the environmental temperature within is accuracy range	$(temp1\_out \leq \frac{(env\_temp\_correct-1)+51}{51}) \vee (temp1\_out \geq \frac{(env\_temp\_correct+1)+51}{51}) \Leftrightarrow (temp1\_out \leq \frac{(env\_temp-1)+51}{51}) \vee (temp1\_out \geq \frac{(env\_temp+1)+51}{51})$
temp2_out_fail	`temp2_out` does not reflect the enironmental temperature within is accuracy range	$(temp2\_out \leq \frac{(env\_temp\_correct-1)+51}{51}) \vee (temp2\_out \geq \frac{(env\_temp\_correct+1)+51}{51}) \Leftrightarrow (temp2\_out \leq \frac{(env\_temp-1)+51}{51}) \vee (temp2\_out \geq \frac{(env\_temp+1)+51}{51})$

Malfunction	**Description**	Formalization
ad1_fail	The AD-conversion is not performed correctly, no guarantee of the output can be given, except of staying in the defined limits of the output voltage	$(ad1\_out \neq ((temp1\_out \cdot 51) - 51)) \wedge (ad1\_out \in [0,5])$
ad2_fail	The AD-conversion is not performed correctly, no guarantee of the output can be given, except of staying in the defined limits of the output voltage	$(ad2\_out \neq ((temp2\_out \cdot 51) - 51)) \wedge (ad2\_out \in [0,5])$
ad1_out_fail	The output of the AD-conversion does not reflect the correct value within its accuracy range	$ad1\_out \neq ((temp1\_out\_correct \cdot 51) - 51) \Leftrightarrow (ad1\_out \leq env\_temp - 1) \vee (ad1\_out \geq env\_temp + 1)$
ad2_out_fail	The output of the AD-conversion does not reflect the correct value within its accuracy range	$ad2\_out \neq ((temp2\_out\_correct \cdot 51) - 51) \Leftrightarrow (ad2\_out \leq env\_temp - 1) \vee (ad2\_out \geq env\_temp + 1)$
out_fail	The temperature sensor does not output to correct environmental temperature within its defined accuracy range	$(out \leq (env\_temp - 1)) \vee (out \geq (env\_temp + 1)) \Leftrightarrow out \neq ad1\_correct$
out_safe	The safe-state is the maximum temperature the sensor is capable to measure	$out = 200$

Table 5.1 – Malfunctions and functional requirements of the automatic light manager

The logic component shall react to possible malfunctions in the temperature sensors. As the internal structure has not yet been decided, the requirement states that even if one of the inputs receives a wrong value or an internal fault occurs the result should at least be safe (*temp_out_safe*). This requirement is expressed using a safety contract:

C_{24}

```
A: none of {
   {temp1_out_fail, temp2_out_fail},
   {#logic_mf=1, temp1_out_fail},
   {#logic_mf=1, temp2_out_fail},
   {#logic_mf=2}
   } occurs.
G: {temp_out_fail AND !temp_out_safe} does not occur.
```

The functional requirement of the logic component is to convert the signal to a digital representation in $°C$ if both input signals are identical, otherwise output the maximum value 200.

C_{25}

A: $(0 \leq temp1\_out \leq 5) \land (0 \leq temp2\_out \leq 5)$
G: $(temp1\_out = temp2\_out) \to temp\_out = ((temp1\_out \cdot 51) - 51) \land (temp1\_out! = temp2\_out) \to temp\_out = 200$

Note, that in this refinement step also the splitter component is automatically generated to split up the number of total malfunctions in the system onto the three components on this abstraction level.

Again to force correct behavior in case of no malfunction, we state the following safety contract:

C_{26}

```
A: none of {{#logic_mf=1},{temp1_out_fail},{temp2_out_fail}
   } occurs.
G: {temp_out_fail} does not occur.
```

Compared to contract C_{19} also the internal malfunctions are considered here, which were not present at the `TempSensor1` component.

In the refinement step of the `Logic` component two independent analog/digital converters are used to digitize the temperature signal. Both converters do not provide safety mechanisms and fail immediately if the input is incorrect or the generic internal malfunction `ad1_fail` occurs.

C_{27}

```
A: none of {{ad1_fail},{temp1_out_fail}} occurs.
G: {ad1_out_fail} does not occur.
```

The functional conversion is described as:

C_{28}

A: $(0 \leq temp1\_out \leq 5) \land (0 \leq temp2\_out \leq 5)$
G: $ad1\_out = ((temp1\_out \cdot 51) - 51)$

Ad2 is specified similarly:

C_{29}	**A:**	`none of {{ad1_fail},{temp1_out_fail}} occurs.`
	G:	`{ad1_out_fail} does not occur.`

and

C_{30}	**A:**	$(0 \leq temp1\_out \leq 5) \wedge (0 \leq temp2\_out \leq 5)$
	G:	$ad1\_out = ((temp1\_out \cdot 51) - 51)$

On the AD-converters the difference between the internal malfunctions and the output malfunction can be seen (Table 5.1). The internal malfunction `ad1_fail` directly violates the functional requirement of the AD-converter. The component is defective and it cannot perform as specified anymore. Hence, a functional deviation can be stated. Since the output malfunctions `ad1_out_fail` also considers input malfunctions, and hence describes the *global correctness* of the output port, the functional description needs to refer to the correct input signal `temp1_out_correct`. In this case, the correct signal can be determined, since the top level component does not suffer from input malfunction. Hence, the output malfunction at the ADC indicates, that the value does not reflect correctly the environmental temperature with an accuracy of $1°C$. Again, the second converter is specified similarly.

The signals of the AD-converters are processed by an `Overwrite` component which compares the results and sets the safe-state if the values differ. The override component is not expected to fail in the lifetime of the device. This is a common assumption in voting architectures to leave over a very small functionality to a component that is formally verified and produced in a more robust way than the rest of the system or replaced in a regular manner during service intervals (Bozzano & Villafiorita, 2011). The safety contract therefore considers only input faults:

C_{31}	**A:**	`none of {{ad1_out_fail, ad2_out_fail}} occurs.`
	G:	`{temp_out_fail AND !temp_out_safe} does not occur.`

and:

C_{32}	**A:**	`none of {{ad1_out_fail}, {ad2_out_fail}} occurs.`
	G:	`{temp_out_fail} does not occur.`

the functional contract for the `Override` component is given as:

C_{33}	**A:**	`-50` $\leq$ `env_temp` $\leq$ `200`
	G:	$(temp1\_out = temp2\_out) \rightarrow temp\_out = ((temp1\_out \cdot 51) - 51) \wedge (temp1\_out! = temp2\_out) \rightarrow temp\_out = 200$

These contracts provide us an additional definition of the `out_fail` malfunction as $out \neq ad1\_out\_correct$. Since the output port does belong in an identical manner to multiple components (they are connected using delegation connectors) it is not surprising, that multiple definitions exist. Nevertheless, these are equivalent!

$$\begin{array}{rc} & out \neq ad1\_out\_correct \\ \Leftrightarrow & out = \neg ad1\_out\_correct \\ \Leftrightarrow & out = ad1\_out\_fail \\ \Leftrightarrow & out = (ad1\_out \leq env\_temp - 1) \vee (ad1\_out \geq env\_temp + 1) \\ \Leftrightarrow & (out \leq env\_temp - 1) \vee (out \geq env\_temp + 1) \\ \Leftrightarrow & out\_fail \end{array}$$

The `Splitter` component in the `Logic` component, the connection of the splitter to the faults of the subcomponents as well as the contract for the splitter component are, again, generated automatically and do not need to be specified separately.

Refinement can now be checked on both levels of abstraction.

$$(C_7 \otimes C_8 \otimes C_9) \preceq C_6$$

as well as

$$(C_{10} \otimes C_{11} \otimes C_{12}) \preceq C_9$$

For an exemplary performed refinement analysis refer to Section 6.2.

5.1.2 Changing the System

Similar to the design steps in the previous section, we assume a fictional but realistic event that causes the design to change. While performing field tests with the build device, it has been observed, that the description of the failure modes of the AD-converters were not complete. In case of a physical shock, both AD-converters may fail and the device delivers constantly $-50°C$ as the measured temperature. The device is not operating inside its requirements, since it is vulnerable to a single fault. The design needs to be updated with respect to the newly discovered malfunction and additional changes are necessary to provide a safe system again. Hence, the additional malfunction needs to be added to the component as a new safety port, and the safety requirement of the AD converters is changed to reflect the new malfunction to:

C_{34}	A:	**none of {{ad1_fail},{temp1_out_fail},{phys_dem}} occurs.**
	G:	**{ad1_out_fail} does not occur.**

and for `AD2`:

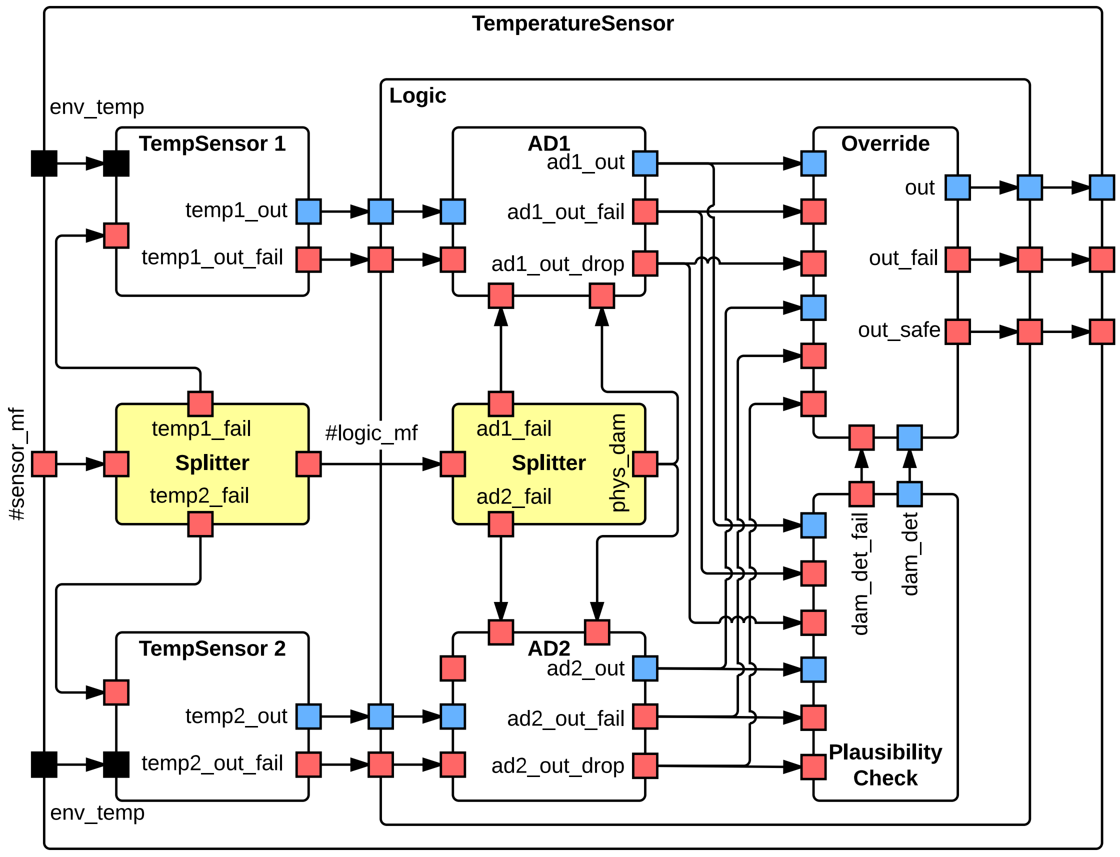


Figure 5.2 – A previously unknown common cause faults damaging both A/D converters has been identified. To still maintain the safety properties a consistency check has been integrated.

C_{35}	A:	`none of {{ad2_fail},{temp2_out_fail},{phys_dem}} occurs.`
	G:	`{ad2_out_fail} does not occur.`

These changes represent the initial set of changes. According to Table 3.6 the verification activities depicted in Table 5.2 have to be re-run. The results of the activities are also presented in this table.

Surprisingly many verification activities are still successful. Although the specification has been changed, which respects now the newly identified malfunction, it is still compliant with the implementation. Since the fault injection analysis, which is used to determine if the behavior under malfunctions is as specified, does rely on the completeness of the identified malfunctions, the shock malfunction has not been tested. Still, the safety manuals describing the failure modes of the components are typically expected to be complete. Since the component and the specification have been modified (the malfunction ports have been added to the component), the interface checks are all successful.

Verification Activity	Description	Result
$V_r(C_{24}, \{C_{34}, C_{35}, C_{31}, C_{32}\})$	Refinement Analysis to verify if contracts of subcomponents correctly refine the specification of the logic component	failed
$V_s(C_{34}, \overleftarrow{I}(\texttt{AD1}))$	Satisfaction analysis to check the compliance of the implementation of `AD1` to its specification	successful
$V_s(C_{35}, \overleftarrow{I}(\texttt{AD2}))$	Satisfaction analysis to check the compliance of the implementation of `AD2` to its specification	successful
$V_i(C_{34}, \texttt{AD1})$	Interface analysis to check compliance of ports and data-types between the safety specification of `AD1` and its interface represented by the component	successful
$V_i(\texttt{AD1}, \overleftarrow{I}(\texttt{AD1}))$	Interface analysis to check compliance of the interface of `AD1`, given by its component, and the implementation of `AD1`	successful
$V_i(C_{35}, \texttt{AD2})$	Interface analysis to check compliance of ports and data-types between the safety specification of `AD2` and its interface represented by the component	successful
$V_i(\texttt{AD2}, \overleftarrow{I}(\texttt{AD2}))$	Interface analysis to check compliance of the interface of `AD2`, given by its component, and the implementation of `AD2`	successful

Table 5.2 – Verification Activities that have to be Re-Run after the initial change

The failed refinement analysis indicates that the requirement of the `Logic` component, to be safe in case of one malfunction, is not fulfilled by the requirements assigned to the subcomponents. This is not surprising, since a common cause failure (`phys_dem`) causes both AD-converters to fail and the override is not able to detect the malfunction and outputs a wrong value.

As the table for the identification of compensation candidates (Table 3.8) indicates, we could change the contract of the logic component or change the design inside. Since we want to limit the change propagation as much as possible, we will add an additional component inside the `Logic` component to handle the additional malfunction. The new `Plausibility Check` component detects the physical damage by monitoring the temperature over time. The plausibility check is based on the assumption that a huge drop of the temperature is not possible between two measurement intervals. Hence, both sensors are monitored and a detection signal is sent to the override component, that a sudden drop has been detected.

To be able to detect the sudden temperature drop by the plausibility check, the malfunctions on the AD-converters need to be made available separately by adding a new safety port. That way it is possible to refer separately to the different malfunctions. Note, the malfunctions are still bound to one functional signal, and all components that use the functional signal are prone to all malfunctions associated to the signal. Still, the detection of the malfunction may be handled by a distinct set of components. The new safety contract for the AD-converters is:

C_{36}	`A:`	`none of {{phys_dem}} occurs.`
	`G:`	`{ad1_out_drop} does not occur.`

and for AD2:

C_{37}	`A:`	`none of {{phys_dem}} occurs.`
	`G:`	`{ad2_out_drop} does not occur.`

Contract C_{34} remains valid, since the physical damage of the device also causes a wrong signal. But the signal malfunction `ad1_out_drop` is only caused by the physical damage internal malfunction. This malfunction indicates that the signal drops to $-50°C$ and keeps this value. The other malfunctions are of a random nature and may change its value at all times. Hence, the behavior on the output are different.

The complete architecture that integrates the plausibility check is depicted in Figure 5.2. The functional requirement for the plausibility check is given as:

	`A:`	$(-50 \leq ad1\_out \leq 200) \wedge (-50 \leq ad2\_out \leq 200)$
C_{38}	`G:`	$((ad1\_out' < 2) \wedge (ad2\_out' < 2) \wedge (ad1\_out' == ad2\_out')) \rightarrow$
		$(phys\_det == 1)$

It checks if the temperature drops faster than physically possible. The safety contract is given as:

	`A:`	`none of {{ad1_out_fail AND !ad1_out_drop,`
C_{39}		`ad2_out_fail AND !ad2_out_drop}} occurs.`
	`G:`	`{dem_det_fail} does not occur.`

The safety contract states when the detection signal might be wrong. The detection of the physical damage is only incorrect if there is a malfunction in the ADC, either on its inputs or internally, that on both channels the temperature is dropping fast at the same rate but no physical damage has happened. The gradient check is only correctly working if we assume that the operating temperature of the device is higher than $-48°C$, since otherwise the drop would be in the physically possible gradient range.

Also the override component needs to be modified. The additional ports need to be added and the functional as well as the safety contract needs to be updated:

C_{40}	**A**:	**none of** `{{ad1_out_fail, ad2_out_fail},{ad1_out_drop, ad2_out_drop, dem_det_fail}}` **occurs.**
	G:	`{temp_out_fail AND !temp_out_warn}` **does not occur.**

It can be observed, that the cut-set of {ad1_out_drop, ad2_out_drop} alone does not lead to a violation of the guarantee. This is crucial, since `ad1_out_drop` and `ad2_out_drop` would be caused by one single malfunction, the physical damage. Only if there is also a problem with the detection signal can all three malfunctions together lead to a wrong output of the temperature sensor that is not safe. In contrast to that, the correct behavior is influenced by all the single malfunctions, hence:

C_{41}	**A**:	**none of** `{{ad1_out_fail}, {ad2_out_fail},{dam_det_fail}, {ad1_out_drop},{ad2_out_drop}}` **occurs.**
	G:	`{temp_out_fail}` **does not occur.**

Also the functional contract of the override component needs to be adapted with respect to the new detection port:

C_{42}	**A**:	`TRUE`
	G:	$(((ad1\_out == ad2\_out) \wedge dem\_det == 0) \rightarrow (out == ad1\_out)) \wedge (((ad1\_out! = ad2\_out) \vee dem\_det == 1) \rightarrow out = 200)$

Now, a couple of verification activities need to be executed. These activities and their results are displayed in Table 5.3.

Verification Activity	**Description**	**Result**
$V_r(C_{24}, \{C_{34}, C_{35}, C_{36}, C_{37}, C_{40}, C_{41} C_{39}\})$	Refinement analysis to verify if contracts of subcomponents correctly refine the specification of the logic component	successful
$V_s(C_{36}, \overleftarrow{I}(\texttt{AD1}))$	Satisfaction analysis to check the compliance of the implementation of `AD1` to its newly added specification	successful
$V_s(C_{37}, \overleftarrow{I}(\texttt{AD2}))$	Satisfaction analysis to check the compliance of the implementation of `AD2` to its newly added specification	successful

Verification Activity	Description	Result
$V_i(C_{36}, \texttt{AD1})$	Interface analysis to check compliance of ports and data-types between the safety specification of `AD1` and its interface represented by the component	successful
$V_i(\texttt{AD1}, \overleftarrow{I}(\texttt{AD1}))$	Interface analysis to check compliance of the interface of `AD1`, given by its component, and the implementation of `AD1`	successful
$V_i(C_{37}, \texttt{AD2})$	Interface analysis to check compliance of ports and data-types between the safety specification of `AD2` and its interface represented by the component	successful
$V_i(\texttt{AD2}, \overleftarrow{I}(\texttt{AD2}))$	Interface analysis to check compliance of the interface of `AD2`, given by its component, and the implementation of `AD2`	successful
$V_i(C_{40}, \texttt{Override})$	Interface analysis to check compliance of ports and data-types between the safety specification of `Override` and its interface represented by the component	successful
$V_i(C_{41}, \texttt{Override})$	Interface analysis to check compliance of ports and data-types between the safety specification of `Override` and its interface represented by the component	successful
$V_i(\texttt{Override}, \overleftarrow{I}(\texttt{Override}))$	Interface analysis to check compliance of the interface of `Override`, given by its component, and the implementation of `Override`	successful
$V_i(C_{39}, \texttt{PlausibilityCheck})$	Interface analysis to check compliance of ports and data-types between the safety specification of the `PlausibilityCheck` and its interface represented by the component	successful
$V_i(\overleftarrow{I}(\texttt{PlausibilityCheck}), \texttt{PlausibilityCheck})$	Interface analysis to check compliance of the interface of `PlausibilityCheck`, given by its component, and the implementation of `PlausibilityCheck`	successful

Table 5.3 – Verification Activities that have to be Re-Run after the second change

Since all verification activities are successful, the change is contained in the logic component, that is, there are no other effects of this change outside this component and all other stated requirements are still valid. In this example the savings are rather small, since only the temperature sensors do not need to be re-verified. But in systems where the ratio of the changes system parts to the not changes system parts is small, the savings grow rapidly. To measure this effect in a quantitative way, a simulation of the approach has been developed, results of which are described in Section 5.2.

Nevertheless, none of the satisfaction analyses have been considered in detail yet. Exemplary for the other components we are going to analyze if the `Override` component correctly implements its specifications.

The implementation of the `Override` component is depicted in Figure 5.3 in MATLAB Stateflow syntax. It consists of two states `override-active` and `override-inactive`. In the inactive state, both input signals match and no physical damage has been detected by the plausibility check. Hence, the identical value of the AD-converters is put on the output. If there is either a deviation in the signal of the AD-converters or the fault detection signal of the plausibility check is active, then the override component switches to the active state and the value 200 is returned.

Also the implementations of the input malfunctions are depicted in the figure. These implementations have the correct value as an input (e.g., ad1_out_correct) and may enable a malfunction to deviate from this signal. The override component is operating on the output of the additionally generated components that enable the processing of input malfunctions.

Two safety contracts (C_{40}, C_{41}) are assigned to the `Override` component and shall be checked against the implementation individually. Contract C_{40} requests the signal to be safe or correct in case of one malfunction in the system or its inputs. Hence, the functional representation of the guarantee of the safety contract is:

$$r_{f1} = \text{``}out = ad1\_correct \vee out = 200''$$

Looking at the implementation of the input malfunctions in Figure 5.3, the usage of `ad1_correct` is possible in the requirement, since this is the name of the input signal before the deviation is performed.

Running the fault injection analysis, we obtain the following results:

$$\begin{aligned} &\text{MBSA}(r_{f1}, \overleftarrow{I}(\texttt{Override}), \\ \{&\texttt{ad1\_out\_fail}, \texttt{ad2\_out\_fail}, \texttt{ad1\_out\_drop}, \texttt{ad2\_out\_drop}, \texttt{dem\_det\_fail}\}) \\ &= \{\{ad1\_out\_fail, ad2\_out\_fail\}, \\ &\{ad1\_out\_drop, ad2\_out\_drop, dem\_det\_fail\}\} \end{aligned}$$

The resulting cut-sets are identical to the assumption of contract C_{42}, the implementation is compliant with the contract.

Similarly, we analyze the second safety contract C_{41}. The functional representation of the guaranteed `out_fail` that shall not occur is:

$$r_{f2} = \text{“}out = ad1\_correct''$$

The cut-sets are again identical.

$$\begin{aligned}
&\text{MBSA}(r_{f1}, \overleftarrow{I}(\texttt{Override}),\\
&\{\texttt{ad1\_out\_fail}, \texttt{ad2\_out\_fail}, \texttt{ad1\_out\_drop}, \texttt{ad2\_out\_drop}, \texttt{dem\_det\_fail}\})\\
&= \{\{ad1\_out\_fail\}, \{ad2\_out\_fail\}, \{dam\_det\_fail\},\\
&\{ad1\_out\_drop\}, \{ad2\_out\_drop\}\}
\end{aligned}$$

Hence, the compliance of the implementation to the stated safety contracts has been shown.

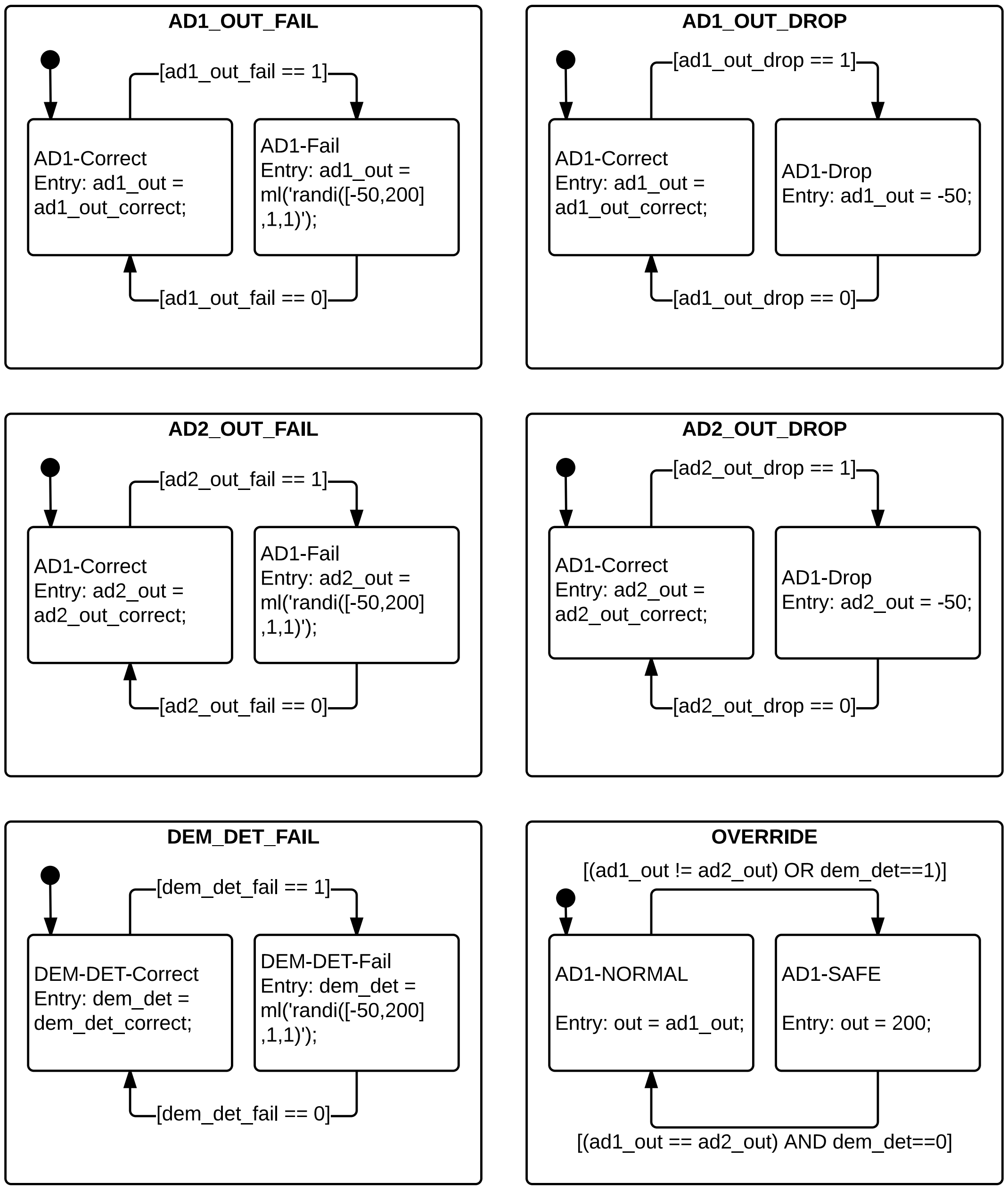


Figure 5.3 – Matlab Stateflow implementation of the OVERRIDE Component (bottom right). To represent the injected input faults additional components have been introduced that handle the injection of faults at the input. The input malfunctions are activated by the corresponding malfunction ports.

5.2 Simulative Evaluation

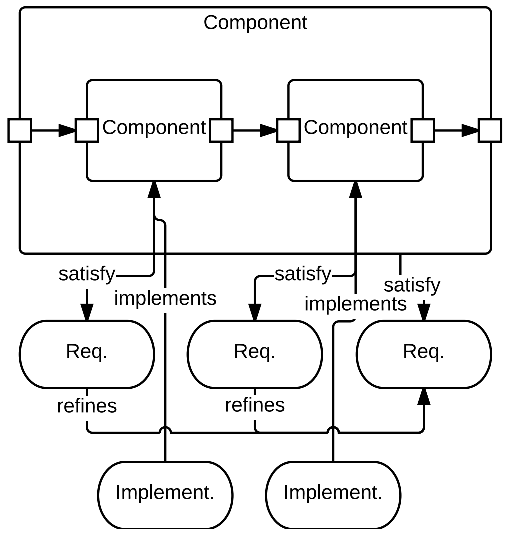
While the example gave an overview of the capabilities of the developed safety specification and how it is used in a top down development process, it was not able to give an estimate of how much effort can be saved while using a contract-based change impact analysis compared to the currently used “re-verify all” approach. Hence, in this section we are investigating the effectiveness of the presented modular verification approach by means of simulation. We compare both approaches across generated systems. We are aware of the situation, that “everything” can under some circumstances be limited to a subsystem or module that is sufficiently independent from the rest of the system. In that sense the generated system refers to these modules. Furthermore, we want to evaluate how the accuracy of tests and analyses, and therefore also the needed degree of formalism affects the effectiveness of both approaches. We simulate multiple change and system sizes to further narrow down the situations in which either approach performs optimally.

5.2.1 Simulative Setup for Comparison

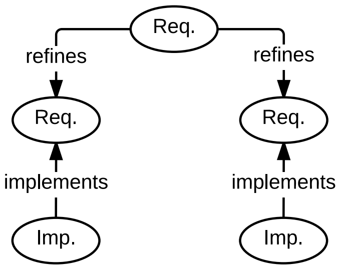
The system design guidelines of the presented specification approach (see Section 4.4) assume a view consisting of requirements that are linked to components. These components may have an attached implementation which satisfy the requirements (an example is depicted in Figure 5.4a). In a contract-based design scenario these components are connected via ports and the requirements specify the behavior of a component in a black-box manner based on the existing ports. While measuring the performance of both impact analysis techniques, we focus on the refinement and satisfaction analyses, since the interface checks can easily be automated and consume no significant computational effort. Hence, for the simulation we have chosen a graph based representation (see Figure 5.4b) of the system, that is limited to requirements and implementations. Still, all refinement and satisfaction analyses are represented by the edges in the graph. As indicated by Figure 3.17 the components are only connected to interface analyses and can therefore be neglected in the simulation. In Figure 5.4b, requirements are linked to their refined requirements and also to the implementations that shall implement the requirement. These links represent a verification or validation activity aiming at guaranteeing a correct refinement or satisfaction relation. A refinement analysis is in this case represented by the top level requirement and all directly refined ones.

In contrast, the current state of the art approach for handling changes in a module is to let an engineer fix the system elements which are likely to be affected by the initial change, and then run all tests and analyses again. If problems have been detected, the engineer will change the corresponding components. This procedure, will be iterated until no further problems have been detected by the performed test and hence all V&V tasks are successfully re-run.

Ideally, we would like to compare the effectiveness of the contract-based change management process with the more standard change management process of complete re-validation (within the considered module) on a large number of real-world architectures in which changes have to be handled with an appropriate process. However, as we do not



(a) Component View



(b) Simulation View

Figure 5.4 – Requirements and implementations are attached to connected components 5.4a and the graph-based representation used in the simulation, where components are neglected 5.4b

have access to a sufficiently large data base, we use generated architectures. Also, instead of letting an engineer select which system elements need to be changed, if inconsistencies are detected, we define all necessary changes in advance in the architecture which have to be identified with either impact analysis process. Such random generation acts as a proxy to estimate the effectiveness on a real world architecture. A further need for this pre-selection of changes is to ensure that in both approaches the human engineer (which is then replaced by the simulation) in both cases takes identical decisions which elements of the architecture are changed when confronted with the same possible set of elements to change. Therefore, to generate architectures and potential changes as realistic as possible, we tried to incorporate experience from real case studies. We identified that a requirement is refined into one to six subrequirements where three and four are the most frequent cases. Hence, we generated the requirements structure based on the normal distribution given in Figure 5.5.

Furthermore the implementations are generated based on the subrequirements of the second to last level R. The leave notes $l(r)$ of the elements $r \in R$ are distributed across 1 to $|l(r)|$ implementations in a uniform distributed way.

Based on these numbers we generate architectures, depicted in Figure 5.6, which are similar to Figure 3.17, in which only the atomic requirements are implemented. In the figure, the requirements and implementations that need to be changed are highlighted in green. The set of necessary changes is generated randomly.

We start the algorithms for both approaches on these systems. We assign a detection accuracy in form of the probability P_{acc} to the performed refinement and satisfaction analyses. The implementations of both approaches for the simulation are depicted

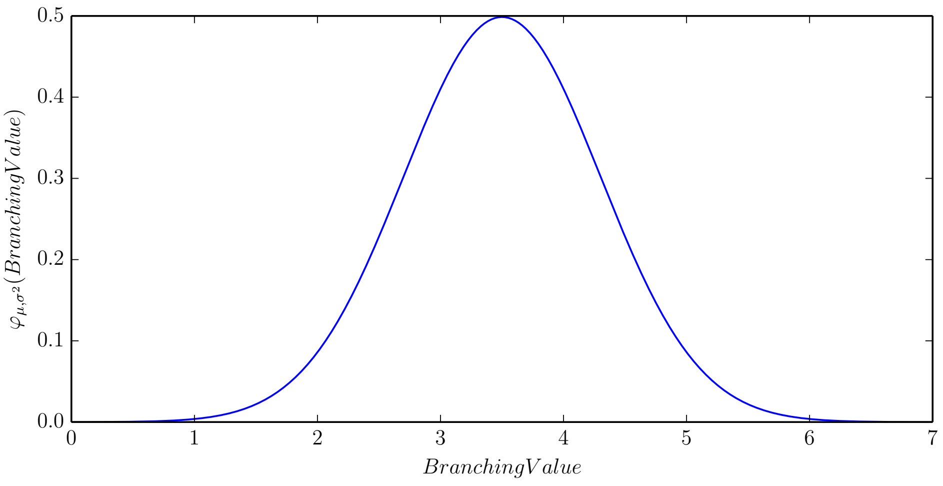


Figure 5.5 – Probability Density Diestribution for the Branching Factor of the generation of the Requirements structure. $\mu = 3,5$ and $\sigma = 0,8$

in Algorithm 3 for the contract-based approach and in Algorithm 4 for the standard approach. The simulation code varies only slightly from the presented impact analysis process described in Section 3.3.4 as we assume that the engineer directly fixes the inconsistent requirements or implementations as soon as the analysis has detected the problem. Although this is a rather optimistic assumption, this affects both approaches in the same way, thereby not favoring any one algorithm in particular.

The recursive contract-based realization depicted in Algorithm 3 is based upon a list of already processed system elements Φ. The procedure `CBA_SIM(G,n)` is at the beginning called with `n` as the initial change. It is the principle of the algorithm to count the number of necessary analyses for each changed element, and then identify, based on the test accuracy P_{acc} if the analyses detect the necessary changes. If a change was detected the procedure is called again on the changed node(s). Note, that C, the set of necessary changes, is used to simulate the decision of the engineer, which would normally select the components that need to be modified. All types of connected links need to be processed separately. Therefore, lines $10 - 18$ handle the refinement links pointing from a requirement to a more concrete requirements, lines $20 - 28$ handle the refinement links pointing to a more abstract requirement down to n (also pointing to the siblings of n). One refinement link is meant here as a link from one requirement to one or multiple refining ones. In lines $30 - 38$ all satisfy links are counted individually. Since we are assuming that the change we made to a component is already correct at the first modification, we do not modify an element twice; this exit is handled in lines $4 - 8$.

The state of the art approach is depicted in Algorithm 4.

At first, all system elements from the necessary change will be detected with probability P_{eng}. This probability corresponds to the skill of the engineer in expecting changes correctly and correcting them. In this phase (lines $4 - 9$) no analyses are performed.

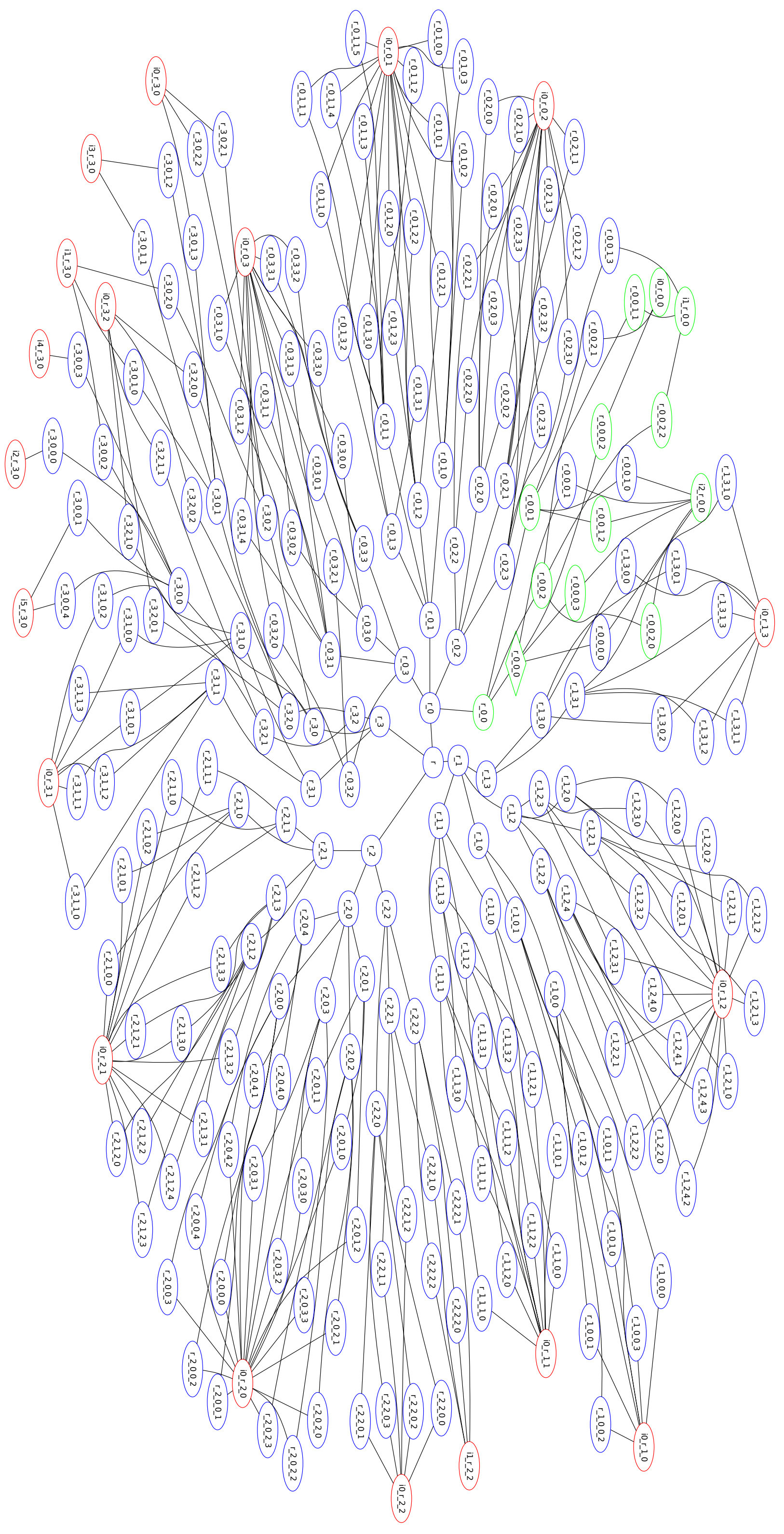


Figure 5.6 – A generated architecture with 231 requirements (blue), 22 implementation (red). Thirteen elements have been selected to be changed (green) which is about 3% of the whole architecture. The initial change is marked with a diamond shape. The simulation results depicted in Figure 5.8 are based on this architecture.

Algorithm 3 Contract-based realization for the simulation

```
Require: $C$: set of nodes $n$ in the necessary change
$\Phi = \emptyset$
numAna = 0
procedure CBA_SIM($G$,$n$)
    if $n \in \Phi$ then
        return
    else
        $\Phi = \Phi \cup n$
    end if

    $\Delta = \bigcup_{p \in \texttt{G.nodes()}} : \texttt{isRefineDown}(n, p) \wedge (n, p) \in \texttt{G.edges()}$
    if $\Delta \neq \emptyset$ then
        numAna++
        if $\exists \delta \in \Delta : \delta \in C$ then
            if random() $\leq P_{acc}$ then
                cba_sim($G$,$\delta$)
            end if
        end if
    end if

    $\Upsilon = \bigcup_{p \in \texttt{G.nodes()}} : \texttt{isRefineUp}(p, n) \wedge (p, n) \in \texttt{G.edges()}$
    if $\Upsilon \neq \emptyset$ then
        numAna++
        if $\exists \upsilon \in \Upsilon : \upsilon \in C$ then
            if random() $\leq P_{acc}$ then
                cba_sim($G$,$\upsilon$)
            end if
        end if
    end if

    $\Sigma = \bigcup_{p \in \texttt{G.nodes()}} : \texttt{isSatisfy}(p, n) \wedge (p, n) \in \texttt{G.edges()}$
    for $\sigma \in \Sigma$ do
        numAna++
        if $\exists \sigma \in C$ then
            if random() $\leq P_{acc}$ then
                cba_sim($G$,$\sigma$)
            end if
        end if
    end for

end procedure
```

Algorithm 4 State-of-the-art approach in simulation

```
Require: $C$: set of necessary changes
 1: $\Phi = \emptyset$
 2: numAna = 0
 3: procedure SOTA_SIM($G$,$i$)
 4:     for $n \in C$ do
 5:         if random() $\leq P_{eng}$ then
 6:             $\Phi = \Phi \cup n$
 7:         end if
 8:     end for
 9:
10:     repeat
11:         $\Phi_{init} = \Phi$
12:         for $e \in G.edges()$ do
13:             if isDeriveUp($e$) || isDeriveDown($e$) then
14:                 numAna ++
15:                 $E$ = getParallelEdges($e$)
16:                 if $\exists (n_1, n_2) \in E : n_1 \in C \backslash \Phi \vee n_2 \in C \backslash \Phi$ then
17:                     if random() $\leq P_{acc}$ then
18:                         $\Phi = \Phi \cup E$
19:                     end if
20:                 end if
21:             end if
22:             if isSatisfy($e$) then
23:                 numAna ++
24:                 if $\exists e = (n_1, n_2) : n_1 \in C \backslash \Phi \vee n_2 \in C \backslash \Phi$ then
25:                     if random() $\leq P_{acc}$ then
26:                         $\Phi = \Phi \cup e$
27:                     end if
28:                 end if
29:             end if
30:         end for
31:     until $\Phi_{init} == \Phi$
32: end procedure
```

Then, all edges in the system graph are inspected and the analyses are counted. This corresponds to the execution of all analyses and tests. Again, the set of necessary changes C identifies the system elements that need to be detected by the verification or validation activity using probability P_{acc}. For refineUp and refineDown edges all parallel edged, belonging to the same refinement analysis are processed together (lines 13 – 21). The satisfy links are counted each (lines 22 – 29). If changes have been identified during the analysis of all edges of the system, the whole system is analyzed again (line 31).

The running time of both algorithms has an upper bound of $\mathcal{O}(|G.nodes()|)$, the number of node, i.e. the number of requirements in the system. For Algorithm 3 this bound can easily be identified, since the set Φ ensures that the algorithm is called on each node only once. For Algorithm 4 this is less obvious. The running time depends on the detection probabilities P_{eng} and P_{acc}. If $P_{eng} = 1$ the running time is only bound by the size of C. Small values for P_{eng} do not influence the running time significantly as long as $P_{acc} > 0$ holds. In that case the worst case probability for each run of the loop to terminate is $|C| \cdot (1 - P_{acc})^2$. Although this probability is decreasing with each run of the loop this value is constant in the size of the number of requirements. Hence, the size of the loop determine the upper bound, resulting in `SOTA_SIM(G,i)` $= \mathcal{O}(G.nodes())$. The loop probability directly infers that the algorithm does not terminate for $P_{acc} = 0$. Hence, P_{acc} is simulated between 1% and 96%. Still, the simulation running time is not the important value that needs to be evaluated. The simulation target is to identify how many changes in C get identified with varying test accuracy P_{acc}.

To be able to compare both algorithms detection rate in relation to the individual accuracy of the refinement analyzes, we need a notion of effort. The effort to re-verify a system stems from the amount of verification activities as well as the used accuracy. A less accurate test (i.e., a test with a low detection probability P_{acc}) should have lower associated effort compared with a test with high accuracy. To account for such effects, we associate to a test having a probability of detection P_{acc} and which was performed n times the following effort:

$$E(n, P_{acc}) = -\log(1 - P_{acc}) \cdot n \tag{5.1}$$

Instead of testing a V&V task with a test of probability p, we could alternatively test this task k times with a test with probability $1 - (1 - p)^k$. Having this equivalence in mind, directly motivates the definition of effort above (by taking the log of the probability of the equivalent test). Therefore, the y-axis in Figure 5.7 can also be seen as the average number of re-runs of a test with 0.5 probability of detection to achieve the same accuracy.

To reach detection rates near 100% a tremendous amount of test-effort has to be spent, while reaching 100% is impossible. This is even true for a formal setting, since specification errors, that is, errors in the model checking engine or wrong application of these methods can never be excluded. It needs to be said that the formalization effort is already included in this approximation. Formal analyses are more accurate but more effort needs to be invested, including the formalization. Similarly a manually performed test could gain a high accuracy if it is performed by many different tests independently.

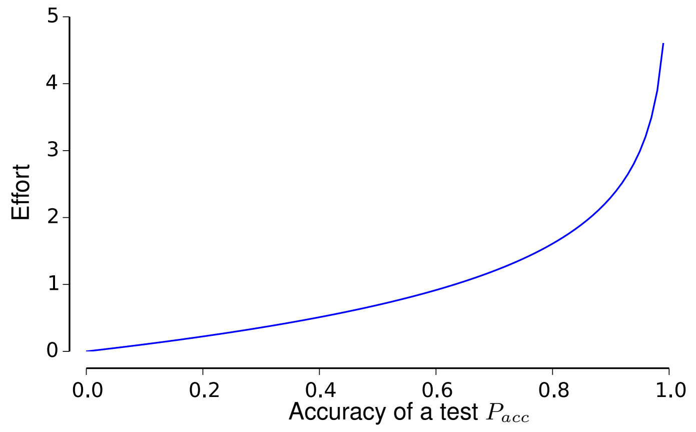


Figure 5.7 – Effort as a function of the accuracy of the test used. The higher the accuracy, the higher the associated effort according to equation (5.1)

This approximation of effort is still pessimistic, since in practice the effort for a formal solution is bounded.

5.2.2 Evaluation Results

Using the generation of architectures and the set of necessary changes we compared the effectiveness of the contract-based approach with the state of the art approach.

Figure 5.8 depicts the results of the simulation of a system depicted in Figure 5.6 with 231 requirements and 22 implementation components. The size of the necessary change, that needs to be performed to reach a consistent state again, is 13. Each dot represents the average results across simulations with 1000 samples for a given probability of detection set for each test which corresponds to a V& V task. The considered probabilities are evenly spaced in steps of 5 between 1 and 96%. The functions plotted in Figure 5.8 show the increasing effort to detect more percent of the necessary changes when increasing the accuracy of the tests used for both change processes, contract-based and state-of-the-art.

As the dots in the upper panel only show the average performance of a setting, we plotted the variability within a setting in the lower panel represented by the corresponding standard deviation.

It can be observed that the contract-based approach reaches an acceptable detection rate near 100% much earlier than the state of the art approach. Nevertheless, the needed accuracy of the single verification and validation activities needs to be very high (the highest accuracy is 0.95 in this figure). The state of the art approach needs much less accuracy to reach a similar detection rate, but since more analyses are executed the overall effort is bigger. In addition to the component based approach and the state of the art approach, the green values represent the execution of all V&V cases in the system once with the given accuracy, to get a better understanding of magnitude of the effort.

Furthermore, it can be observed that the distribution of the samples around the mean-value decreases in both cases for higher test accuracy. As expected, the deviation

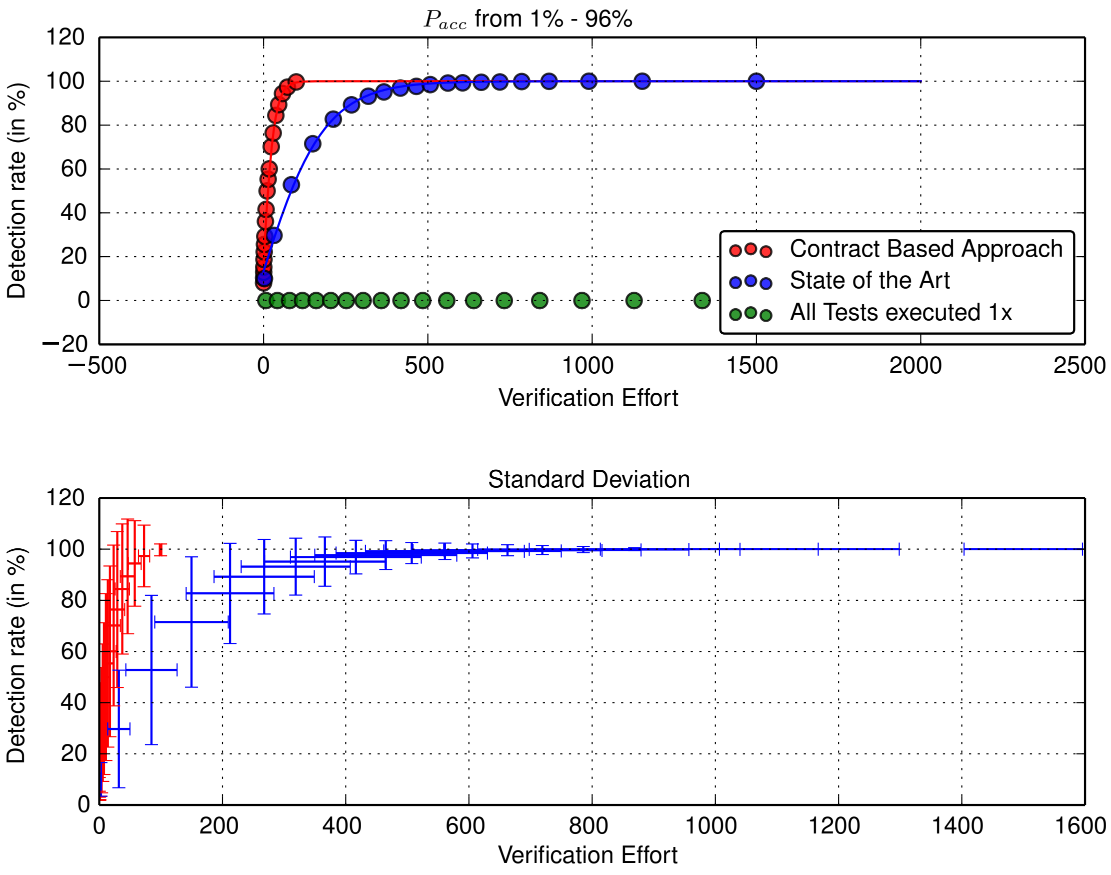


Figure 5.8 – Results of system consisting 231 requirements and 22 implementations, while changing 13 elements of the system. The sample-size is 1000

is bigger using the contract-based approach compared with the identical test accuracy of the standard approach. This is caused by analyses that do not detect problems in the system caused by previous changes. In that case the propagation stops at this branch and the system is not further investigated. This does not happen, for the standard approach, where the analyses will be executed regardless of the position in the system.

Interestingly, after approximately 60% analysis accuracy the standard approach only deviates in the effort spent, reaching nearly 100% detection rate. To some extent this was expected, but the extraordinarily good results are mainly caused by the assumed independence between two consecutive runs of the same analysis and hence increasing the accuracy exponentially. In practice, it is unlikely that a problem that has not been detected the first time will be detected by running the exact test again. Still, this might not hold for activities like reviews, tests with random inputs or tests executed by different persons. Nevertheless, this independence favors the state of the art approach as every violation of this independence assumption will decrease the performance.

Figure 5.8 displays the results for one change size only. We have run the simulation

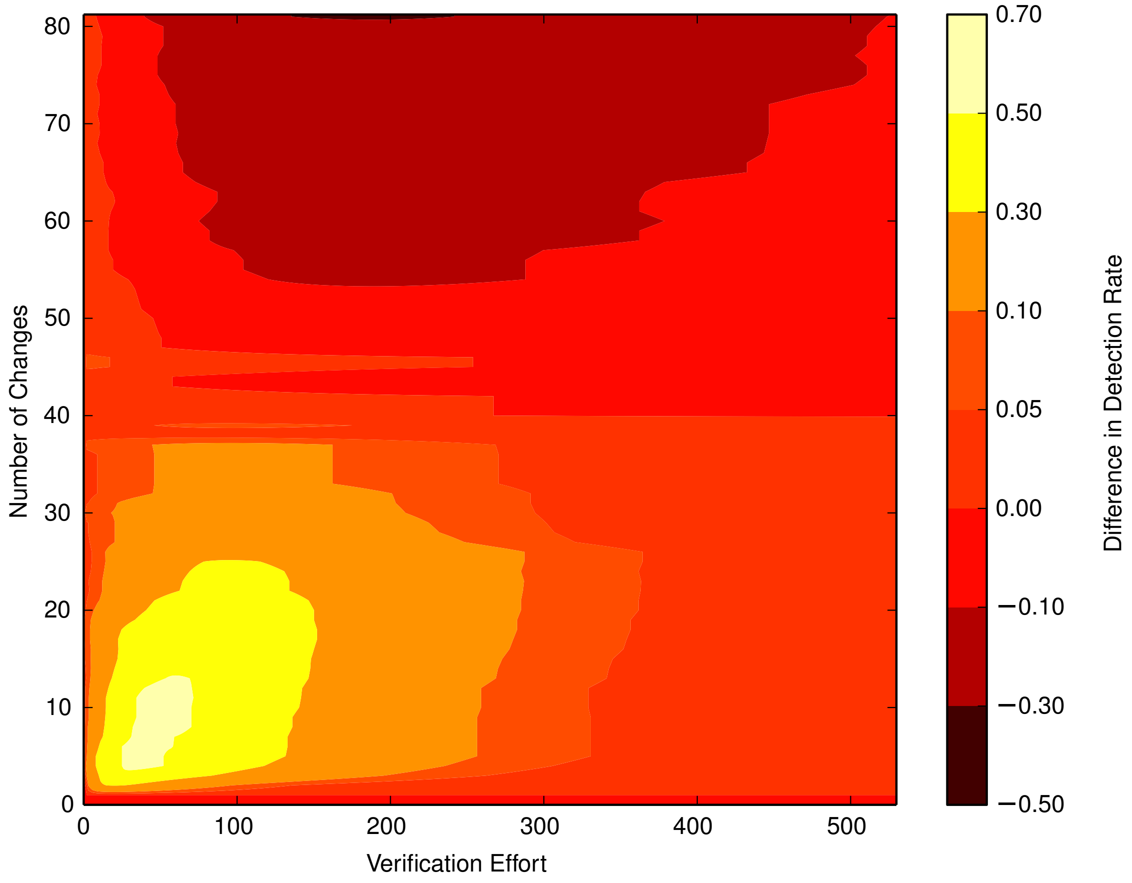


Figure 5.9 – Difference of detection rates (in %) between the component based approach and the standard approach

for various change sizes and visualized the differences in detection rates for given effort. The result is displayed in Figure 5.9.

It can be observed that the contract-based approach is creating a much higher detection rate if there are only a few changes (below 41 absolute changes ≈ 16.2% of the system size) and the effort is very limited (below 300). As already indicated in Figure 5.8 both approaches perform similarly if the available effort for the re-verification of the system is increasing. After passing the border of 41 changes, the standard approach performs better in all cases.

This is mainly caused by the iterative nature of the contract-based approach. The algorithm analyzes the requirements and implementations in the direct neighborhood of the changed element and aborts if no further change seem necessary (see e.g. lines 10 – 18 in Algorithm 3 for RefineDown links). If this decision is wrong, what might happen with a low test accuracy, the algorithm will abort too early and many necessary changes are not detected. Hence, to gain a high detection rate, the individual test accuracy needs to be also very high (see Figure 5.8). The simulation revealed, that the resulting effort grows faster than the size of the system, and hence allowing the standard approach to be more efficient for large changes.

In addition to these results the simulation provided evidence that the effort for a contract-based approach scales linearly with the size of the change. Figure 5.10 displays the effort needed by both approaches to reach a detection rate of 98%. Also the break-even point at approximately 40 changes is clearly visible.

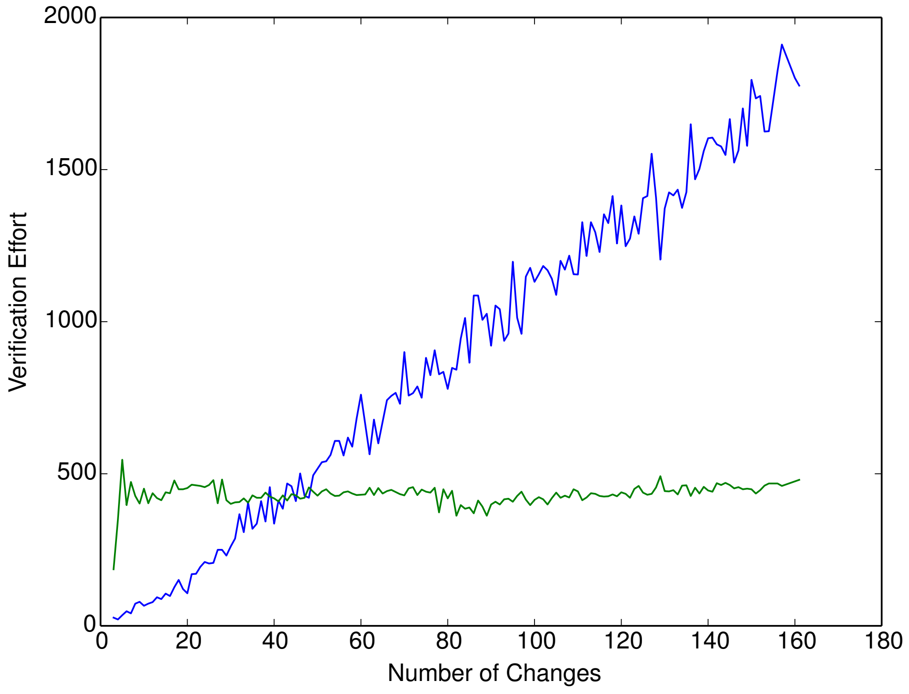
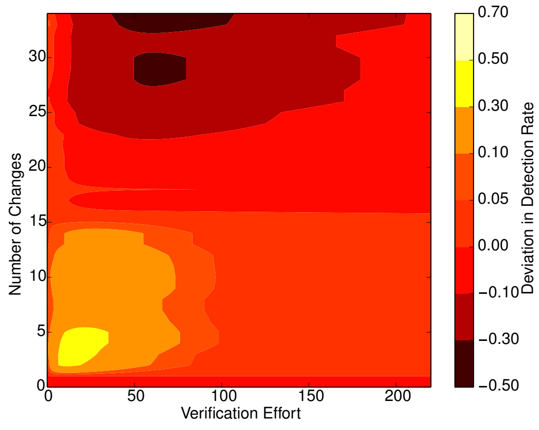


Figure 5.10 – Effort of both approaches at detecting rate 98% (300 Samples). The contract-based approach is displayed in blue, while the state-of-the-art approach is depicted in green.

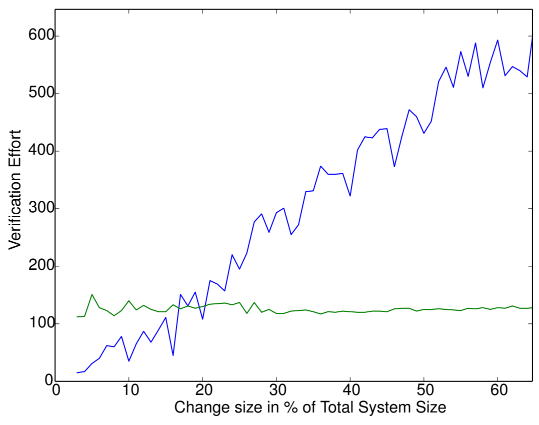
The figure looks similar for all other detection rates. It can be observed that the effort for the standard approach stays constant, while the effort of the contract-based approach increases linearly.

We evaluated the break even point also for systems of different sizes (see Figure 5.11). It can be observed that the break even point is not influenced by the size of the system. All simulated system sizes (71, 240, 771 and 1600 elements) indicate that the contract-based approach is beneficial until a relative change size of the system of 15% to 16% has been reached.

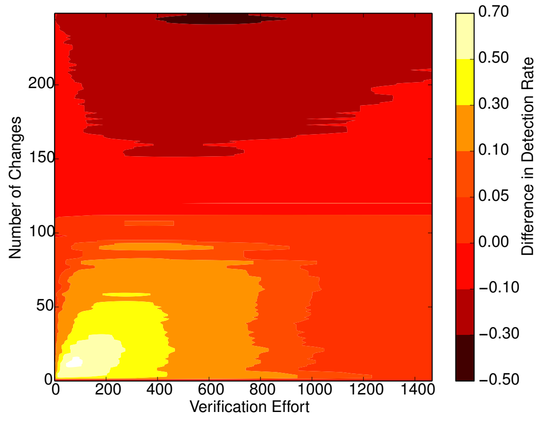
The break even analysis gives also evidence of how much costs could be saved by the contract-based approach. For small changes, that are below 7% of the total system size over 50% of the verification costs could be saves. For changes below 5% of the system size the saving is even more than 75%.



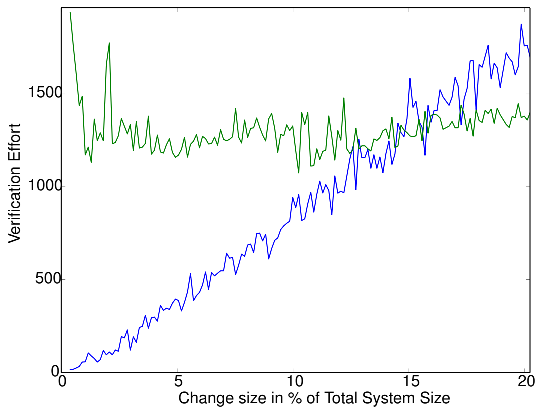
(a) #R=61, #I=10



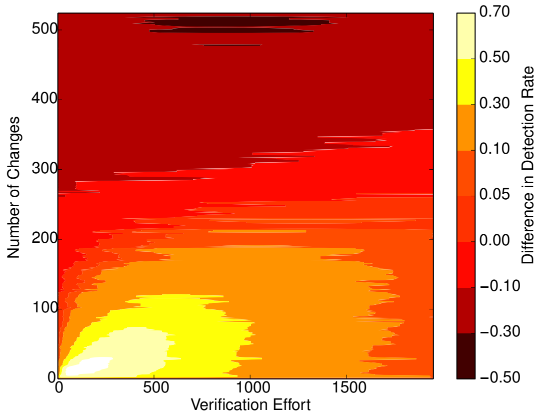
(b) Break even at 16%



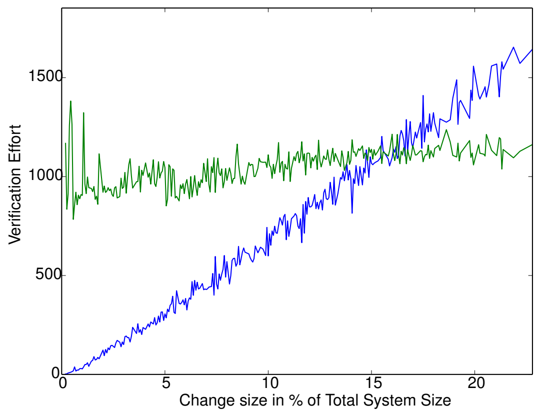
(c) #R=648, #I=87



(d) Break even at 15%



(e) #R=1414, #I=229



(f) Break even at 16% (70% detection rate)

Figure 5.11 – Comparison of the results for systems of different magnitude. The identified relative break even point is independent of the absolute system size. The plots for the break even points are scaled to the relative system size in %.

5.3 Conclusion

We have investigated the benefits and limitations of a localized change impact analysis process compared with a more standard process of complete re-verification of all elements within the systems design.

While we find the localized approach to be more effective for small proportion of necessary changes (compared with the system size) such localized approach performs less well in situations in which larger parts of the system need to be changed. Within the scope of our applied system parameters we discovered a border of $\approx$15% changes of the whole system, after which the standard approach is more efficient in terms of verification effort spend. This stems from the fact that the contract-based approach aborts the analysis process if one refinement analysis is successful. If this happens at one of the first analyzes, only a faction of the system gets analyzed and multiple changes have not been considered. For large changes, the individual accuracy of each refinement analysis needs to be extremely high, in order to be sure to detect the changes correctly, resulting in an extreme effort for huge systems if the detection rate shall remain identical. Still, the abortion of the process if one refinement analysis is successful is not always necessary. In the example presented in section 5.1, we perform multiple changes at a time and the re-verify in one chunk. If skilled engineers perform changes to a system this is a valid and effort-saving approach. Hence, the discovered border of 15% can be considered as a save worst case estimate. Looking at the possible savings in effort we can give evidence that for small changes, that are below 7% of the total system size, over 50% of the verification costs could be saved. For changes below 5% of the system size the savings is even more than 75%.

The simulation results also indicate that the use of formalized requirements and automated verification activities is highly recommended within the contract-based change management approach, since a high test-accuracy is necessary in order to obtain useful results. Yet, the standard approach does not benefit from the use of a high test accuracy. The detection rates are nearly identical with a test accuracy $\geq$ 60%. Only the overall effort increases, therefore the additional effort in formalizing requirements and performing computationally intensive analyses is not well spent.

However, it is worth noting that our simulation setup favors the process for complete re-validation since we assumed the tests and analyses results to be independent between two consecutive runs of the same test. Also, we assumed the same accuracy of implementation tests, as well as the entailment analyses.

CHAPTER 6

Prototype Implementations

In this chapter we are going to present briefly the different prototypes that have been developed for this thesis. The main purpose of this chapter is to demonstrate how the change impact analysis can be integrated in a distributed development environment. Multiple tools have been developed for this. Even though the simulation framework used in Section 5.2 has a huge code base, we will focus on the user oriented software only. Hence, two tools and a tool setup are presented. In Section 6.1 we describe how an impact analysis service can be integrated in a distributed working environment using various development tools and formats like AUTOSAR, EAST-ADL, Simulink, IBM DOORS or MS Excel. In this prototype we implemented to process of verification activity invalidation as well as their re-run and a graphical system representation that allows to view the current change request and its status. In Section 6.2 we present the refinement analysis for formalized safety contracts. Finally, we give a short overview in section 6.3 of how to perform the satisfaction analysis of safety contracts using a fault injection tool.

6.1 A Change Impact Analysis Service for Distributed Development Environments

The *Reference Technology Platform* (RTP) is a tool interaction platform based on the OSLC communication principle ("OSLC: Open Services for Lifecycle Collaboration," 2012), that connects various development tools via HTTP connected services that share a basic common meta-model, which can be seamlessly extended by the users. The first prototype of the RTP was built in the European integration project CESAR (Rajan & Wahl, 2013). Having access to all distributed development artifacts over a single interface makes it possible to develop a central impact analysis service that highlights the affected verification activities by a change and supports the developers by displaying the compensation candidates.

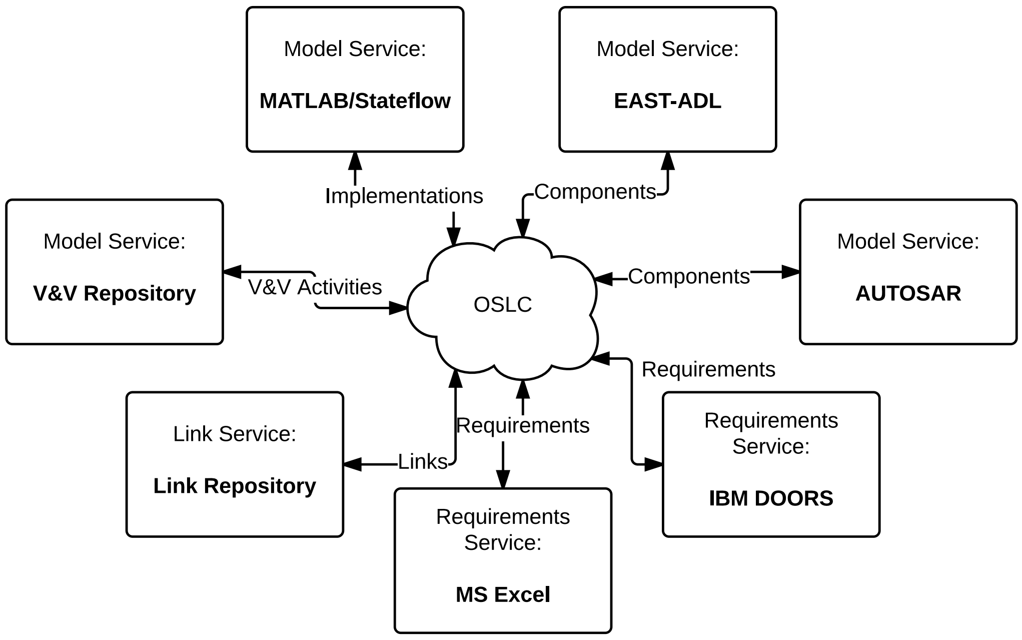


Figure 6.1 – Overview of the available services in the used RTP instance

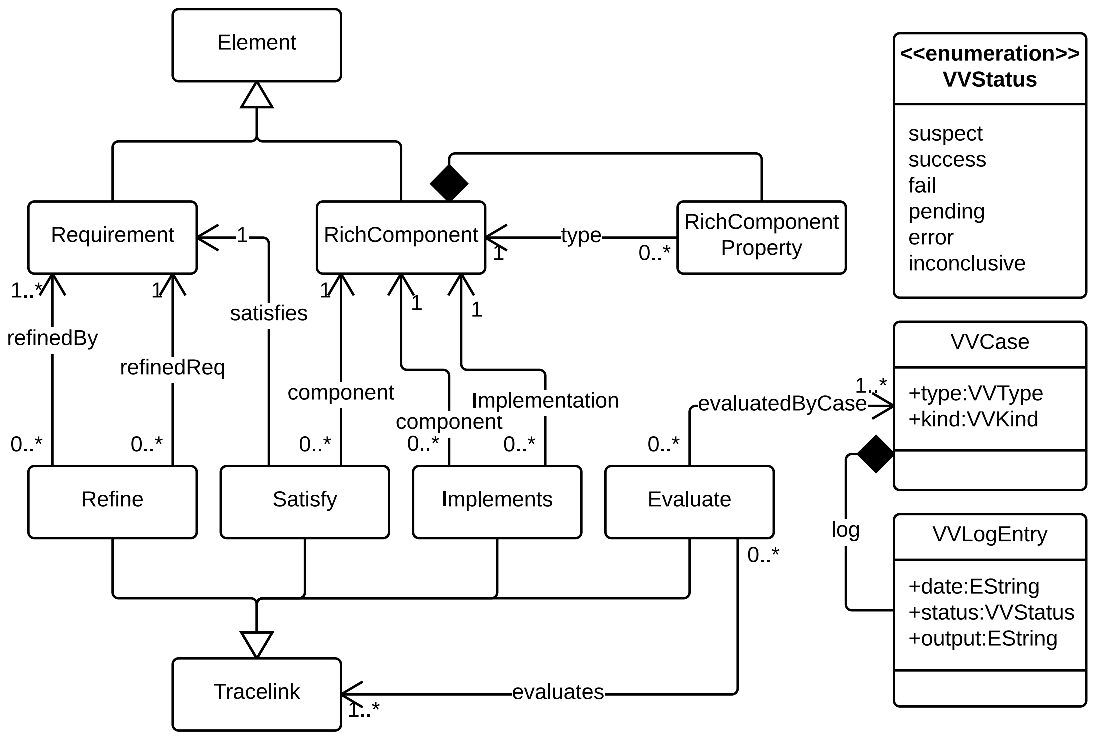
6.1.1 RTP Setup

The services integrated in the used RTP instance are depicted in Figure 6.1. Components are provided by an EAST-ADL and an AUTOSAR service, Requirements are provided by IBM DOORS and MS Excel and implementations are provided by a MATLAB/Stateflow service. This RTP instance features a dedicated link repository which stores the refine, satisfy and implementation links. This separate repository allows to perform queries related to links at a central point in the system. The verification and validation activities are represented by separate model entities and are served as an additional service. The service for the V&V activities had to be newly developed since no process enactment tool was connected to the RTP at the time of writing this thesis.

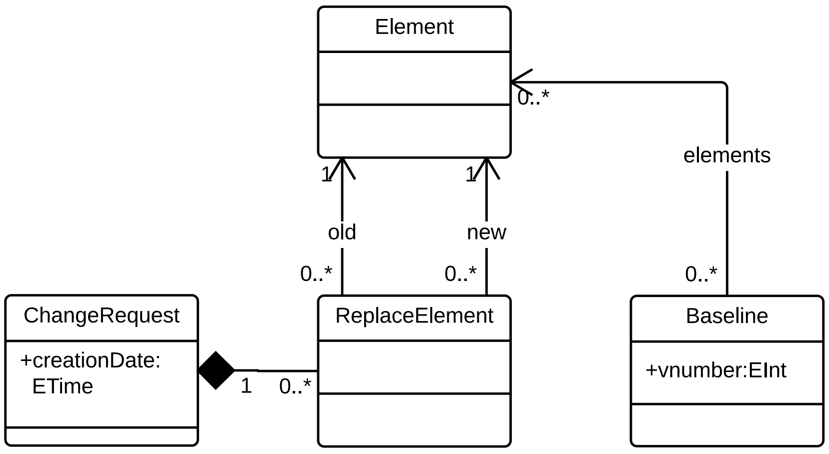
The simplified meta-model used in the RTP instance is depicted in Figure 6.2. In addition to the already introduced tracelinks an *Evaluate* link is used to connect the verification activities to the tracelinks they shall verify. The verification activities themselves can be of kind *refine* or *satisfy* and include a log, where the status of each run is stored. Implementations and components are both represented as *RichComponents* which is a concept that has been taken over from HRC (Enzmann et al., 2008). The part relations of components are realized by a Type-Prototype concept as it is used in an identical manner in AUTOSAR.

Since the OSLC configuration management specification[1] is, at the time of writing this thesis, still in a phase of frequent and extensive changes, we implemented a change

[1]https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=oslc-ccm



(a) General System Elements



(b) Impact Analysis Specific Model Elemenents

Figure 6.2 – Meta-Model used in the RTP instance

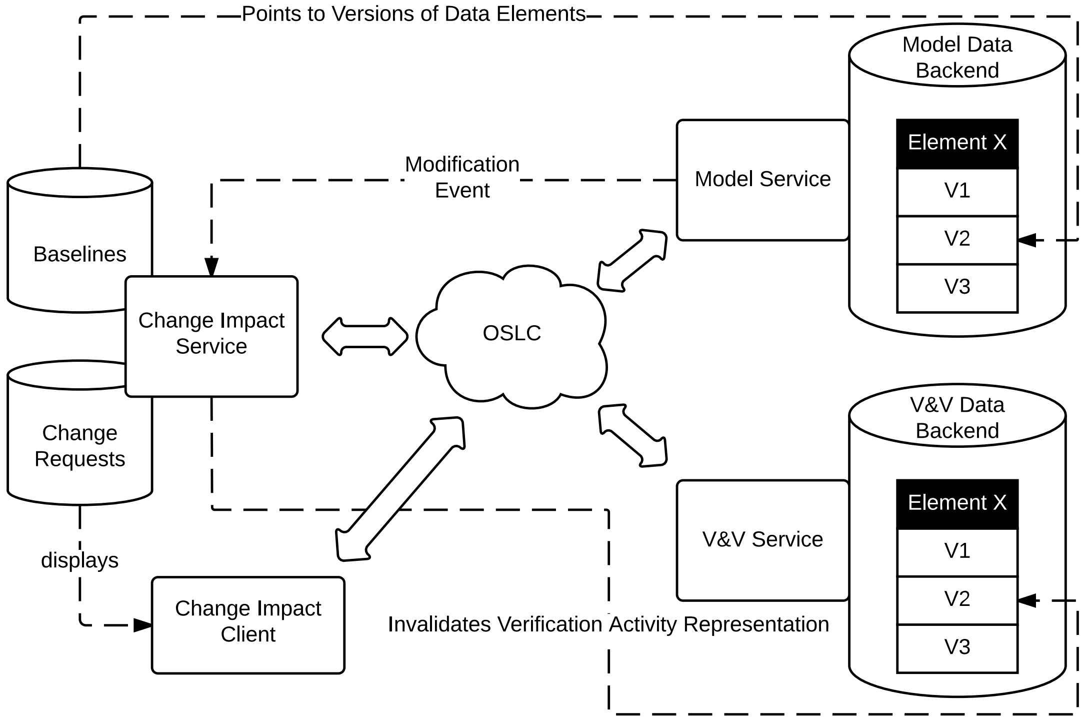


Figure 6.3 – Overview of the available services in the used RTP instance

handling mechanisms based on change events especially for this scenario. Figure 6.3 depicts the basic mechanism. As a prerequisite, each data repository needs to be able to version its elements. This is a typical requirement such as requested in ISO 10007 (2003). These elements are also exposed over the service and referenced by a unique address. In addition we require also tracelinks to be versioned and pointing to versioned elements. This is necessary since the semantics of a link may not be applied for a changed element, at least not without an additional analysis. For example, a component covers a special requirement, but after changing the requirement it has to be considered again whether the attached component is still the best to implement this requirement. The link is only valid for a special version of a requirement. Whenever an element is changed in the repository, the service sends a notification in the form of a *change event* to the change impact service. The change causes a new version of the element to be created. Furthermore, after the impact analysis service receives the change event, the connected verification and validation activities are invalidated, by setting the appropriate status in the V&V model elements. Baselines are also created by the change impact service. They refer to the specific version of the included elements.

6.1.2 Algorithmic Changes for Distributed Development Environments

The change impact process presented in section 3.3 assumes that only one person is performing a change to the system. Since the verification activities might run for a

Algorithm 5 Contract-based approach

```
Require: $\Psi = \emptyset$
 1: procedure CBA($i$)
 2:     $V$ = connected_VV($i$)
 3:     setSuspect($V$)
 4:     $\Psi = \Psi \cup V$
 5:     while $\Psi \neq \emptyset$ do
 6:         $v \in \Psi, \Psi = \Psi \setminus \{v\}$
 7:         if isSuspect($v$) then
 8:             run($v$)
 9:             if isFailed($v$) then
10:                 $\Psi = \Psi \cup \{v\}$
11:             end if
12:         end if
13:         if isFailed($v$) then
14:             $T$ = targets($v$)
15:             for $t \in T$ do
16:                 if suitableForChange($t$) then
17:                     modify($t$)
18:                     $V2$ = connected_VV($t$)
19:                     setSuspect($V2$)
20:                     $\Psi = \Psi \cup V2$
21:                 end if
22:             end for
23:         end if
24:     end while
25: end procedure
```

significant amount of time, it cannot be expected that the results are available immediately. To prevent a wrong status of the verification activities between the start of the analysis and the result, two additional stati *suspect* and *processing* have been introduced. A suspect status is set as soon as a verification activity has been identified to be affected by a change. If the analysis is finally running, the status is changed to processing. The suspect status allows a slightly simpler algorithm to handle the verification activities (see Algorithm 5)

The process starts with a given initial change i. As the component that is changed is linked to several other components and requirements, various verification and validation activities have to be addressed. In case the changed element is a requirement, the verification activities that have to be addressed are two entailment checks, one where i is the top-level requirement, and the other, where i is a subrequirement. Also, at least one satisfaction analysis to check if the implementation fulfills the requirements has to be performed. All the previous results of these analyses and tests are not valid anymore since the requirement has been altered. The status of these verification and validation activities is therefore set to *suspect*, indicating that a re-run is necessary. If the analysis fails, it is added to the list of V&V activities (Ψ) linked to components in which an additional change might be needed to guarantee an overall consistent state. From the elements that are linked from the failed V&V activity (`targets`), the engineer selects further system elements and modifies them. The modification, again, leads to the invalidation of previous gained analysis results. The suspicious activities will be added to the set of verification and validation activities Ψ, that still need to be processed. The algorithm terminates if Ψ is empty. This indicates that all activities have been performed again, with an successful result.

In contrast to the algorithms used for the simulation (see Section 5.2), where the expected change was pre-calculated, this decision is given to the engineer in Algorithm 5. Hence, the algorithm might not terminate if two system elements are continuously changed after another. Nevertheless, this behavior is not expected in practice. Since the introduction of suspect flags does not alter the detection rate and test accuracy identified in Chapter 5 the gained results still remains valid. However, the running time in a real world scenario highly depends upon the running time of the chosen verification technology. The running time may vary between constant grows for manual review or exponential grows for model checking based analyses.

6.1.3 Change Request Representation

The central point of user interaction with the change impact service is the change impact client *ReMain*. ReMain displays the changes in the current change request to the user and indicates which verification activities are affected. Furthermore, status lights behind the verification activities show the status of the activity. Yellow for suspect verification activities, green for successful ones and red for activities that failed. The components and requirements displayed in the change request view are selected only to be visible if they are either a potential compensation candidate of a failed verification activity (highlighted by a blue name), or have been so. The full architecture with all components, requirements and implementations can be browsed in the left view of the client, where all connected services are listed in a browsable fashion.

6.1.4 Using the Change Impact Service

If the impact analysis service is used while developing a system, each developer who is working on the project is running an instance of the change management client. Starting from a specified baseline the changes made by all developers are visible in the client. In fact, only the latest changes are visible, if an element is changed multiple times. This decision is based on the need of the developers to identify the parts of the system that have been changed in a simple way. Highlighting multiple version would have complicated the now simple interface with no additional benefit. Now, if an element has been changed by a developer, all other developers are able to detect possible influences to elements

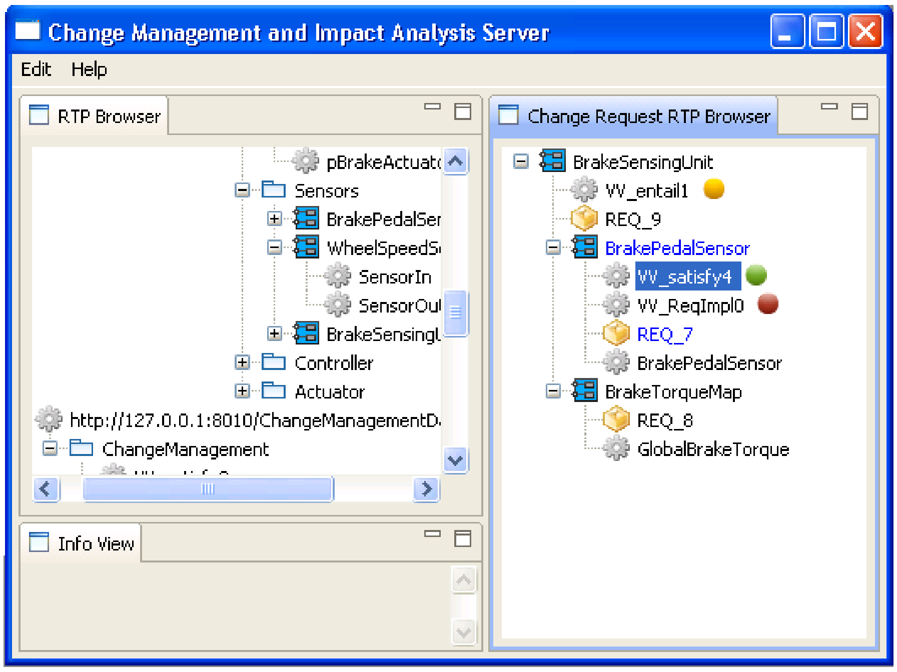


Figure 6.4 – Representation of the current change request in the change impact client ReMain

maintained by them. The indicators allow judging if further changes might be necessary or the change is already contained in the highlighted region.

6.2 Checking Safety Contracts with Divine

DIVINE (Barnat et al., 2013) is an explicit state LTL model checker that is used as the backend for analyzing safety contracts. This thesis does not focus on providing the fastest possible analysis of LTL properties, hence performance was not a concern for selecting a suitable checker. Divine has been chosen since it provides a simple integrated language to describe systems by means of processes.

To analyze the refinement of contracts, the virtual integration condition of their LTL formulas needs to evaluate to true. This satisfiability check on LTL formulas can be transformed to a classical model checking problem as described by Rozier and Vardi (Rozier & Vardi, 2007) as well as Li et al. (Li et al., 2014) using a generic model, allowing all possible behavior, to check the property against.

Hence, the analysis prototype is performing three process steps:

1. Parse the input contracts
2. Generate LTL expressions for contracts
3. Generate the Random model

4. Generate the refinement condition(s)
5. start the checker

We use a very simple example to still create formulas that are comprehensible without tools. The component architecture is depicted in figure 6.5.

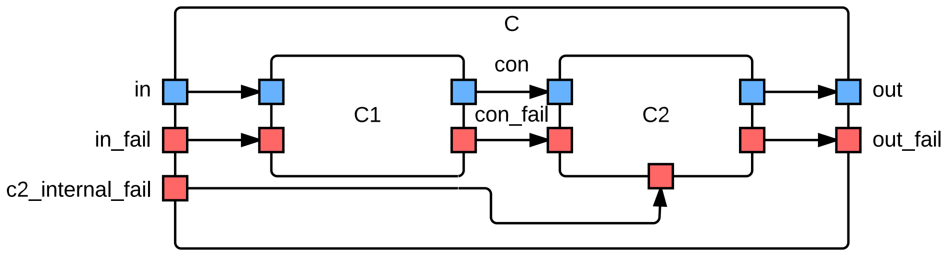


Figure 6.5 – Component architecture used as the example to illustrate the refinement analysis process.

To be able to easily integrate the refinement analysis in various toolchains, the contracts are provided by a simple text file. An example of a very simple input file is depicted in Figure 6.6. The top level contract is marked with an additional “t” after the assumption marker “A” and the guarantee marker “G.” The complete grammar for the syntax of the safety contracts is listed in Figure 6.7.

```
/* This is the top-level contract */
At: none of {{in_fail},{c2_internal_fail}} occurs.
Gt: {out_fail} does not occur.

// contract for C1
A: {in_fail} does not occur.
G: {con_fail} does not occur.

//Contract for C2
A: none of {{con_fail},{c2_internal_fail}} occurs.
G: {out_fail} does not occur.
```

Figure 6.6 – Simple example input file stating the top level contract and the subcontracts.

The random model, which is used to check the generated LTL property against, needs to be able to perform any change of the used ports at any time. Hence, for each port a process is created using the build in language DVE of DIVINE. An example process for the input malfunction port `in_fail` is depicted in Figure 6.8.

The generation of the LTL property is based on the virtual integration condition on formulas, as presented in Section 2.1.5.

⟨*assumptionTop*⟩ ::= ’At:’ , ⟨*safetypattern*⟩;

⟨*promiseTop*⟩ ::= ’Pt:’ , ⟨*safetypattern*⟩;

⟨*assumption*⟩ ::= ’A:’ , ⟨*safetypattern*⟩;

⟨*promise*⟩ ::= ’P:’ , ⟨*safetypattern*⟩;

⟨*safetypattern*⟩ ::= ⟨*p1*⟩ | ⟨*p2*⟩ | ⟨*p3*⟩ | ⟨*p4*⟩;

⟨*p1*⟩ ::= ’none of’, ⟨*exprset*⟩, ’occurs’, [’.’];

⟨*p2*⟩ ::= ⟨*expr*⟩, ’does’, ’not’, ’occur’, [’.’];

⟨*p3*⟩ ::= ⟨*mode*⟩, ’only’, ’followed’, ’by’, ⟨*mode*⟩;

⟨*p4*⟩ ::= ⟨*mode*⟩, ’only’, ’after’, ⟨*mode*⟩;

⟨*exprset*⟩ ::= ’{’, ⟨*expr*⟩ , {’,’, ⟨*expr*⟩ }, ’}’;

⟨*expr*⟩ ::= ’{’, ⟨*malfunction*⟩, {’,’, ⟨*malfunction*⟩}, ’}’;

⟨*malfunction*⟩ ::= ⟨*andExpr*⟩ | ⟨*singleMalfunction*⟩ | ⟨*mfCountPort*⟩;

⟨*not*⟩ ::= ’!’;

⟨*singleMalfunction*⟩ ::= [⟨*not*⟩], ALPHA, {(ALPHA | ‘_’)};

⟨*andExpr*⟩ ::= ⟨*singleMalfunction*⟩, ‘`and`’ , ⟨*singleMalfunction*⟩;

⟨*mfCountPort*⟩ ::= ⟨*countName*⟩, ’=’, ⟨*countValue*⟩;

⟨*countName*⟩ ::= ALPHA, {(ALPHA | ‘_’)};

⟨*countValue*⟩ ::= NUM, {NUM};

Figure 6.7 – Grammar of the safety pattern

```
process in_fail_switch {
state in_fail0, in_fail1;
init in_fail0;
trans
in_fail0 -> in_fail1 {effect in_fail=1;},
in_fail1 -> in_fail0 {effect in_fail=0;},
in_fail0 -> in_fail0 {guard in_fail==0;},
in_fail1 -> in_fail1 {guard in_fail==1;};
}
```

Figure 6.8 – One example process from the random model. The input ports (infail in this case) can be switched at any time.

Since safety contracts are not in canonical form, we use the parallel composition defined by Hugar:

$$\left(\left(\bigwedge_i A_i\right) \vee \bigvee_i \left(A_i \wedge G_i^{-1}\right), \bigwedge_i G_i\right)$$

A contract $C = (A, G)$ is refining contract $D = (B, H)$ given not in canonical form iff $B \to A$ and $C \to D$. Hence, the virtual integration condition for a top level contract $C = (A, G)$ and C_i subcontracts is:

$$\left[A \to \left(\bigwedge_i A_i \vee \bigvee_i \left(A_i \wedge G_i^{-1}\right)\right)\right] \wedge \left[\left(\left(\bigwedge_i A_i \vee \bigvee_i \left(A_i \wedge G_i^{-1}\right)\right) \to \bigwedge_i G_i\right) \to (A \to G)\right]$$

This expression can be simplified to:

$$\left(\bigwedge_i A_i \wedge G\right) \vee \bigvee_i (A_i \wedge \neg G_i) \vee \neg A$$

For the example above the generated formula is:

```
(((G(!(in_fail==1)))) && (((G(!(con_fail==1))) &&
(G(!(c2_internal_fail==1))))) && ((G(!(out_fail==1))))) ||
(((G(!(in_fail==1)))) && !((G(!(con_fail==1))))) ||
((((G(!(con_fail==1))) && (G(!(c2_internal_fail==1))))) &&
!((G(!(out_fail==1)))))|| !(((G(!(in_fail==1))) &&
(G(!(c2_internal_fail==1)))))
```

The output of DIVINE is that the property holds, that is, refinement is given. If we modify the assumption of the top level contract that the malfunction `c2_internal_fail` is not present, refinement is not given anymore and the model checker creates a counter example, which is:

```
c2_internal_fail = 1, con_fail = 0, in_fail = 0, out_fail = 0
```

Hence, the occurrence of a single internal malfunction on `C2` will cause the property to fail. This counter example gives evidence that the architecture is not safe with respect to the given assumptions and that additional measures are necessary to handle the internal malfunction.

6.3 Satisfaction Check of Safety Contracts

To perform the satisfaction check on safety contracts we use a fault injection tool called MBSA (Peikenkamp et al., 2006). We had not yet automated the process of creating all necessary MBSA inputs from the safety specification, but perform these steps manually. We use the MBSA prototype, which has been developed for the European project SAFE[2]. In that project the MBSA has been integrated in an eclipse platform which is capable to handle EAST-ADL and AUTOSAR files. In this section we briefly introduce which steps are necessary to run a satisfaction analysis and how to perform them.

The MBSA (Peikenkamp et al., 2006) performs fault injection in a model of the system's nominal behavior. The correctness of this model can be checked with model checking techniques against the functional requirements. The MBSA can be used to automatically assess which combinations of malfunctions lead to the violation of a selected functional requirement. The resulting cut-sets (of malfunctions) can be represented as a fault-tree. A cut-set is said to be minimal if no event can be removed from the set and the combination of malfunctions still leads to a failure (Kececioglu, 1991). The analysis currently supports nominal behavioral models formalized in MATLAB/Stateflow[3] while requirements are provided in a formal language called RSL (Baumgart et al., 2011) to enable automatic processing.

We need the following prerequisites to perform the analysis:

- EAST-ADL or AUTOSAR model to represent the component architecture
- Safety contracts describing the fault propagation behavior, that shall be analyzed
- MATLAB Stateflow implementation of the component that shall be analyzed
- A set of malfunctions that shall be injected

First, the top level requirement needs to be stated. This requirement needs to be the functional representation of the safety contract that shall be analyzed. How to formulate the functional representation is described in detail in section 4.5.2.

The functional part of the RSL (Mitschke et al., 2010) is used to describe the top level event. Some frequently used patterns are:

[2]http://www.safe-project.eu/
[3]http://www.mathworks.de/products/simulink/

- **always(CONDITION)**
  specification of an invariant condition for the system.
- **Whenever EVENT1 occurs EVENT2 occurs during [INTERVAL].**
  After the occurence of EVENT1 an instance of EVENT2 needs to occur in the defined interval.
- **Whenever EVENT occurs CONDITION holds during [INTERVAL].**
  After the occurence of EVENT the condition has to hold at every time of the specified interval.

An event is specified using a variable name and a value and is triggered as soon as the specified value is assigned to the variable. Conditions can be stated in C-Syntax.

The top level requirement can be stored in one of the loaded models, for example, in a SAFE-Extension to an AUTOSAR file. In the SAFE meta-model the description of the malfunctions is performed in a model element *malfunction*. These descriptions can be selected using the configuration interface of the MBSA (see Figure 6.9). Hence, not all malfunctions need to be injected, but the analysis can also perform a partial injection analysis. Next, the Simulink model needs to be selected. The model can be either located in the workspace of the eclipse platform or in the filesystem of the host.

Next, the malfunctions need to be configured for the implementation model. An additional tool, the *failure mode editor* is used for this. In this tool (see Figure 6.10) for each malfunction of the corresponding deviation in the implementation model can be configured. To achieve this, there are two different possibilities: first a pre-defined fault behavior from a library can be selected. This library defines the deviation of a variable in the implementation model from its intended value. Currently supported deviations are:

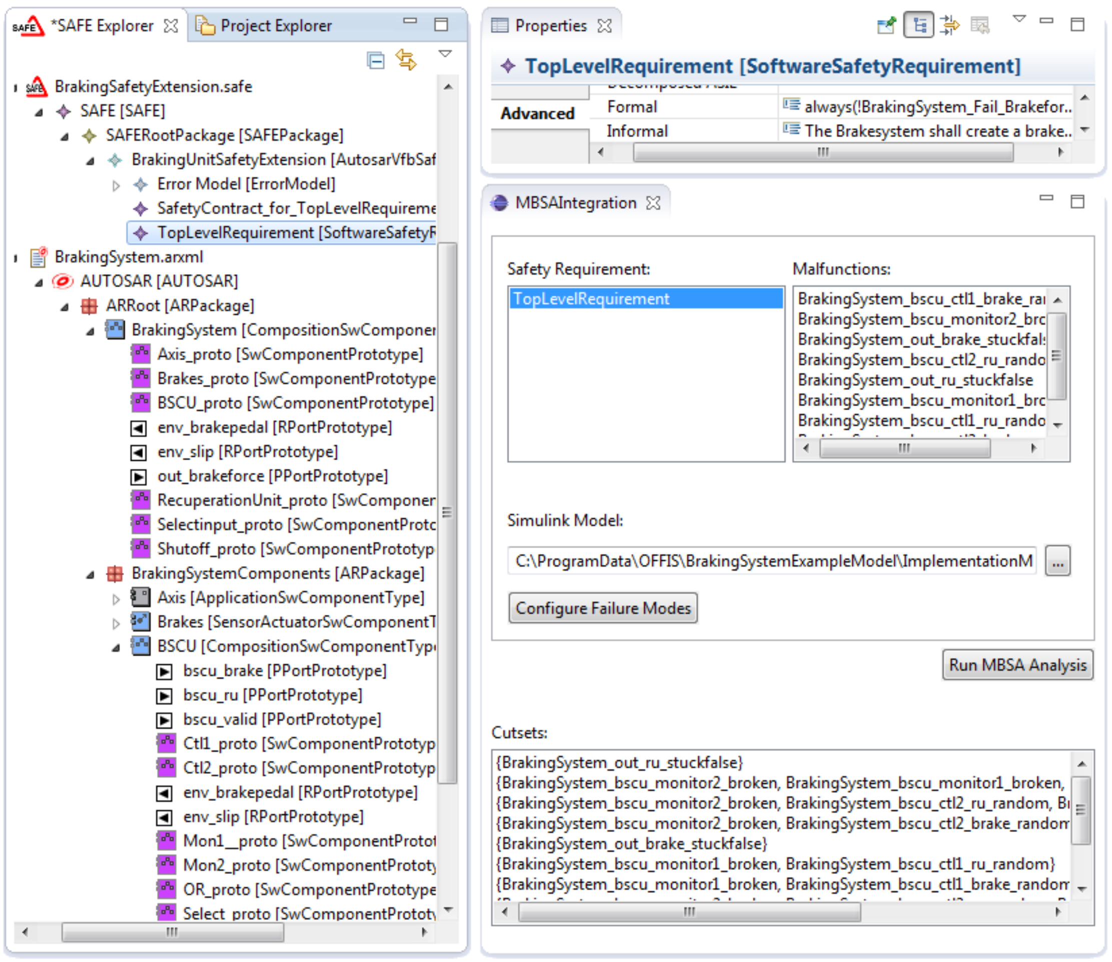
Stuck-at: this pattern describes the case in which an internal variable of the system model is stuck at an erroneous value.
Random: this pattern is used to describe cases in which random changes to the value of an internal variable occur.

The second possibility is to use a user defined malfunction behavior. This approach is used in case the desired fault behavior cannot be modeled using the fault library. The user defined mode allows the custom modeling of this fault behavior in the same language of the implementation model. Accordingly, the fault is not injected later in the process in the nominal model but it is embedded in the nominal model. Additional input variables are then added to the model to control the activation of the injected fault.

As with the user define fault behavior the top level event (top level requirement) can be specified directly inside the model. A state-flow block can be used as an observer to define the behavior, and variables can be set that indicate, for example, the entry of a fail state (see Figure 6.11). The functional requirement is then referencing this variable:

$$always(!Fail_Brakeforce)$$

After configuring the malfunctions the analysis can be started. The current implementation supports two analysis engines as a backend. The first one is based on the VIS



BrakingSafetyExtension.safe
SAFE [SAFE]
SAFERootPackage [SAFEPackage]
BrakingUnitSafetyExtension [AutosarVfbSaf
Error Model [ErrorModel]
SafetyContract_for_TopLevelRequireme
TopLevelRequirement [SoftwareSafetyR
BrakingSystem.arxml
AUTOSAR [AUTOSAR]
ARRoot [ARPackage]
BrakingSystem [CompositionSwCompone
Axis_proto [SwComponentPrototype]
Brakes_proto [SwComponentPrototype
BSCU_proto [SwComponentPrototype]
env_brakepedal [RPortPrototype]
env_slip [RPortPrototype]
out_brakeforce [PPortPrototype]
RecuperationUnit_proto [SwCompone
Selectinput_proto [SwComponentProt
Shutoff_proto [SwComponentPrototyp
BrakingSystemComponents [ARPackage]
Axis [ApplicationSwComponentType]
Brakes [SensorActuatorSwComponentT
BSCU [CompositionSwComponentTyp
bscu_brake [PPortPrototype]
bscu_ru [PPortPrototype]
bscu_valid [PPortPrototype]
Ctl1_proto [SwComponentPrototyp
Ctl2_proto [SwComponentPrototyp
env_brakepedal [RPortPrototype]
env_slip [RPortPrototype]
Mon1__proto [SwComponentProto
Mon2_proto [SwComponentProtot
OR_proto [SwComponentPrototype
Select_proto [SwComponentProtot

TopLevelRequirement [SoftwareSafetyRequirement]

Formal	always(!BrakingSystem_Fail_Brakefor..
Informal	The Brakesystem shall create a brake..

Safety Requirement: TopLevelRequirement

Malfunctions:
BrakingSystem_bscu_ctl1_brake_ra
BrakingSystem_bscu_monitor2_brc
BrakingSystem_out_brake_stuckfal
BrakingSystem_bscu_ctl2_ru_rando
BrakingSystem_out_ru_stuckfalse
BrakingSystem_bscu_monitor1_brc
BrakingSystem_bscu_ctl1_ru_rando

Simulink Model: C:\ProgramData\OFFIS\BrakingSystemExampleModel\ImplementationM

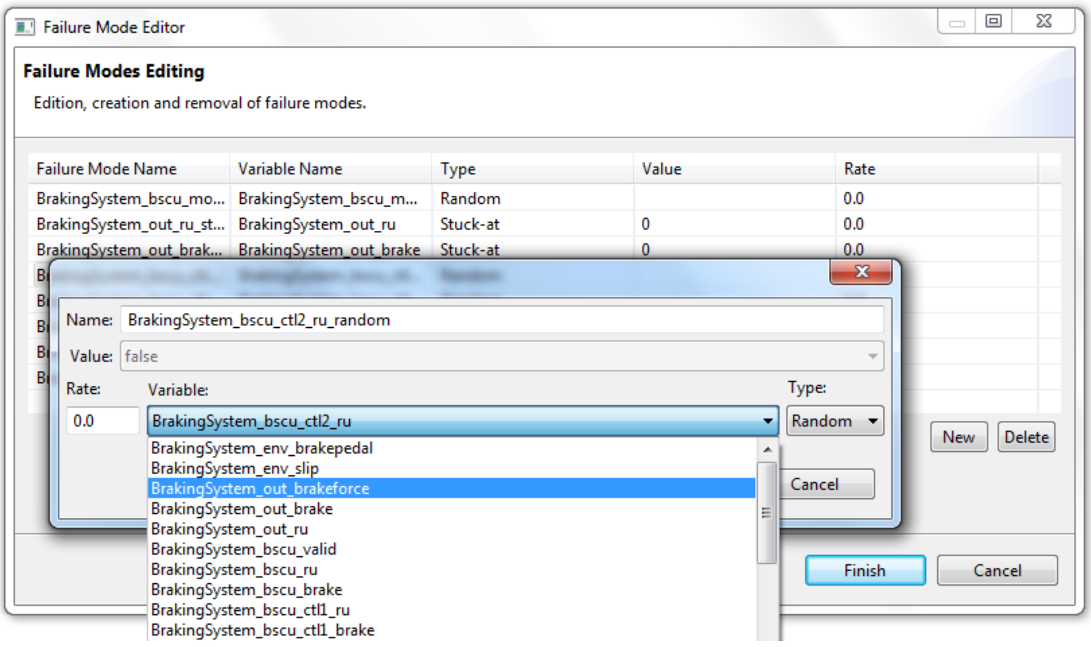
Cutsets:
```
{BrakingSystem_out_ru_stuckfalse}
{BrakingSystem_bscu_monitor2_broken, BrakingSystem_bscu_monitor1_broken,
{BrakingSystem_bscu_monitor2_broken, BrakingSystem_bscu_ctl2_ru_random, B
{BrakingSystem_bscu_monitor2_broken, BrakingSystem_bscu_ctl2_brake_randon
{BrakingSystem_out_brake_stuckfalse}
{BrakingSystem_bscu_monitor1_broken, BrakingSystem_bscu_ctl1_ru_random}
{BrakingSystem_bscu_monitor1_broken, BrakingSystem_bscu_ctl1_brake_randon
```

Figure 6.9 – Configuration interface of the MBSA integrated in the SAFE tool platform

model checker [4]. The VIS based backend guarantees complete results since the full state space is checked. The monte-carlo simulation based engine can be used to cope with models that are more challenging in terms of the state space size. In this case, however, the completeness is not guaranteed anymore. In the present work, we use the VIS based engine.

At run-time, the Stateflow model is transformed into the input language of the model checker. This nominal model is extended (injected) with the faults. Additionally, an observer automaton for the analyzed requirement is generated and injected in the model if an RSL expression is used in the functional requirement. The resulting overall model is finally passed to the VIS model checker. The analysis identifies all state sequences leading from the set of initial system states over the activation of faults to the observation of the violation of the functional requirement. These paths are the basis for computing

[4] http://vlsi.colorado.edu/~vis/



Failure Mode Editor

Failure Modes Editing

Edition, creation and removal of failure modes.

Failure Mode Name	Variable Name	Type	Value	Rate
BrakingSystem_bscu_mo...	BrakingSystem_bscu_m...	Random		0.0
BrakingSystem_out_ru_st...	BrakingSystem_out_ru	Stuck-at	0	0.0
BrakingSystem_out_brak...	BrakingSystem_out_brake	Stuck-at	0	0.0

Name: BrakingSystem_bscu_ctl2_ru_random

Value: false

Rate:	Variable:	Type:
0.0	BrakingSystem_bscu_ctl2_ru	Random

BrakingSystem_env_brakepedal
BrakingSystem_env_slip
BrakingSystem_out_brakeforce
BrakingSystem_out_brake
BrakingSystem_out_ru
BrakingSystem_bscu_valid
BrakingSystem_bscu_ru
BrakingSystem_bscu_brake
BrakingSystem_bscu_ctl1_ru
BrakingSystem_bscu_ctl1_brake

Figure 6.10 – Configuration of malfunctions using the FailureModeEditor

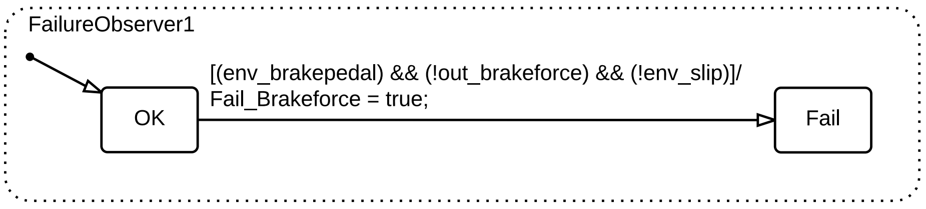


Figure 6.11 – Observer to specify the top level funtional requirement directly in the Stateflow model

the set of minimal cut-sets leading to the failure.

The identified cut-sets are displayed in the MBSA integration interface.

CHAPTER 7

Conclusion

Certification costs consume a major part of the total development budget of safety critical embedded systems. These costs consist to a great extent of the verification and validation activities, which guarantee the compliance of the system to its requirements and safety standards. Systems are rarely developed from scratch, but are built upon existing products that are modified to various degrees. Hence, it is unfavorable that even in case of small changes the safety case, that is, the argumentation why a system is sufficiently safe, needs to be revised. In many cases the verification and validation activities need to be re-executed for the whole system. This is mainly caused by the inability to determine the possible side-effects of a change. This problem is caused by changes that might "propagate" and cause undesirable behavior in even distinct components of the system. Current impact analysis techniques are not suited to limiting the resulting re-verification effort since the set of verification activities affected by a change cannot be determined precisely enough. Furthermore, impact analyses developed for software cannot be transferred to logical system descriptions, which are not yet implemented. In this thesis a new impact analysis technique for functional safety concepts has been developed, which offer a linear relation between the size of the change and the re-verification effort using a new concept of detecting change propagation and a modular system safety specification.

7.1 Summary of Obtained Results

We evaluated the existing change impact analyses (see Section 3.1) that are suited for embedded systems and identified two main issues: First, the set of possibly affected system elements is too large and no exact pruning strategy exists to select the actually affected elements from the set of potentially affected ones. Second, if approaches limit the set of possibly affected system elements in a probabilistic setting, mostly based on experience from previously built systems, the confidence in the correctness of this selection is not sufficient for qualification or certification purposes. Hence, a new impact analysis technique based on contracts was developed (see Section 3.3.4). The approach has the benefit of exactly determining if a component is affected by a change. This feature is founded on the semantics of the specified requirements of the system, representing one of the main differences from the other existing approaches that are either based on traceability links or predefined dependencies. Even if these dependencies are extracted from behavioral models such as source code, they focus only on selected properties such as variable usage or call relations and represent therefore a sub-set of the intended behavior. Furthermore, using the semantic relationship between requirements we provide a decision support for the engineer. It is possible to identify whether a change can be compensated by modifying a set of requirements rather than modifying an implementation. In addition, to avoid costly implementation tasks we can guarantee that the changes in requirements preserve the current status of the verification activities. Hence, no further re-verification is needed (see Section 3.3.5). This decision support system is using the slack in system. Since slack in automotive applications is not included intentionally, it is typically very little. Hence, slack is identified in the complete system and a shift of it is calculated. Since, many requirements in distinct parts of the system might be modified, this task is very unlikely to be performed correctly by a human on a large scale system.

To enable an automatic processing of requirements, the impact analysis technique requires a formal system specification. After analyzing the existing safety specification techniques that provide means for modularization (see Section 4.1), we selected formal safety patterns as a base for our extensions. Several arguments back this decision: First, safety patterns have defined semantics in LTL, which integrate well with the contract-based approach that is needed for the impact analysis. Second, although they translate into a formal specification, the pattern itself are easy to understand phrases that do not require special training in, for example, propositional logic. Third, since the safety specification is based on written text, rather than the currently popular graph based structures like GSN or SHIP, all existing requirements management tools in the companies can continue to be used without the need of any modifications. Still, the existing safety patterns were not sufficient to express functional safety concepts as it is needed to comply with such safety standards as the ISO 26262. To this end, the language needed to be extended to support multiple failure modes, in particular the concept of safe states (see Section 4.3.1). Then, to support easy specification of safety mechanism, contract templates have been defined to cover the most common elements of safety concepts (see Section 4.3.2). One of the major contribution of this work is the introduction of an abstraction mechanisms in safety contracts to allow a top down design of the systems specification using refinement. This abstraction is based on the idea of specifying the number of potential malfunctions in a part of the system without detailing the underlying faults (see Section 4.3.3). The concrete faults in a system are defined on the lowest level of decomposition only. This new abstraction technique has the benefit of preserving the already verified properties even if the system is further refined. This type of specification is not only necessary to be modular in a way that the impact analysis can determine the correct affected regions, but also reflects much better the needs from safety standards and industry, which typically require avoiding single or double faults, without specifying further the details of the faults. In fact, the atomic faults of a system are not known at the time of building a functional safety concept. Furthermore, an analysis had to be developed to check the compliance of an implementation to the given safety specification. Even though we used existing fault-injection technologies, the relation between the safety specification and the functional requirements of the system had to be detailed. Finally, we are able to fully represent safety contracts as a functional deviation from the intended behavior usable in a functional injection analysis (see Section 4.5.2).

Having developed a combination of a change impact analysis and a safety specification mechanism able to perform an impact determination with a linear relation between the verification effort and the size of the change, we needed to collect evidence to support this claim. First, we fully specified a case-study in safety and functional terms and applied changes to it, to demonstrate the potentials of the impact analysis as well as the safety specification (see Section 5.1). Although a significant savings was achieved within the example model, this approach could not be used to show the efficiency of the approach in general. Because of the lack of multiple sufficiently large models and realistic changes, we developed a stochastic simulation framework that acts as a proxy to compare the developed impact analysis approach with the current state-of-the-art approach of re-verifying the complete system (see Section 5.2). The simulation approach has the advantage over the analysis of real user models that results are not distorted by multiple involved engineers. In the simulation we can guarantee that the engineers take identical decisions in both approaches. In addition, the key properties of the approaches such as the size of the system, the size of the change or the accuracy of the conducted verification activities could be easily adapted using a stochastic technique. Within this simulation framework, we found indeed a linear relationship between the size of the change and the re-verification effort. Surprisingly, the contract-based approach performs better for changes below 15% of the total system size, independent of the size of the system. This rather low threshold is a pessimistic assessment, since the simulation calculated the status of the verification activities after each change performed to an element to always determine the exact propagation of a change. In a real-life situation multiple changes would be conducted in one chunk before running the analysis again. Still, even in this setting, the contract-based approach could save more than 70% of the effort compared with the standard approach. Another interesting result of the evaluation is the fact that the approaches demand a different quality of the verification activities. While the contract-based approach requires formal methods to reach a very high confidence in the results of the verification activities, the standard approach does not benefit from an increased accuracy of the tests. The increase in the overall detection rate is diminished substantially beyond an individual test accuracy of 60%. On the other hand, the contract-based approach delivers poor results if the verification accuracy is below 90%.

Furthermore, in extent to the theoretical achievements we have implemented prototypes to evaluate how a change impact analysis service could be integrated in modern distributed development environments, synchronizing huge numbers of developers (see Section 6.1). Our prototype is based on a reference technology platform (RTP) which uses an OSLC compliant data exchange mechanism. We extended this RTP towards change management capabilities and tested the handling of such a service with positive results.

7.2 Evaluation of Success Criteria

Five success criteria were defined in Section 1.2 detailing the scientific question of the thesis:

1. *The effort to determine that a system is still safe after a change is incorporated has a linear relation to the number of development artifacts that have been changed.*
   Within the stochastic simulation framework we found a linear relation between the size of the change and the re-verification effort.

2. *The confidence in the safety of the system after the change is incorporated is identical or higher compared with the current practiced approach.*
   The evaluation has shown that the detection rate of the newly developed change impact approach is higher than the currently used technique of a complete re-verification of the system, if not more than 15% of the whole system have been changed. Still, this value is a pessimistic-bound and may be significantly larger in practice.

3. *The support of the engineer during the adaptation of the system as well as all used analysis techniques are fully automated.*
   We have demonstrated that all necessary verification activities needed for the impact analysis can be automated. The refinement analysis for the new safety contracts is implemented, and for the satisfaction analysis all needed automation steps are described, though not completely implemented. Still, a framework to perform satisfaction testing manually exists. Furthermore, the impact process itself is implemented and evaluated in an automatic prototype.

4. *The developed impact analysis approach is easy to apply in practice. In particular, guidance for the engineers is available.*
   We evaluated the applicability for all elements of the approach that are directly used by the engineers. The interface to the process support (the ReMain tool) allows a simple and intuitive representation of the affected system elements. The used specification mechanism uses natural language phrasing and the provided templates allow a specification of safety concepts without a long training period. In addition we have provided design guidelines on how to build an initial system suitable for a contract-based change impact analysis.

5. *The approach is in line with the current automotive safety standard ISO 26262.*
   The selection of the capabilities that are integrated in the safety contracts were especially developed to support the functional safety concept of the ISO 26262. An ontology with the needs of the FSC has been created and fully implemented in the system specification language.

7.3 Weaknesses

The developed change impact analysis approach was explicitly designed to be applied to functional safety concepts. In this application area the approach performs well and all claimed properties hold. Still, extending the approach towards technical system descriptions will introduce new assumptions. The approach can contain changes since all design aspects (see Section 2.2.2) are modeled. For these aspects side effects can be calculated and detected. It is necessary to know all relevant aspects to completely design a perspective. For logical system descriptions the needed aspects are typically easy to determine, in most cases the aspects *functional*, *safety* and *timing* are sufficient. For some technical systems it is also possible to exclude some aspects. For example, heat problems typically do not occur in low-power devices. But this assumption introduces an additional risk. Furthermore, for many of these aspects no contract-based description language is available, like electromagnetic interference. Hence, it is unlikely that the approach can be applied to non-logical descriptions in the near future.

Furthermore, the approach suffers as do nearly all formal verification approaches, from a limitation in the size of problems able to be analyzed. Still, the modularization in the approach reduces this problem, since only fractions of the system are analyzed, in contrast to many other approaches (see Section 4.1). However, a large branching factor of the requirements will potentially lead to reaching the computational limits of the refinement analysis. Since we did not focus on providing a very performant check, we cannot quantify this limit yet. Nevertheless, we could compute refinements with a branching factor of 9 within seconds. With a more optimized refinement analysis, problems of realistic size are feasible.

7.4 Future Research Topics

There are still open research topics, which were excluded by the scope of this thesis defined in Section 1.4. One of the limitations already mentioned in the previous section is the performance of the refinement analysis. The currently chosen approach of classical model-checking using an LTL formula was easy to implement and delivered the needed results, but other more powerful techniques will most likely increase the performance drastically. Bounded model checking approaches seem most promising, since it is likely to determine a safe upper bound, which is given by the FTTI.

Furthermore, the safety-contract-based specification can be extended towards probabilistic malfunction definitions. This would allow to define a distribution for the occurence of the malfunctions and quantify the correctness of the results on the output ports.

Also the evaluating simulation can be extended in multiple ways. First, it could be evaluated how different test accuracies used for the refinement and the satisfaction analysis influence the results. Second, based on industrial example data the forming of regions which are simultaneously modified could be integrated to evaluate the shift in the break-even point. Third, if the performance of the analyses has been identified in more detail, an estimation about run-times and computational effort with more accurate units of measure can be possible. In addition, it could be evaluated if it is beneficial for the overall detection rate of the impact analysis approach to vary the test accuracy over time. Especially for the contract-based approach, it is promising to start with higher test accuracy and decrease it at elements distinct to the origin of the change.

Bibliography

Adelard. (1998). *Adelard safety case development manual.* Northampton Square, London.

Adler, R., Domis, D., Höfig, K., Kemmann, S., Kuhn, T., Schwinn, J.-P., & Trapp, M. (2011). Integration of component fault trees into the uml. In J. Dingel & A. Solberg (Eds.), *Models in software engineering* (Vol. 6627, pp. 312–327). Lecture Notes in Computer Science. Springer Berlin Heidelberg

Alpern, B. & Schneider, F. B. (1985). Defining liveness. *Information processing letters*, *21*(4), 181–185.

Ambler, S. W. (2002). *Agile modeling: effective practices for extreme programming and the unified process.* John Wiley & Sons.

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., ... Powell, D. (1990, February). Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, *16*(2), 166–182.

Armengaud, E., Bourrouilh, Q., Griessnig, G., Martin, H., & Reichenpfader, P. (2012). Using the cesar safety framework for functional safety management in the context of iso 26262. ERTS.

Armengaud, E., Biehl, M., Bourrouilh, Q., Breunig, M., Farfeleder, S., Hein, C., ... Zoier, M. (2012). Integrated tool-chain for improving traceability during the development of automotive systems. In *Proceedings of the 2012 embedded real time software and systems conference.*

Arnold, R. & Bohner, S. (1993, September). Impact analysis-towards a framework for comparison. In *Proceedings of the conference on software maintenance 1993 (csm-93)* (pp. 292–301).

ARP 4761. (1996). *Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment.* Aerospace Recommended Practice.

Arts, T., Dorigatti, M., & Tonetta, S. (2014). Making implicit safety requirements explicit. In A. Bondavalli & F. Di Giandomenico (Eds.), *Computer safety, reliability, and* *security* (Vol. 8666, pp. 81–92). Lecture Notes in Computer Science. Springer International Publishing

Atego. (2012, November). Atego workbench. http://www.atego.com/products/atego-workbench/.

ATESST2 Consortium. (2010, June). *East-adl domain model specification* (tech. rep. No. Version 2.1 RC3).

Atkinson, C. & Kühne, T. (2001). The essence of multilevel metamodeling. In M. Gogolla & C. Kobryn (Eds.), *Uml 2001 – the unified modeling language. modeling languages, concepts, and tools* (Vol. 2185, pp. 19–33). Lecture Notes in Computer Science. Springer Berlin Heidelberg

AUTOSAR GbR. (2010). *Specification of the virtual functional bus.* Version 1.0.1.

AUTOSAR GbR. (2014). *Layered software architecture.* Release 4.2.1.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, *1*, 11–33.

Baier, C. & Katoen, J.-P. (2008). *Principle of model checking.* The MIT Press.

Baldwin, C. Y. & Clark, K. B. (2000). *Design rules: the power of modularity.* MIT press.

Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., ... Weiser, J. (2013). DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)* (Vol. 8044, pp. 863–868). LNCS. Springer.

Barr, M. (2013, October). Bookout v. toyota: 2005 camry l4 software analysis.

Baufreton, P., Blanquart, J., Boulanger, J., Delseny, H., Derrien, J., Gassino, J., ... Ricque, B. (2010). Multi-domain comparison of safety standards. In *Proceedings of the 2010 embedded real time software and systems conference.*

Baumgart, A. (2013, April). A contract-based installation methodology for safety–related automotive systems. In *Technical papers presented at sae 2013 world congress & exhibition.* SAE

Baumgart, A., Böde, E., Büker, M., Damm, W., Ehmen, G., Gezgin, T., ... Weber, R. (2011, March). *Architecture modeling.* OFFIS. Retrieved from http://ses.informatik.uni-oldenburg.de/download/bib/paper/OFFIS-TR2011_ArchitectureModeling.pdf

Baumgart, A., Böde, E., Ellen, C., Peikenkamp, T., Sieverding, S., Sordon, N., … Tiran, S. (2013, July). Mbat deliverable d_wp3.1_1_3: meta models for rtp v2. Project internal Deliverable.

Baumgart, A., Hörmaier, K., & Deuter, G. (2014, October). Model-based method to achieve EMC for distributed safety-relevant automotive systems. In *Proceedings of the simul 2014, the sixth international conference on advances in system simulation* (pp. 263–270).

Baumgart, A., Reinkemeier, P., Rettberg, A., Stierand, I., Thaden, E., & Weber, R. (2010, October). A model-based design methodology with contracts to enhance the development process of safety-critical systems. In S. L. Min, R. Pettit, P. Puschner, & T. Ungerer (Eds.), *Software technologies for embedded and ubiquitous systems 8th ifip wg 10.2 international workshop, seus 2010, waidhofen/ybbs, austria, october 2010, proceedings* (pp. 59–70). LNCS 6399. Springer.

Benveniste, A., Caillaud, B., Malot, M., Adedjouma, M., Bogush, R., Velu, J.-P., … Sinha, R. (2011, November). Completeness/consistency/correctness d_sp2_r3.3_m3_vol4. http://cesarproject.eu/fileadmin/user_upload/CESAR_D_SP2_R3.3_M3_Vol4_v1.000_PU.pdf.

Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., & Sofronis, C. (2008). Multiple viewpoint contract-based specification and design. In F. de Boer, M. Bonsangue, S. Graf, & W.-P. de Roever (Eds.), *Formal methods for components and objects* (Vol. 5382, pp. 200–225). Lecture Notes in Computer Science. Springer Berlin Heidelberg

Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., … Larsen, K. (2012). *Contracts for systems design.* Research Centre Rennes - Bretagne Atlantique.

Biehl, M., DeJiu, C., & Törngren, M. (2010). Integrating safety analysis into the model-based development toolchain of automotive embedded systems.

Bishop, P. & Bloomfield, R. (1997). The ship safety case approach: a combination of system and software methods. In R. Shaw (Ed.), *Safety and reliability of software based systems* (pp. 107–121). Springer London

Böde, E., Gebhardt, S., & Peikenkamp, T. (2010). Contract based assessment of safety critical systems. In *Proceeding of the 7th european systems engineering conference (eusec 2010).*

Boehm, B. & Basili, V. R. (2005). Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, *426*.

Bohner, S. (2002, December). Extending software change impact analysis into cots components. In *Software engineering workshop, 2002. proceedings. 27th annual nasa goddard/ieee* (pp. 175–182).

Bohner, S. A. (1996). Software change impact analysis. *IEEE Computer Society.*

Bondavalli, A. & Simoncini, L. (1990). Failure classification with respect to detection. In *Distributed computing systems, 1990. proceedings., second ieee workshop on future trends of* (pp. 47–53).

Born, M., Favaro, J., & Kath, O. (2010). Application of iso dis 26262 in practice. In *Proceedings of the 1st workshop on critical automotive applications: robustness &#38; safety* (pp. 3–6). CARS '10. Valencia, Spain: ACM

Bozzano, M. & Villafiorita, A. (2003). Improving system reliability via model checking: the fsap/nusmv-sa safety analysis platform. In *Computer safety, reliability, and security.* Springer Berlin Heidelberg.

Bozzano, M. & Villafiorita, A. (2011). *Design and safety assesment of critical systems.* Auerbach Publications.

Broy, M. (2006). Challenges in automotive software engineering. In *Proceedings of the 28th international conference on software engineering* (pp. 33–42). ACM.

Büchner, F., Glöe, D. G., & Mainka, E.-U. (2003, January). *Hitex white paper: using riskcat to cope with iec 61508.* Hitex Developent Tool GmbH.

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005, September). Towards a taxonomy of software change: research articles. *Journal of Software Maintenance and Evolution: Reserach and Practice*, *17*(5), 309–332

Casais, E. (1994). The automatic reorganization of object oriented hierarchies - a case study.

CBSNEWS. (2010, May). Toyota "unintended acceleration"has killed 89. Retrieved from %5Curl%7Bhttp://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/%7D

Ciolkowski, M., Laitenberger, O., & Biffl, S. (2003, November). Software reviews: the state of the practice. *IEEE Software*, *20*(6), 46–51

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). *Model checking.* MIT Press.

Clarkson, P., Simons, C., & Eckert, C. (2004). Predicting change propagation in complex design. *Journal of Mechanical Design (Transactions of the ASME)*, *126*(5), 788–797.

CMMI Product Team. (2010, November). Cmmi for development, version 1.3: improving processes for developing better products and services. http://www.sei.cmu.edu/reports/10tr033.pdf.

Damm, W. (2005, June). Controlling speculative design processes using rich component models. In *Application of concurrency to system design, 2005. acsd 2005. fifth international conference on* (pp. 118–119).

Damm, W., Dierks, H., Oehlerking, J., & Pnueli, A. (2010). Towards component based design of hybrid systems: safety and stability. In Z. Manna & D. Peled (Eds.), *Time for verification* (Vol. 6200, pp. 96–143). Lecture Notes in Computer Science. Springer Berlin Heidelberg

Damm, W., Hungar, H., Josko, B., Peikenkamp, T., & Stierand, I. (2011, March). Using contract-based component specifications for virtual integration testing and architecture design. In *Design, automation test in europe conference exhibition (DATE), 2011* (pp. 1–6).

Damm, W., Josko, B., & Peikenkamp, T. (2009). *Contract based iso cd 26262 safety analysis.* SAE Technical Paper.

Dassault Systems. (2012, November). Reqtify. http://www.3ds.com/products/catia/portfolio/geensoft/reqtify/.

De Alfaro, L. & Henzinger, T. A. (2001). Interface automata. *ACM SIGSOFT Software Engineering Notes*, *26*(5), 109–120.

de la Vara, J. L., Borg, M., Wnuk, K., & Moonen, L. (2014). *Survey on safety evidence change impact analysis in practice: detailed description and analysis.* Simula Research Laboratory. Simula Research Laboratory.

de Roever, W.-P. (1998). The need for compositional proof systems: a survey. In *Compositionality: the significant difference* (pp. 1–22). Springer.

Delahaye, B., Caillaud, B., & Legay, A. (2011). Probabilistic contracts: a compositional reasoning methodology for the design of systems with stochastic and/or non-deterministic aspects. *Formal Methods in System Design*, *38*(1), 1–32

Dick, J. (2005, November). Design traceability. *Software, IEEE*, *22*(6), 14–16.

DO 178C. (2011, December). *Software considerations in airborne systems and equipment certification.* RTCA, Inc.

DoDAF/DM2 2.02. (2010, August). *The dodaf architecture framework version 2.02.* U.S. Department of Defence.

Dröschel, W. & Wiemers, M. (1999). Das v-modell 97. Oldenbourg.

Echtle, K. (1990). *Fehlertoleranzverfahren.* Springer-Verlag.

Eckert, C., Clarkson, P., & Zanker, W. (2004). Change and customisation in complex engineering domains. *Research in Engineering Design*, *15*, 1–21

Eckert, C., Weck, O., Keller, R., & Clarkson, P. J. (2009). Engineering change: drivers, sources, and approaches in industry. In *International conference on engineering design, iced'09.*

Ellen, C., Etzien, C., & Oertel, M. (2012). Integration of automatic allocation solving in a perspective oriented systems design process. In *Proceedings of the date 2012 conference.*

Ellims, M., Bridges, J., & Ince, D. (2006). The economics of unit testing. *Empirical Software Engineering*, *11*(1), 5–31

EN 50129. (2003). *Railway applications – Communication, signalling and processing systems – Safety related electronic systems for signalling.* European Commitee for Electrotechnical Standardization.

Enzmann, M., Döhmen, G., Andersson, H., & Härdt, C. (2008, April). *Speeds methodology - a white paper.*

Espinoza, H., Ruiz, A., Sabetzadeh, M., & Panaroni, P. (2011). Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems. In *Software certification (wosocer), 2011 first international workshop on* (pp. 1–6). IEEE.

Fenn, J., Hawkins, R., Williams, P., Kelly, T., Banner, M., & Oakshott, Y. (2007). The who, where, how, why and when of modular and incremental certification. In *System safety, 2007 2nd institution of engineering and technology international conference on* (pp. 135–140). IET.

Firesmith, D. G. (2003). *Common concepts underlying safety, security, and survivability engineering.* Carnegie Mellon University.

Föster, M. (2012, Oktober). *Dependable reuse & guarded integration of automotive software components* (tech. rep. No. ESL-2012/FAT-P). RWTH Embedded software laboratory.

Fowler, M. (2002, November). *Patterns of enterprise application architecture.* Boston, MA, USA: Addison Wesley.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: improving the design of existing code.* Boston, MA, USA: Addison-Wesley Longman Publishing.

Fricke, E., Gebhard, B., Negele, H., & Igenbergs, E. (2000). Coping with changes: causes, findings, and strategies. *Systems Engineering*, *3*(4), 169–179.

Friedrich, J., Hammerschall, U., Kuhrmann, M., & Sihling, M. (2009). Das v-modell xt. In *Das v-modell xt* (pp. 1–32). Informatik im Fokus. Springer Berlin Heidelberg

Gallagher, K. & Lyle, J. (1991, August). Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, *17*(8), 751–761.

Gebhardt, V., Rieger, G. M., Mottok, J., & Gießelbach, C. (2013). *Funktionale sicherheit nach iso 26262: ein praxisleitfaden zur umsetzung.* dpunkt.verlag GmbH.

Gezgin, T., Weber, R., & Oertel, M. (2014). Multi-aspect virtual integration approach for real-time and safety properties. In *Proceedings of the international workshop on design and implementation of formal tools and systems 2014*.

Gotel, O. & Finkelstein, C. (1994, April). An analysis of the requirements traceability problem. In *Requirements engineering, 1994, proceedings of the first international conference on* (pp. 94–101).

Gould, J., Glossop, M., & Ioannides, A. (2000). *Review of hazard identification techniques.* Health and Safety Laboratory.

Graaf, B., Lormans, M., & Toetenel, H. (2003). Embedded software engineering: the state of the practice. *IEEE Software*, *20*(6), 61–69.

GSN Community Standard. (2011, November). *Version 1.* Origin Consulting Limited.

Gustafsson, J., Ermedahl, A., Sandberg, C., & Lisper, B. (2006, December). Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Real-time systems symposium, 2006. rtss '06. 27th ieee international* (pp. 57–66).

Hassan, A. & Holt, R. (2004, September). Predicting change propagation in software systems. In *Software maintenance, 2004. proceedings. 20th ieee international conference on* (pp. 284–293).

Henzinger, T., Qadeer, S., & Rajamani, S. (1998). You assume, we guarantee: methodology and case studies. In A. Hu & M. Vardi (Eds.), *Computer aided verification* (Vol. 1427, pp. 440–451). Lecture Notes in Computer Science. Springer Berlin Heidelberg

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, *12*(10), 576–580.

Huang, G. & Mak, K. (1999). Current practices of engineering change management in uk manufacturing industries. *International Journal of Operations & Production Management*, *19*(1), 21–37. eprint: http://dx.doi.org/10.1108/01443579910244205

Humphrey, W. S. (2000). *Introduction to the team software process (sm).* Addison-Wesley Professional.

Hungar, H. (2011a, September). *Components and contracts: a semantical foundation for compositional refinement.*

Hungar, H. (2011b, November). Compositionality with strong assumptions. In *Nordic workshop on programming theory* (pp. 11–13). Mälardalen Real–Time Research Center.

IBM. (2012, November). Rational change. http://www-01.ibm.com/software/awdtools/change/.

IEEE 1471. (2000). *Recommended practice for architectural description of software-intensive systems.* Institute of Electrical and Electronics Engineers.

ikv++ technologies ag. (2010, May). Medinitm analyze: functional safety analysis for iso 26262. http://www.ikv.de/index.php?option=com_docman&task=doc_download&gid=41.

ISO 10007. (2003, June). *Quality management systems – guidelines for configuration management.* International Organization for Standardization.

ISO 26262. (2011, November). *Road vehicles – functional safety.* International Organization for Standardization.

ISO/IEC. (2011, November). Iso/iec 29148: systems and software engineering - life cycle processes - requirements engineering.

ISO/IEC 12207. (2008, February). *Systems and software engineering – software life cycle processes.* International Organization for Standardization.

Jarratt, T., Eckert, C., Caldwell, N., & Clarkson, P. (2011). Engineering change: an overview and perspective on the literature. *Research in Engineering Design*, *22*, 103–124

Jarratt, T. A. W. (2004). *A model-based approach to support the management of engineering change* (Doctoral dissertation, Cambridge University Engineering Department).

Jiang, H. (2013). *Key findings on airplane economic life.* Boeing Commercial Airplanes. Retrieved from %5Curl%7Bhttp://www.boeing.com/assets/pdf/commercial/aircraft_economic_life_whitepaper.pdf%7D

Joshi, A. & Heimdahl, M. P. E. (2005). Model-based safety analysis of simulink models using scade design verifier. In *Proceedings of the 24th international conference on computer safety, reliability, and security.* SAFECOMP'05. Fredrikstad, Norway.

Kacimi, O., Ellen, C., Oertel, M., & Sojka, D. (2014, January). Creating a reference technology platform:performing model-based safety analysis in a heterogeneous development environment. In *Proceedings of modelsward 2014* (pp. 645–652). SCITEPRESS.

Kaiser, B., Liggesmeyer, P., & Mäckel, O. (2003). A new component concept for fault trees. In *Proceedings of the 8th australian workshop on safety critical systems and software - volume 33*. SCS '03. Canberra, Australia: Australian Computer Society.

Kececioglu, D. (1991). *Reliability engineering handbook: volume i*. PTR Prentice Hall, Englewood Cliffs, New Jersey.

Kelion, L. (2015, May). Airbus a400m plane crash linked to software fault.

Kelly, T. & Weaver, R. (2004). The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*. Citeseer.

Kelly, T. (1999). *Arguing safety: a systematic approach to managing safety cases* (Doctoral dissertation, University of York).

Kelly, T. (1997). A six-step method for the development of goal structures. *York Software Engineering, Flixborough, UK*.

Kidd, M. & Thompson, G. (2000). Engineering design change management. *Integrated Manufacturing Systems*, *11*(1), 74–77.

Kilpinen, M. S. (2008). *The emergence of change at the systems engineering and software design interface* (Doctoral dissertation, University of Cambridge).

Klug, K. & Tugenberg, S. (1993). *Functional lockstep arrangement for redundant processors*. 5,226,152. Google Patents. Retrieved from %5Curl%7Bhttps://www.google.com/patents/US5226152%7D

Koren, I. & Krishna, C. M. (2007). *Fault-tolerant systems*. Elsevier, Morgan Kaufmann.

Korpi, J. & Koskinen, J. (2007). Supporting impact analysis by program dependence graph based forward slicing. In K. Elleithy (Ed.), *Advances and innovations in systems, computing sciences and software engineering* (pp. 197–202). Springer Netherlands

Krasner, G. E., Pope, S. T. et al. (1988). A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, *1*(3), 26–49.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2), 125–143.

Lamport, L. & Merz, S. (1994). Specifying and verifying fault-tolerant systems. In *Formal techniques in real-time and fault-tolerant systems* (pp. 41–76). Springer.

Laprie, J.-C. (1994). Dependability: the challenge for the future of computing and communication technologies. In *Dependable computing - edcc-1* (pp. 405–408). Springer.

Law, J. & Rothermel, G. (2003). Whole program path-based dynamic impact analysis. In *Proceedings of the 25th international conference on software engineering* (pp. 308–318). ICSE '03. Portland, Oregon: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=776816.776854

Lee, E. A. & Seshia, S. A. (2010). *Introduction to embedded systems - a cyber-physical systems approach* (1st ed.). Lee and Seshia. Retrieved from http://chess.eecs.berkeley.edu/pubs/794.html

Lehnert, S. (2011). *A review of software change impact analysis*. Ilmenau University of Technology.

Lehnert, S., Farooq, Q., & Riebisch, M. (2013). Rule-based impact analysis for heterogeneous software artifacts. In *Software maintenance and reengineering (csmr), 2013 17th european conference on* (pp. 209–218). IEEE.

Leveson, N. G. & Weiss, K. A. (2004). Making embedded software reuse practical and safe. In *Proceedings of the 12th acm sigsoft twelfth international symposium on foundations of software engineering* (pp. 171–178). SIGSOFT '04/FSE-12. Newport Beach, CA, USA: ACM

Li, J., Pu, G., Zhang, L., Vardi, M. Y., & He, J. (2014). Fast ltl satisfiability checking by sat solvers. *CoRR*, *abs/1401.5677*.

Lisagor, O., McDermid, J., & Pumfrey, D. (2006). Towards a practicable process for automated safety analysis. In *24th international system safety conference* (pp. 596–607). Citeseer.

Lisagor, O. (2010). *Failure logic modelling: a pragmatic approach* (Doctoral dissertation, University of York).

Lock, S. & Kotonya, G. (1999). An integrated, probabilistic framework for requirement change impact analysis. *Australasian Journal of Information Systems*, *6*(2). Retrieved from http://journal.acs.org.au/index.php/ajis/article/view/292

Mäckel, O. & Rothfelder, M. (2001). Challenges and solutions for fault tree analysis arising from automatic fault tree generation: some milestones on the way. In *Proceedings of the world multiconference on systemics, cybernetics and informatics: information systems development-volume i - volume i* (pp. 583–588). ISAS-SCI '01. IIIS. Retrieved from http://dl.acm.org/citation.cfm?id=646789.704234

Madslien, J. (2011, December). Gm chevrolet volt: buyers spooked by electric car fires.

Mandrioli, D. & Meyer, B. (Eds.). (1992). *Advances in object-oriented software engineering*. Upper Saddle River, NJ, USA: Prentice-Hall.

Marwedel, P. & Wehmeyer, L. (2007). *Eingebettete systeme*. Springer London. Retrieved from http://books.google.de/books?id=94L961HZ5lYC

MathWorks. (2011, December). *Iec certification kit for iso 26262 and iec 61508*. Retrieved from %5Curl%7Bhttp://www.mathworks.de/products/iec-61508/index.html%7D

McDermid, J. & Pumfrey, D. (1994, June). A development of hazard analysis to aid software design. In *Computer assurance, 1994. compass '94 safety, reliability, fault tolerance, concurrency and real time, security. proceedings of the ninth annual conference on* (pp. 17–25).

Mens, T. & Tourwe, T. (2004, February). A survey of software refactoring. *Software Engineering, IEEE Transactions on*, *30*(2), 126–139.

Meyer, B. (1992). Applying design by contract. *IEEE Computer*, *25*, 40–51.

Mitschke, A., Loughran, N., Josko, B., Oertel, M., Rehkop, P., Häusler, S., & Benveniste, A. (2010). *RE Language Definitions to formalize multi-criteria requirements V2*. The CESAR Consortium. Retrieved from http://cesarproject.eu/fileadmin/user%5C_upload/CESAR%5C_D%5C_SP2%5C_%20R2.2%5C_M2%5C_v1.000.pdf

Moore, E. & Shannon, C. (1956). Reliable circuits using less reliable relays. *Journal of the Franklin Institute*, *262*(3), 191–208

Nancy. (1995). *Safeware* (H. Goldstein, Ed.). Addison-Wesley.

Neumann, P. (1986, September). On hierarchical design of computer systems for critical applications. *Software Engineering, IEEE Transactions on*, *SE-12*(9), 905–920.

Nicholson, M., Conmy, P., Bate, I., & McDermid, J. (2000). Generating and maintaining a safety argument for integrated modular systems. In *Adelard for the health and safety executive, hse books, isbn 0-7176-2010-7, and contract research.*

Nitsche, G., Gruttner, K., & Nebel, W. (2013, September). Power contracts: a formal way towards power-closure?! In *Power and timing modeling, optimization and simulation (patmos), 2013 23rd international workshop on* (pp. 59–66).

Nuseibeh, B., Easterbrook, S., & Russo, A. (2000, April). Leveraging inconsistency in software development. *Computer*, *33*(4), 24–29.

Nuseibeh, B. & Easterbrook, S. (2000). Requirements engineering: a roadmap. In *Proceedings of the conference on the future of software engineering* (pp. 35–46). ICSE '00. Limerick, Ireland: ACM

Oertel, M., Battram, P., Kacimi, O., Gerwinn, S., & Rettberg, A. (2015). A compositional safety specification using a contract-based design methodology. In W. Leister & N. Regnesentral (Eds.), *Pesaro 2015: the fifth international conference on performance, safety and robustness in complex systems and applications* (pp. 1–7). IARIA.

Oertel, M., Gerwinn, S., & Rettberg, A. (2014, July). Simulative evaluation of contract-based change management. In *Industrial informatics (indin), 2014 12th ieee international conference on* (pp. 16–21).

Oertel, M. & Josko, B. (2012). Interoperable requirements engineering: tool independent specification, validation and impact analysis. In *Artemis technology conference 2012.*

Oertel, M., Kacimi, O., & Böde, E. (2014). Proving compliance of implementation models to safety specifications. In A. Bondavalli, A. Ceccarelli, & F. Ortmeier (Eds.), *Computer safety, reliability, and security* (Vol. 8696, pp. 97–107). Lecture Notes in Computer Science. Springer International Publishing

Oertel, M., Mahdi, A., Böde, E., & Rettberg, A. (2014). Contract-based safety: specification and application guidelines. In *Proceedings of the 1st international workshop on emerging ideas and trends in engineering of cyber-physical systems (eitec 2014).*

Oertel, M., Malot, M., Baumgart, A., Becker, J., Bogusch, R., Farfeleder, S., . . . Rehkop, P. (2013). Requirements engineering. In A. Rajan & T. Wahl (Eds.), *Cesar - cost-efficient methods and processes for safety-relevant embedded systems* (pp. 69–143). Springer Vienna

Oertel, M. & Rettberg, A. (2013). Reducing re-verification effort by requirement-based change management. In G. Schirner, M. Götz, A. Rettberg, M. Zanella, & F. Rammig (Eds.), *Embedded systems: design, analysis and verification* (Vol. 403, pp. 104–115). IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg

OMG SysML. (2012). *OMG Systems Modeling Language, Version 1.3.* Object Management Group. Retrieved from http://www.omg.org/spec/SysML/1.3/

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks* (Doctoral dissertation, University of Illinois at Urbana-Champaign).

OSLC: Open Services for Lifecycle Collaboration. (2012, December). http://open-services.net/.

Papadopoulos, Y. & Maruhn, M. (2001, July). Model-based synthesis of fault trees from matlab-simulink models. In *Dependable systems and networks, 2001. dsn 2001. international conference on* (pp. 77–82).

Pasquini, A., Papadopoulos, Y., & McDermid, J. (1999). Hierarchically performed hazard origin and propagation studies. In K. Kanoun (Ed.), *Computer safety, reliability and security* (Vol. 1698, pp. 688–688). Lecture Notes in Computer Science. 10.1007/3-540-48249-0_13. Springer Berlin / Heidelberg. Retrieved from http://dx.doi.org/10.1007/3-540-48249-0%5C_13

Paynter, S., Armstrong, J., & Haveman, J. (2000). Adl: an activity description language for real-time networks. *Formal Aspects of Computing*, *12*(2), 120–144

Peikenkamp, T., Cavallo, A., Valacca, L., Bödede, E., Pretzer, M., & Hahn, E. M. (2006). Towards a unified model-based safety assessment. In *Proceedings of safecomp* (pp. 275–288).

Peng, H. & Tahar, S. (1998). *A survey on compositional verification.*

Pnueli, A. (1977). The temporal logic of programs. In *Foundations of computer science, 1977., 18th annual symposium on* (pp. 46–57). IEEE.

Podgurski, A. & Clarke, L. (1990, September). A formal model of program dependences and its implications for software testing, debugging, and maintenance. *Software Engineering, IEEE Transactions on*, *16*(9), 965–979.

Pohl, K., Hönninger, H., Achatz, R., & Broy, M. (Eds.). (2012). *Model-based engineering of embedded systems: the spes 2020 methodology.* Springer-Verlag Berlin Heidleberg.

Prisaznuk, P. (1992, May). Integrated modular avionics. In *Aerospace and electronics conference, 1992. naecon 1992., proceedings of the ieee 1992 national* (39–45 vol.1).

Rajan, A. & Wahl, T. (Eds.). (2013). *Cesar - cost-efficient methods and processes for safety-relevant embedded systems.* Springer Vienna.

Ramesh, B., Powers, T., Stubbs, C., & Edwards, M. (1995, March). Implementing requirements traceability: a case study. In *Requirements engineering, 1995., proceedings of the second ieee international symposium on* (pp. 89–95).

Reinkemeier, P., Stierand, I., Rehkop, P., & Henkler, S. (2011). A pattern-based requirement specification language: mapping automotive specific timing requirements. In R. Reussner, A. Pretschner, & S. Jähnichen (Eds.), *Software engineering 2011 workshopband* (pp. 99–108). Gesellschaft für Informatik e.V. (GI).

Robertson, S. & Robertson, J. (1999). *Mastering the requirements process.* Addison Wesley.

Ross, H.-L. (2014). *Funktionale sicherheit im automobil: iso 26262, systemengineering auf basis eines sicherheitslebenszyklus und bewährten managementsystemen.* Carl Hanser Verlag GmbH & Co. KG.

Rozier, K. Y. & Vardi, M. Y. (2007). Ltl satisfiability checking. In D. Bosnacki & S. Edelkamp (Eds.), *Spin* (Vol. 4595, pp. 149–167). Lecture Notes in Computer Science. Springer.

Runeson, P., Andersson, C., Thelin, T., Andrews, A., & Berling, T. (2006, May). What do we know about defect detection methods? [software testing]. *Software, IEEE*, *23*(3), 82–90.

Ryder, B. G. & Tip, F. (2001). Change impact analysis for object-oriented programs. In *Proceedings of the 2001 acm sigplan-sigsoft workshop on program analysis for software tools and engineering* (pp. 46–53). Snowbird, Utah, USA

Ryder, B. (1979, May). Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, *SE-5*(3), 216–226.

Sage, A. P. & Rouse, W. B. (2009). *Handbook of systems engineering and management* (2nd Edition). John Wiley & Sons.

Santelices, R. & Harrold, M. J. (2010). *Probabilistic slicing for predictive impact analysis.* Georgia Institute of Technology.

Schäfer, A. (2003). Combining real-time model-checking and fault tree analysis. In *Fme 2003: formal methods.* Springer Berlin Heidelberg.

Schätz, B., Pretschner, A., Huber, F., & Philipps, J. (2002). Model-based development of embedded systems. In J.-M. Bruel & Z. Bellahsene (Eds.), *Advances in object-oriented information systems* (Vol. 2426, pp. 298–311). Lecture Notes in Computer Science. Springer Berlin Heidelberg

Sharvia, S. & Papadopoulos, Y. (2008). Non-coherent modelling in compositional fault tree analysis. *The International Federation of Automatic Control, Seoul*, 6–11.

Shaw, A. (1989, July). Reasoning about time in higher-level language software. *Software Engineering, IEEE Transactions on*, *15*(7), 875–889.

Sljivo, I., Jaradat, O., Bate, I., & Graydon, P. (2015, January). Deriving safety contracts to support architecture design of safety critical systems. In *16th ieee international symposium on high assurance systems engineering* (pp. 126–133). IEEE. Retrieved from http://www.es.mdh.se/publications/3763-

Software, I. R. (2015). Doors. http://www-03.ibm.com/software/products/de/ratidoor.

Sommerville, I. (2010). *Software engineering* (9th ed.). Harlow, England, UK: Addison-Wesley.

Sommerville, I. & Sawyer, P. (1997). *Requirements engineering: a good practice guide.* John Wiley & Sons, Inc.

SPEEDS. (2007). SPEEDS core meta-model syntax and draft semantics. SPEEDS D.2.1.c.

Standish Group. (1995). The standish group report: CHAOS. http://www.projectsmart.co.uk/docs/chaos-report.pdf.

Steward, D. V. (1981). The design structure system: a method for managing the design of complex systems. *IEEE transactions on Engineering Management*, (EM-28).

Storey, N. (1996). *Safety-critical computer systems.* Addison-Wesley.

Svenningsson, R., Vinter, J., Eriksson, H., & Törngren, M. (2010). Modifi: a model-implemented fault injection tool. In *Computer safety, reliability, and security.* Springer Berlin Heidelberg.

Terwiesch, C. & Loch, C. H. (1999). Managing the process of engineering change orders: the case of the climate control system in automobile development. *Journal of Product Innovation Management*, *16*(2), 160–172.

Tip, F. & Palsberg, J. (2000). Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th acm sigplan conference on object-oriented programming, systems, languages, and applications* (pp. 281–293). OOPSLA '00. Minneapolis, MN, USA: ACM

Vidács, L., Beszédes, Á., & Ferenc, R. (2007). Macro impact analysis using macro slicing. In *Icsoft (se)* (pp. 230–235).

Visser, W., Havelund, K., Brat, G., Park, S., & Lerda, F. (2003). Model checking programs. *Automated Software Engineering*, *10*(2), 203–232

Von Neumann, J. (1956). Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, *34*, 43–98.

Walker, M. & Papadopoulos, Y. (2008). Synthesis and analysis of temporal fault trees with pandora: the time of priority and gates. *Nonlinear Analysis: Hybrid Systems*, *2*(2), 368–382.

Walker, M. & Papadopoulos, Y. (2009). Qualitative temporal analysis: towards a full implementation of the fault tree handbook. *Control Engineering Practice*, *17*(10), 1115–1125.

Wallace, M. (2005). Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, *141*(3), 53–71.

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th international conference on software engineering* (pp. 439–449). ICSE '81. San Diego, CA, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=800078.802557

Westman, J., Nyberg, M., & Törngren, M. (2013). Structuring safety requirements in iso 26262 using contract theory. In F. Bitsch, J. Guiochet, & M. Kaâniche (Eds.), *Computer safety, reliability, and security* (Vol. 8153, pp. 166–177). Lecture Notes in Computer Science. Springer Berlin Heidelberg

Wright, I. (1997). A review of research into engineering change management: implications for product design. *Design Studies*, *18*(1), 33–42.

Ye, F. & Kelly, T. (2004, September). Component failure mitigation according to failure type. In *Computer software and applications conference, 2004. compsac 2004. proceedings of the 28th annual international* (pp. 258–264).