Thesis for The Degree of Doctor of Philosophy

CHALMERS | UNIVERSITY OF GOTHENBURG

Software Robustness: From Requirements to Verification

Ali Shahrokni

Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology |University of Gothenburg
Göteborg, Sweden, 2013

Software Robustness:
From Requirements to Verification

Ali Shahrokni

Göteborg 2013
ISBN 978-91-7385-831-1
Doktorsavhandlingar vid
Chalmers tekniska högskola
Ny serie Nr 3512
ISSN 0346-718X

Technical report 103D
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Göteborg, Sweden 2013.

To Laleh & my family

Abstract

The importance of software quality increases as software products become more intertwined with our everyday lives. A critical software quality attribute is robustness, i.e. that the software shows stable behavior in stressful conditions and when receiving faulty inputs. Even though this has been a long-term goal in software engineering, few studies directly target robustness. The overall goal of this thesis is to identify gaps in the knowledge and take steps towards improving and creating methods to work with software robustness.

To identify gaps in the state of knowledge, this thesis first describes a systematic review of the academic literature on software robustness. The results, based on analysis of 144 relevant papers, suggest that the most prominent contributions on robustness are methods and tools for random testing on the external interfaces of systems. Another finding is the lack of empirical evidence and guidelines on how to define and specify robustness. Additionally, there is a lack of methods to elicit, analyze, and specify robustness requirements in a systematic way, and to test these requirements.

To address the goals of the thesis, we have worked with five industrial companies. We examined the state of practice by conducting interviews and analyzing requirements documents at some of our partner companies to identify improvement potential. The results show that there also is a lack of systematic methods to specify and test quality requirements in practice. Furthermore, unverifiable quality requirements are still a source of problem and high cost to software development projects.

To address these issues, we constructed a framework for analysis, elicitation, and specification of software robustness (ROAST). Based on simple models for root causes and symptoms of robustness failures, we have identified 19 patterns for robustness requirements. Further, ROAST includes a notion of specification levels that helps practitioners refine high-level requirements to a verifiable level. The framework has been evaluated using document analysis, interviews, and surveys at the partner companies. The evaluations have investigated the usefulness, quality, and generalizability of ROAST and have helped us improve the framework over time.

The last part of the thesis uses the patterns in ROAST, to specify generic robustness properties that the system should fulfill. We present a testing framework, RobusTest, that uses these properties to automatically generate robustness test cases. This provides a more focused testing than complete random testing. We have implemented and evaluated parts of this framework and found robustness issues in open source and well-tested industrial systems.

This thesis provides guidelines for and discusses how practitioners can more systematically work with robustness from requirements elicitation and analysis to testing.

Keywords

requirements specification, requirements patterns, non-functional requirements, quality requirements, robustness, requirements refinement, robustness testing

Acknowledgment

First of all, I would like to thank my main supervisor Pr. Robert Feldt for his invaluable support, guidance, encouragement, and feedback that has helped and enabled me to pursue my Ph.D. studies. I would also like to thank my supervisor Pr. Johan Karlsson for his valuable advice and support during my time as a Ph.D. candidate in capacity of examiner and supervisor.

I appreciate the time and effort my former examiner Pr. Reiner Hähnle put on this project. Dr. Peter Öhman initiated the initial project of my Ph.D studies and introduced me to the field of software engineering and I am grateful for that. I also appreciate the feedback from my secondary supervisor Dr. Thomas Thelin.

Special thanks to all of my colleagues at the division of software engineering for creating a friendly and motivating atmosphere and for their help and support. Ruben Alexandersson, Ana Magazinius, Joakim Pernståhl, Emil Alégroth, Antonio Martini, and my other colleagues, I am lucky to know you and have worked with you and grateful for all your support and encouragement during my studies and in my life.

I am very grateful for the help and support of my industrial partners at Volvo Technology AB and other companies that have given me material and resources throughout this project.

My inexpressible appreciation goes to my family and friends that have always supported me. Special thanks to Laleh for always being there for me and standing beside me with love during the tough and happy times. I thank my mother, grandmother, and sister for their unconditional love and support and my little nephew who has brought much joy and color to my life. I also want to thank Laleh's lovely family for all their help and support. I am specially grateful to my late father who always was the main source of motivation and support in my personal and professional life.

This research has been carried out in joint research projects financed by the Swedish Governmental Agency of Innovation Systems (VINNOVA) and Chalmers University of Technology. It has also been supported by the Swedish Research School in Verification and Validation (SWELL) within the VinnPro program funded by VINNOVA.

List of Publications

Appended papers

This thesis is based on the following papers:

[A] A. Shahrokni, R. Feldt "A Systematic Review of Software Robustness", *Journal of Information and Software Technology, 2013*, 55(1), pp. 1–17.

[B] A. Shahrokni, R. Feldt, F. Pettersson, A. Bäck "Robustness Verification Challenges in Automotive Telematics Software",
Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA, July 1-3, 2009, pp. 460–465.

[C] A. Shahrokni, R. Feldt "Industrial Challenges with Quality Requirements in Safety Critical Software Systems",
Technical report 2013:04 ISSN 1652-926X Chalmers University of Technology
Parts of this report has been published at: *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications, September 4-6, 2013, Santander, Spain*, pp. 78–81.

[D] A. Shahrokni, R. Feldt "ROAST: A Framework for Software Robustness Requirements Elicitation and Specification",
In Submission to Journal of Systems and Software, 2013.

[E] A. Shahrokni, R. Feldt "RobusTest: A Framework for Automated Testing of Robustness in Software",
Proceedings of the 18th Asia Pacific Software Engineering Conference (APSEC), December 05-08, 2011, Ho Chi Minh city, Vietnam, pp. 171–178.

[F] A. Shahrokni, R. Feldt "Semi-automated Robustness Testing in Industrial Context Using RobusTest",
In Submission to the 6th IEEE International Conference on Software Testing, Verification and Validation, 2014.

Other papers

The following papers are published or in revision but not appended to this thesis, either due to contents overlapping that of appended papers, or contents not related to the thesis.

[a] A. Shahrokni, R. Feldt "RobusTest: Towards a Framework for Automated Testing of Robustness in Software,"
Proceedings of the Third International Conference on Advances in System Testing and Validation Lifecycle, VALID 2011, Barcelona, Spain, October 23-29, 2011, pp. 78–83.

[b] A. Shahrokni, R. Feldt "Towards a Framework for Software Robustness Requirements based on Patterns,"
Proceedings of Requirements Engineering: Foundation for Software Quality, 16th International Working Conference, REFSQ 2010, Essen, Germany, June 30 - July 2, 2010, pp. 79-84

[c] R.B. Svensson, T. Gorschek, B. Regnell, R. Torkar, A. Shahrokni, R. Feldt, "Prioritization of Quality Requirements: State of Practice in Eleven Companies,"
Requirements Engineering Conference (RE), Trento, Italy, August 29 - September 2, 2011, pp. 69-78.

[d] R.B. Svensson, T. Gorschek, B. Regnell, R. Torkar, A. Shahrokni, R. Feldt, "Quality Requirements in Industrial Practice - an extended interview study at eleven companies,"
IEEE Transactions on Software Engineering, 2012, 38(4), pp. 923-935.

[e] R. Gabriel, J. Sandsjö, A. Shahrokni, M. Fjeld "BounceSlider: Actuated Sliders for Music Performance and Composition,"
Proceedings of the International Conference on Tangible, Embedded and Embodied Interaction 2008, TEI'08, 1 (1), pp. 127-130.

[f] J. J. Rodriguez, A. Shahrokni, M. Fjeld "One-Dimensional Force Feedback Slider: Digital Platform,"
Proceedings of the IEEE VR 2007 Workshop on Mixed Reality User Interfaces: Specification, Authoring, Adaptation (MRUI'07), pp. 47-51.

[g] A. Shahrokni, J. Jenaro, T. Gustafsson, A. Vinnberg, J. Sandsjö, M. Fjeld "One-dimensional force feedback slider: going from an analogue to a digital platform,"
Proceedings of the 4th Nordic Conference on Human-Computer interaction: Changing Roles (Oslo, Norway, October 14 - 18, 2006), NordiCHI '06. ACM Press, 189(1), pp. 453-456.

Contents

Abstract **v**

Acknowledgment **vii**

List of Publications **ix**

1 Introduction **1**
1.1 Preamble 1
1.2 Background and related work 4
1.2.1 Quality requirements & attributes 4
1.2.2 Software robustness 8
1.2.3 Requirements patterns 11
1.2.4 Refinement of quality requirements 11
1.2.5 Property-based testing 12
1.3 Research questions 14
1.4 Research methodology 16
1.4.1 Action research 16
1.4.2 Systematic literature review 19
1.4.3 Design reseach 19
1.4.4 Evaluatory methodologies 21
1.5 Research setting 24
1.6 Overview of appended papers 26
1.6.1 Paper A 26
1.6.2 Paper B 28
1.6.3 Paper C 29
1.6.4 Paper D 33
1.6.5 Paper E 37
1.6.6 Paper F 38
1.7 Discussion 41
1.7.1 How do we define robustness in different contexts and for different types of systems? *(RQ1)* 41
1.7.2 What is the state of knowledge on software robustness? *(RQ2)* 42

1.7.3 How do practitioners work with requirements, and verification and validation, especially on robustness, at different phases of development? *(RQ3)* 43
1.7.4 How can we elicit and specify robustness requirements in a systematic way? *(RQ4)* 45
1.7.5 How can robustness testing and assurance activities be aligned with the requirements? *(RQ5)* 46
1.8 Future research 48
1.9 Conclusion 49

2 Paper A **51**
2.1 Introduction 52
2.2 Related work 53
2.3 Research methodology 54
2.3.1 Research questions 55
2.3.2 Sources of information 56
2.3.3 Search criteria 56
2.3.4 Study selection 57
2.3.5 Data extraction and synthesis 58
2.3.6 Threats to validity 62
2.4 Results and analysis 62
2.4.1 Phase focus of studies 63
2.4.2 System focus 72
2.4.3 Quality of research/contribution 74
2.5 Discussion 77
2.6 Conclusion 79

3 Paper B **81**
3.1 Introduction 82
3.2 Background 82
3.2.1 Dependability and robustness definition 82
3.2.2 Goals and trends in telematics software systems in the automotive industry 83
3.2.3 Open standards and architectures 84
3.3 State of automotive telematics verification practice 85
3.3.1 State of practice 85
3.4 Challenges 87
3.4.1 3rd party components 87
3.4.2 Variation in deployment 88
3.5 Towards solutions 89
3.5.1 Reviews 89
3.5.2 Certification 89
3.5.3 Alignment of requirements and verification 89
3.5.4 Formal methods 89
3.5.5 Automated testing tools for robustness testing 90
3.6 Conclusion and future work 91

4 Paper C **93**
4.1 Introduction . . . 94
4.2 Background and related work . . . 95
4.3 Case company description . . . 97
4.4 Research questions and methodology . . . 98
4.5 Results, analysis, and discussion . . . 101
4.5.1 Distribution of requirements (RQ1) . . . 101
4.5.2 Type and quantification of quality requirements (RQ2) . 103
4.5.3 Safety and reliability requirements (RQ3) . . . 105
4.5.4 Robustness requirements (RQ3) . . . 109
4.5.5 Comparison with Sony Ericsson (RQ4) . . . 109
4.6 Validity threats and limitations . . . 110
4.7 Conclusion . . . 111

5 Paper D **113**
5.1 Introduction . . . 114
5.2 Background and related work . . . 116
5.2.1 Dependability . . . 117
5.2.2 Requirement patterns . . . 117
5.2.3 Refinement of quality requirements . . . 118
5.3 Robustness: a detailed analysis . . . 119
5.3.1 Robustness definition . . . 119
5.3.2 Root cause analysis model . . . 120
5.3.3 Symptom analysis model . . . 125
5.4 The ROAST framework . . . 127
5.4.1 Requirements patterns . . . 129
5.4.2 Requirements specification levels . . . 141
5.4.3 Using ROAST . . . 146
5.5 Evaluation . . . 147
5.5.1 Evaluation 1: comparing ROAST and non-ROAST requirements . . . 149
5.5.2 Evaluation 2: survey and workshop . . . 151
5.5.3 Evaluation 3: document analysis . . . 151
5.5.4 Evaluation 4: requirements analysis . . . 152
5.6 Discussion . . . 153
5.6.1 Threats to validity . . . 155
5.7 Conclusion . . . 155

6 Paper E **157**
6.1 Introduction . . . 158
6.2 Background . . . 159
6.2.1 CRASH . . . 159
6.2.2 Property-based testing . . . 160
6.3 Design of RobusTest . . . 161
6.3.1 Specified response to timeout and latency pattern . . . 162
6.3.2 Specified response to input with unexpected timing pattern 165
6.4 Integration with JUnit . . . 166

6.4.1 Automatic generation of test cases based on properties . 168
6.5 Empirical evaluation 169
6.5.1 Design 169
6.5.2 Results and analysis 171
6.6 Discussion and conclusion 172
6.7 Future Work 173

7 Paper F **175**
7.1 Introduction 176
7.2 Background 177
7.2.1 Timed Input/Output Automata 177
7.2.2 Property based testing 178
7.2.3 CRASH 179
7.3 Design of RobusTest 179
7.3.1 Timeout 183
7.3.2 Input with unexpected timing 184
7.4 Empirical evaluation 185
7.4.1 Company description 186
7.4.2 Case study design 186
7.4.3 Implementation of RobusTest 187
7.4.4 Results and analysis 188
7.5 Discussion 191
7.6 Conclusion 192

Bibliography **195**

Chapter 1

Introduction

1.1 Preamble

The increasing presence of software in our everyday life has increased the quality of our lives. The automation of previously manual tasks and introduction of new and more complex systems and features are evident everywhere around us, for example in the automotive and medical fields. Our increased dependence on software products has increased demand for more reliable and systematically developed software. *Software engineering* is an engineering field to develop, operate, and maintain software using systematic, disciplined, and quantifiable approaches [1].

However, only using systematic methods for software development is no guarantee for the quality of the resulting software. Software development processes need to incorporate specific methods to increase the different aspects of quality.

The first step is to define quality in software and to analyze the consequences of the absence of a certain quality aspect. This step should be performed during the analysis and requirements phase in order to lay the foundation for the rest of the development process. Requirements are usually divided into two main categories: functional requirements and quality requirements [2, 3]. Quality requirements are also known as non-functional requirements, quality constraints or -ilities [3]. While functional requirements specify the function that a system or component must be able to perform, quality requirements focus on the quality or the degree to which a system meets the specified requirements.

The development team also needs proper tools to design and implement software to achieve a system with the specified quality. Both during and after development, the team should be able to verify whether the desired quality has been achieved. This activity is directly based on the requirements. The goal of this phase is to ensure that the developed software complies with the functionality and quality stipulated in the requirements specifications.

The quality of a system consists of several quality attributes [2]. Dependability, safety, security, reliability, performance, robustness, and availability

are some of the well-known quality attributes.

An important aspect of quality for a high-quality system is the ability to function correctly and in an expected manner independent of events in the environment. This ability is also known as *robustness*. Problems related to robustness usually manifest themselves as degradations of functionality or degradation of other quality attributes [4]. This can be one of the reasons why robustness has gained less attention in industry and academia, compared with several other quality attributes such as performance, dependability, and reliability; it is a property of systems described in terms of other properties, and is thus less direct.

In the IEEE Standard Glossary of Software Engineering Terminology robustness is defined as [1]:

> The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

The purpose of this thesis is to investigate how we can make software systems more robust by introducing methods and tools that assist practitioners during different phases of software development. Studies suggest that including robustness requirements in the specifications can increase the quality and safety of software significantly [4, 5]. Improving robustness can also dramatically decrease the number of bugs and fatal failures in systems [4, 5]. Despite these findings, there is no research that gives clear guidelines on how to specify robustness in an effective and systematic way. In this thesis, we propose a framework, ROAST, that gives guidelines on how to elicit and specify robustness in software.

Another important part of this thesis is robustness assurance in the form of testing. The amount of existing academic research contributions in this area is higher than for robustness requirements [6, 7]. However, no study focuses on alignment of robustness requirements and verification activities. Most studies on robustness testing focus on random and fuzz testing of the system's interface, which only addresses a limited part of robustness issues. Therefore, we have created a framework, RobusTest, to assist practitioners in testing the requirements elicited by ROAST in a partially automated manner.

We have evaluated ROAST and RobusTest in an industrial context, where some of the companies are specialized in developing safety-critical systems. RobusTest was also evaluated on open source systems. To answer the research question we have used a number of different research methodologies and data collection approaches: interviews, content analysis, case studies, literature reviews, and evaluatory experiments.

The thesis consists of this introduction chapter and six other chapters, each based on a research paper. The introduction chapter is structured in the following way: Section 1.2 of the chapter goes through some related work for the main concepts and areas presented and used in the thesis. The main and derivative research questions are discussed in Section 1.3. The research practices and methodologies we used to answer these questions and evaluate the results are described in Section 1.4. Most of the research presented in

the thesis has been performed in an industrial setting, which is explained in Section 1.5. Section 1.6 gives an overview of the other chapters included in the thesis and explains which studies are included in the work and which research questions they answer. The methodology used in each chapter is also discussed here. Section 1.7 discusses the results with respect to the research questions and published academic literature, and places the work in relation to the software engineering body of knowledge. Potential future directions and research opportunities are discussed in Section 1.8. Finally, Section 1.9 concludes the chapter with a summary of research questions, methods and contributions the thesis offers.

1.2 Background and related work

Besides a basic understanding of software engineering, the reader of the thesis needs to have an overall understanding of some specific concepts such as requirements engineering and robustness. In this section, we introduce these concepts, which position the thesis in the current body of knowledge. The state of research of quality requirements is presented in Section 1.2.1, where we introduce some existing models and categorizations in the area. Software robustness definitions are presented and discussed in Section 1.2.2. A definition of the term is also given, which will consistently be used throughout the thesis. Section 1.2.3 gives a background for how similarities can be captured in the form of patterns in different phases of software development, especially in requirements engineering. Finally, Section 1.2.4 discusses refinement of requirements, especially quality requirements from abstract high-level requirements to detailed and verifiable ones.

1.2.1 Quality requirements & attributes

The increased use of software systems in every aspect of human life, with closer connections between man and machine, puts greater demands on software quality. Software quality attributes are the focus of many research studies, but the results are not in all cases unanimous. In this section, we give a short overview of the research on the software quality requirements and attributes that have inspired this work.

The IEEE Standard for Software Quality Metrics Methodology defines software quality as [8]:

> the degree to which software possesses a desired combination of attributes. This desired combination of attributes shall be clearly defined; otherwise, assessment of quality is left to intuition.

Robustness, safety, security, dependability, performance, availability are some of the attributes that customers and users expect to see in a high-quality system.

Many different classifications of quality attributes exist in literature. Boehm has introduced one of the first and most well-known quality models [9]. This model can be seen in Figure 1.1. Boehm divides quality attributes into three main categories: portability requirements, which include properties of independence of the system; as-is utilities, which mainly focus on the qualities of the system during run-time that are most important for customers and users and that have a direct impact on the perceived behavior of the system; and maintainability, which deals with the ease of maintaining a system. The maintainability attributes usually do not have any major impact on the quality that the users experience. Companies ensure these qualities to simplify maintenance and future expansion of the system, and they are usually included in the company's software development policy and internal documents rather than in the requirements specification documents. Robustness, which is the

main quality attribute of interest in this thesis, is in this model, a sub-category of reliability.

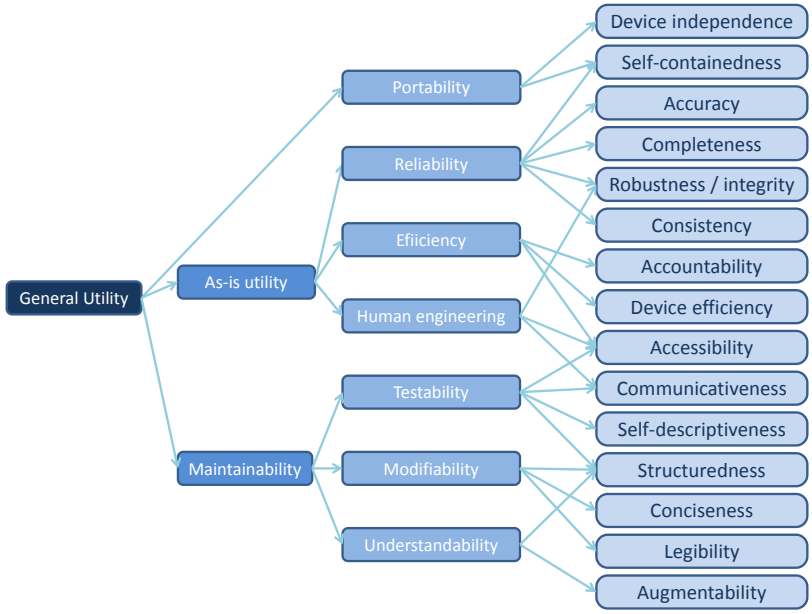

Figure 1.1: Boehm's software quality tree [9]

Some ISO standards on categorizations of quality requirements have subsequently been introduced. ISO 9126 and recently ISO 25010 (Figure 1.2) are the standards used for categorizing quality attributes.

However, ISO 25010 has another categorization that does not include robustness. We attribute this to the fact that robustness failures usually manifest as degradation of functionality or other quality attributes in the systems. Since ISO 25010 mostly categorizes symptoms and manifestations, it does not include robustness. In other words, robustness is a second degree or underlying quality attribute.

Quality attributes are instantiated as quality requirements when applied to the context of a software project. Quality requirements are also known as non-functional requirements [3]. There is an ongoing discussion in the field regarding the term *quality requirements* since non-functional requirements can be interpreted as requirements that do not function [8, 10]. Therefore, with a few exceptions in the appended papers, the thesis has used the term quality requirements to describe these types of requirements.

Researchers have defined quality requirements in a number of ways. Table 1.1 summarizes some prominent definitions of quality requirement [3].

Another classification provided by [18] identifies six types of quality requirements: interface requirements, performance requirements, operating requirements, life cycle requirements, economic requirements and political re-

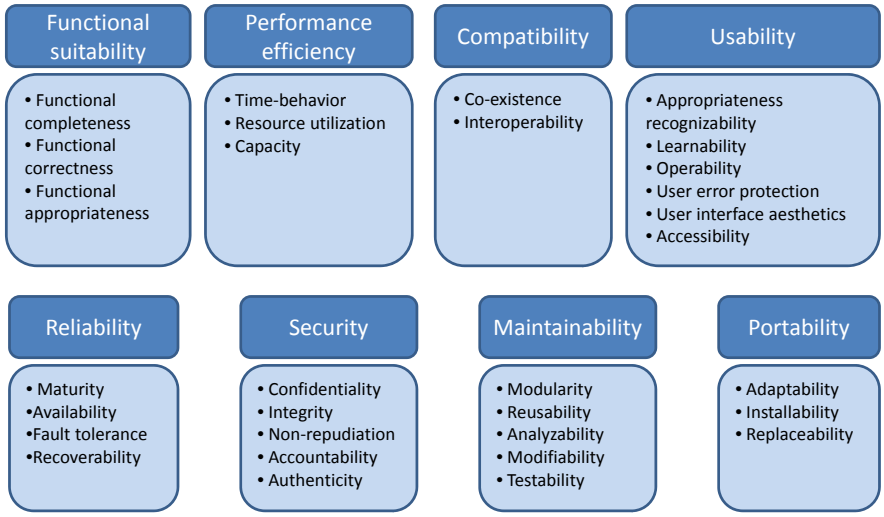

Figure 1.2: ISO 25010 standard on quality attributes

quirements. Robustness requirements are classified as operating and interface requirements in this model. The operating classification is important to notice, since many studies on robustness only consider the interface aspects of robustness requirements.

IEEE Standard for Software Quality Metrics [8] proposes a guideline on how to specify quantified quality requirements. One of the main activities in this guideline is to establish quality requirements in the following steps:

1. Identify a list of possible quality requirements
2. Set the list of quality requirements by prioritizing them
3. Quantify each quality factor

To identify a list of possible quality requirements (step 1), practitioners need to have a systematic way of classifying quality requirements. The list of quality requirements is determined in collaboration with the customer based on the list of possible quality requirements (step 2). Quantifying quality factors is another important step where practitioners specify the selected quality requirements in a verifiable format (step 3). We discuss this step further in this section and in Section 1.2.4.

In a literature review on software quality requirements, Chung summarizes some methods and approaches to specify, refine, and represent these types of requirements [2]. An important activity to quantify or refine quality requirements is to have a clear structure for the specification of these requirements. Robertson presents one structure on how to document quality requirements using [19]: identification number, NFR type, related use case, description,

Table 1.1: Definitions of the term quality requirement [3]

Source	Definition
Antón [11]	Describes the non-behavioral aspects of a system, capturing the properties and constraints under which a system must operate.
Davis [12]	The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.
IEEE 830-1998 [13]	Term is not defined. The standard defines the categories of functionality, external interfaces, performance, attributes (portability, security, etc.), and design constraints. Project requirements (such as schedule, cost, or development requirements) are explicitly excluded.
Jacobson, Booch and Rumbaugh [14]	A requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement.
Kotonya and Sommerville [15]	Requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet.
Ncube [16]	The behavioral properties that the specified functions must have, such as performance, usability.
Wiegers [17]	A description of a property or characteristic that a software system must exhibit or a constraint that it must respect, other than an observable system behavior.

rationale, originator, fit criterion, customer satisfaction, customer dissatisfaction, priority, conflicts, supporting material and history. Do Prado Leite et al. introduce another requirement specification structure based on scenarios with the following representation [20]: title, goal, context, resources, actors, episodes, and exceptions.

Van Lamsweerde et al. introduce a more detailed method for software quality specification; a goal-oriented approach is used to present the KAOS framework. KAOS models both functional and quality goals by using features such as type, attributes, and links with each other and with other elements of requirement models such as agents, scenarios, or operations [21].

Additionally, I* formal requirement framework [22, 23] uses the concept of soft goals for modeling quality requirements. This framework argues that quality requirements need to be transformed into functionality in order to have

an effect on the software development process. Therefore, a goal-oriented representation is appropriate to specify quality requirements since these requirements are usually expressed on high levels of abstraction that are later refined into more detailed and "operationalizable" requirements. In other words, quality requirements need to have a major effect on design decisions if they are to affect the system's behavior.

Another category of academic contribution presents frameworks for specifying quality requirements that utilize use cases, misuse cases, and UML-models [24–27].

Some of the results presented in this section have inspired parts of the ROAST framework. We can identify robustness requirements by applying patterns introduced in ROAST to the context of the software system under development. The identified robustness requirements are finally filtered by the developers and the customer based on characteristics of the project such as budget, time to market, and desired robustness of the system. Additionally, practitioners need to quantify and refine the high level requirements in order to obtain a detailed and verifiable set of requirements.

1.2.2 Software robustness

IEEE Standard Glossary of Software Engineering Terminology defines robustness as [1]:

> The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

To place robustness in its correct context, we need to relate it to similar quality attributes. Therefore, Table 1.2 lists some of these attributes and provides their definition according to the ISO 25010 standard on software quality requirements.

Robustness has been considered a quality attribute for achieving higher dependability in systems. Dependability is an "umbrella", "integrative" concept with multiple attributes [28]. Formally it and its basic sub-concepts are defined as [29]:

> the ability to deliver service that can justifiably be trusted in a software system.

Dependability is the aggregate of the following basic attributes: *availability* (readiness for correct service), *reliability* (continuity of correct service), *safety* (absence of catastrophic consequences for the user(s) and the environment), *confidentiality* (absence of unauthorized disclosure of information), *integrity* (absence of improper system state alterations), and *maintainability* (ability to undergo repairs and modifications) [29].

Robustness is defined informally as: "dependability with respect to erroneous input" [29]. However, it is clear that Avizienis et al. characterize robustness as a secondary and specializing attribute rather than a main attribute of dependability [29]:

Table 1.2: Definition of quality attributes according to ISO 25010

Attribute	**Definition**
Dependability	ability to perform as and when required and includes quality attributes availability, reliability, confidentiality, integrity and trustworthiness, maintainability, and safety.
Reliability	the probability that the system is continuously operational (i.e., does not fail) in time interval (0,t) given that it is operational at time 0.
Safety	the probability that the system will not incur any catastrophic failures in time interval (0,t).
Fault tolerance	degree to which a system, product or component operates as intended despite the presence of hardware or software faults
Security	degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.

> An example of specializing secondary attribute is *robustness*, i.e. dependability with respect to external faults, that characterizes a system reaction to a specific class of faults.

Thus, it seems that robustness can either be seen as a specialized attribute within the more general concept of dependability, or it can be seen as an extension of dependability to the situation of invalid input or stressful environmental conditions.

Robustness and security also have similarities in the sense that defects in the system design can be the source of robustness issues as well as security vulnerabilities that can be exploited by intruders. However, security issues are often due to faulty design, which enables the intruders to use the expected functionality and features of the system to perform their attacks.

Of note is the recent clarification by Laprie of the related concept of *resilience* as "the persistence of dependability when facing changes" [30]. Resilience is related to robustness in that the key is unforeseen changes: The changes apply to robustness in inputs and environmental conditions, and more generally to resilience in terms of any changes that affects the service delivery of the system.

One aspect that is common for several uses of the term *robustness* is that a system should show "acceptable" behavior in spite of exceptional or unforeseen operating conditions [31]. One task of robustness requirements is therefore to specify different levels of acceptable behavior for the system. This specification is related to graceful degradation of a system, i.e. that it can deliver parts of

Table 1.3: Different (mis)uses of the term robustness in industry

Definition	Description
Robustness as Quality (RaQ)	Robustness used in general to refer to quality and non-functional properties of a system.
Robustness as Dependability (RaD)	Robustness used to refer to all of the different dependability attributes, such as reliability, availability etc.
Robustness as Graceful Degradation (RaGD)	Robustness used to refer to the system's behavior degrading gracefully so that it still operates partially correctly or provides acceptable functionality in spite of errors.
Robustness as Input Stability (RaIS)	Robustness used to refer to the system being stable and able to function despite erroneous, exceptional or unexpected inputs.
Robustness as Execution Stability (RaES)	Robustness used to refer to the system being stable and able to function despite erroneous, exceptional or unexpected events in its execution environment (excluding inputs to the system but including the hardware and any software used to execute the system).

its originally intended behavior or function despite erroneous operating conditions.

To sum up, robustness is a broad term that is hard to capture in a single definition [32]. Our studies in industry show that robustness is interpreted and used in different and sometime incorrect ways by industrial practitioners. We found multiple incorrect and ambiguous definitions that overlap with other existing concepts. Table 1.3 summarizes some of these misuses.

The use of the term robustness to refer to quality in general (RaQ) should be discouraged. The concepts of robust and robustness are sometimes used to refer to some general quality characteristics. The reason that the term robustness has been used in this very general sense might be because it has system-wide relevance, i.e. that many quality attributes of a system can be affected if the system is not robust.

The robustness as dependability (RaD) interpretation of the term should also be avoided since it is inexact. Robustness is related to dependability in the sense that dependability attributes are affected, typically deteriorated, if the system is not robust. However, there are other dependability-related concepts that have this property, security being one.

Another 'misuse' of the term robustness is to equate it with graceful degradation. Graceful degradation is one common way to achieve a more predictable system. However, we discourage equating the terms since graceful degradation is only one, possible component or goal of creating a robust system. Some sys-

tems might always require full functionality and allow no degradation in system functions, but we still want to be able to discuss their robustness.

In this thesis, we use robustness as *input stability* in the presence of erroneous input and *execution stability* despite unpredicted operating conditions. Here, operating condition refers to the conditions in the environment that affect the system, such as shared resources with other systems and services.

1.2.3 Requirements patterns

Identifying and using similarities in the form of patterns is an increasing popular way to improve reusability and quality of work in different areas in the field of software engineering. Many studies have recently suggested design patterns of different types. In his book *Design Patterns*, Gamma presents an overall description of different design patterns [33]. Design patterns have roots in object-oriented frameworks [33]. Flexibility and extendability are two desired qualities in such frameworks. Reusability and structured processes are the other main aims when using design patterns.

Based on Gamma's model, other studies have investigated the possibility of applying patterns to other parts of the software development process. Gross and Yu suggest design patterns for quality requirements [34]. Fowler [35], Geyer-Schulz, and Hahsler [36] suggest patterns for high-level analysis of software, while Adams investigates patterns in fault-tolerant telecommunication systems [37].

Applying patterns in the requirements phase has also been studied in software engineering. Many books and articles have indicated similarities and patterns in requirements in the same domain and sometimes in different domains [38–40]. Konrad and Cheng propose a framework for specifying requirement patterns for embedded systems [41]. Using the ideas presented for design patterns, they identify high-level patterns for the requirements specifications for these systems and perform case studies to evaluate their solutions. Furthermore, requirement patterns have been studied on quality requirements such as security [42] and performance [43].

Another area where requirement patterns are widely used and popular is in software product lines [38]. This is due to the reusability achieved when using patterns. In the thesis, we have use the concept of patterns in the requirements phase to develop the ROAST framework.

1.2.4 Refinement of quality requirements

Requirements refinement is a standard practice in requirements engineering to transform qualitative high-level requirements at the start of a project into more specific and verifiable ones. The main goals of requirement refinement are to achieve completeness and to decrease ambiguity [44]. Yue [45] defines completeness as when you can formally or informally show that the requirements suffice for the goals to be achieved. Ambiguity is when a requirement can be interpreted in more than one way, and is a common problem for requirements specifications, especially written in natural language.

In goal-oriented requirements engineering, goals describe the objectives of the software system. Goals, standards, and policies are used to create scenarios that will be translated into requirements in the final specification [46, 47]. The goal-oriented approach increases the completeness of requirements.

Additionally, Gaus and Weinberg [44] propose a method to reduce ambiguity of requirements by refining them. Their method proposes to define functions, then specify attributes for the functions and constraints, and finally specify preferences for attributes. Although this work is mainly focused on functional requirements, parts of it can be valid for quality requirements, especially when they are on lower levels of abstraction and operationalizable.

Based on these results, Ho [43] proposes a framework for refinement of performance requirements called Performance Refinement and Evolution Model (PREM). PREM has four different levels, PREM-0 to PREM-3. PREM-0 identifies the goal of the requirement with a qualitative description that points out the performance focus in the system. An example is "the response time for adding a user shall not be too long" [43]. PREM-1 adds quantitative measures that are meaningful and obvious to an end user. The previous example would become "the response time for adding a user shall be within 1750 milliseconds". PREM-2 and PREM-3 add different levels of realism to the specification of factors that can affect the performance. For PREM-2 these factors are simulated while for PREM-3 they represent actual workload and environment data from the production environment. The example evolves into "The average response time for adding a user shall not be more than 1.87 seconds when, on average, the server receives 0.07 *Add User* requests per second" on level 2.

In the thesis, parts of the PREM framework concerning specification levels are adopted to the robustness context used in ROAST by redefining PREM levels for robustness.

1.2.5 Property-based testing

A property is a generic statement that specifies what a system should or should not do [48] in contrast to a test case, which is the execution of a set of specific operations in a certain order. Using property-based testing (PBT), high-level properties of the system that should hold are defined and used to generate test cases in order to verify and validate certain behaviors and constraints of the system. In PBT tools, a property is typically specified in a low-level specification language. A PBT specification language should provide temporal and logical operators and location specifiers to the tester [48]. This written specification is then used to automatically generate test cases for that property. The expected behavior of the system is also specified in the property specification that can be used by the oracle to automatically analyze the results from the test execution.

The property is used to validate the results created by the system under test (SUT). For the SUT to satisfy a property, the property should hold whenever a test case based on the property is executed [48]. In PBT the specified property should capture every aspect and case of the SUT if we are to conclude the correctness of the SUT with regards to that property.

A well-known tool for property testing is QuickCheck, which was initially developed for functional programming languages such as Haskell and Erlang but has now been developed for Java and other languages [49]. An example of a property written in QuickCheck to test the functionality when reversing a list of integers is as follows [50]:

$$
\begin{aligned}
&prop\_reverse() -> \\
&\quad ?FORALL(L, list(int()), \\
&\qquad reverse(reverse(L)) == L).
\end{aligned}
$$

A commercial extension of QuickCheck was used in [50] for verifying timing properties of an instant messaging server. Using the Erlang language Hughes et al. generate test cases to evaluate the timing of responses of an instant messaging server and compare the results of a property based approach with a state-machine approach. However, this study does not focus on robustness testing but rather argues how property-based testing can perform well for testing the timing aspects of an asynchronous system compared to the state machine approach. In other words, the tests conducted in this study evaluate the temporal relations for correct messages between clients and the server.

In the thesis, we have used property-based testing in the RobusTest framework that generates automated test cases and verifies the behavior of the system against certain properties that should hold.

1.3 Research questions

As discussed, software robustness assurance is an efficient way to increase the dependability and general quality of a system. The overall question to answer is how we can make software systems robust and how the robustness can be assured. To find an answer, we need to refine the question into the context of state of practice and state of the art.

The research questions in this thesis were refined in an exploratory manner. During our work in the industry and studying the academic literature, we realized there are many interpretations of robustness in both industry and academia. Therefore, the first question we want to address is:

> RQ1. How do we define robustness in different contexts and for different types of systems?

After clarifying the definition of robustness we want to answer the following question:

> RQ2. What is the state of knowledge on software robustness in academic publications?

By answering RQ2, we aim to give an overview of the field of robustness in academic literature and identify gaps in the state of knowledge. To answer this question, we also need to classify the existing literature and assess its quality. Therefore, we have defined the following sub-questions to RQ2:

1. What are the academic contributions at different phases of development regarding software robustness?
2. Are the results specific to certain types of systems?
3. What kind of quality and evaluation do the studies uphold?
4. Are there any gaps in the state of knowledge on robustness?

We also need to explore the state of practice on quality attributes and robustness in the software industry. The aim is to identify problems and challenges in the current way of working on these attributes. Studying other quality attributes is justified, and in some cases useful, because of their commonalities with robustness and because of the different naming conventions in industry.

> RQ3. How do practitioners work with requirements, and verification and validation, especially on robustness, at different phases of development?

We utilized an explorative approach to answer the main research goal, which is to develop systematic methods to work with software robustness. One gap in the state of knowledge that we identified was in the requirements phase. To address this gap we need to answer the following question:

RQ4. How can we systematically elicit and specify requirements to achieve a high level of robustness in software systems?

Studies suggest that completeness of requirements and having clear guidelines on how to write requirements and ensure their completeness is an efficient way to increase robustness of a system [4,5,51]. In our experience, many companies focus too little on specifying quality requirements, especially robustness requirements. The same behavior can be observed when it comes to testing robustness requirements. This lack of focus on robustness can result in unexpected failures, crashes, security compromises and quality degradation. In this thesis, we do not directly address robustness in design and implementation phases. Instead, we propose methods to specify, test, and assure robustness of a system. Therefore, the last research question is:

RQ5. How can robustness testing and assurance activities be aligned with the requirements?

Robustness considers the unacceptable input space in addition to the acceptable inputs of the system; therefore, the whole range of inputs and states of the system needs to be considered for complete robustness testing. The goal of these tests is to find as many bugs and failures as early as possible. Since robustness testing requires a large input space, and thereby many tests, to be executed, it is generally infeasible to test robustness manually.

Therefore, we propose that robustness testing should be performed with automated or semi-automated tool support, preferably with automated test case generation and result analysis.

Figure 1.3 show the connections between the research questions as discussed above. The figure also shows which studies and papers address each research question. The rest of the thesis describes the methods, research settings, results, and discussion of our research in pursuit of answers to these questions.

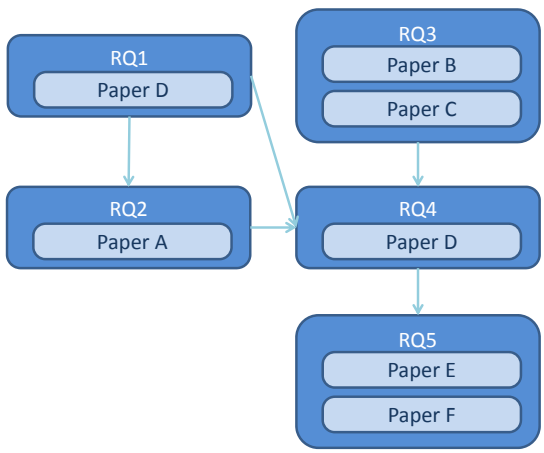

Figure 1.3: Research questions

1.4 Research methodology

In this thesis, we have used different research methodologies to answer the research questions stated in the previous section. One research methodology that we used is action research, which is when the researcher as an external entity uses methods such as observation, interviews, surveys and questionnaires to investigate a phenomenon and make changes in organizations based on the results. In addition to different types of action research methodologies, we used systematic literature reviews, design research, and evaluatory experiments. Table 1.4 gives an overview of the different methodologies that were used in the different studies incorporated in the thesis. In addition to data collection methods and research methodologies, Table 1.4 presents the type of research conducted in each paper. In the thesis we use the four research types defined by Robson [52]:

Exploratory: research to gain insights to generate new ideas and hypotheses.

Descriptive: research to describe phenomena or situations.

Explanatory: research to explain a phenomenon or a problem.

Improving: research to improve an aspect(s) of the studied phenomenon.

Table 1.4: Overview of methodologies used in the studies

Paper	Type of research	Data collection method
Paper A	descriptive	systematic review
Paper B	descriptive	semi-structured interviews
Paper C	explanatory, exploratory	semi-structured interviews, content analysis
Paper D	exploratory, improving	semi-structured interviews, content analysis, design research
Paper E	exploratory, improving	design research, experiment
Paper F	improving, evaluatory	design research, experiment

In the continuation of this section, we will present the above mentioned research methodologies and how we used them in the studies presented in the thesis.

1.4.1 Action research

Action research is a type of empirical research, where the focus is on practitioners, what they do, and the resulting artifact. Action research has been defined in a number of ways. McCutcheon [53] defines it as:

> systemic inquiry that is collective, collaborative, self-reflective, critical and undertaken by participants in the inquiry.

According to [54] there are four basic themes for action research: empowerment of participants, collaboration through participation, acquisition of knowledge, and social change. According to Zuber-Skerrit [55], action research is a spiral process that consists of four phases: planning, acting, observing and reflecting.

Further, according to Masters [54], action research can be of three types:

1. the scientific-technical view of problem solving;
2. practical-deliberative action research; and
3. critical-emancipatory action research.

This thesis mainly uses the first type of action research. In action research methodologies, the project is driven by a particular person or a group of people who are regarded as experts or authority figures. Action research is product centered but promotes practitioner participation in the improvement process.

Several research methods are commonly used in the action research methodology. In the studies included in the thesis, we used interviews, surveys, observation or/and document analysis. More specifically, we used semi-structured interviews, explained in Section 1.4.1.1, and content analysis, explained in Section 1.4.1.2.

1.4.1.1 Semi-structured interviews

A semi-structured interview is a type of interview that is primarily used to collect qualitative data [56]. The method is typically used when the interviewer explores a specific topic or area. The interviews are performed using an interview guide with prepared questions. After each prepared question, the interviewer can ask follow-up questions that encourage respondents to elaborate or clarify their answers. The method is typically used to obtain the respondent's point of view on a matter, rather than for quantitative data collection.

Semi-structured interview is a very common method to extract qualitative data and is discussed in several books and articles with that focus [56–58]. The main strength of the method compared to structured interviews is that the follow-up questions allow the researcher to acquire a deeper level of detail for the studied phenomenon. The weakness is the amount of time it takes compared to a structured interview and the fact that results can vary based on the skill of the interviewer.

In this thesis, the method was used to answer research questions RQ3 and RQ4 in papers B and D. The main interview questions concerned the state of practice on verification and validation in industry, specifically for robustness testing. In addition, the questions aimed to elicit what methods and tools are used for robustness testing and how the tools can be improved. On each topic, we asked more detailed questions depending on the answers received from the respondent.

1.4.1.2 Content analysis

Content or document analysis consists of activities for finding and classifying interesting information from documents [59]. The documents might be reports, records, method description, standards and so on.

Weber [59] defines *content analysis* as:

> A research method that uses a set of procedures to make valid inferences from text. These inferences are about the sender(s) of the message, the message itself, or the audience of the message.

Weber also states that:

> A central idea in content analysis is that many words of the text are classified into much fewer content categories. Each category may consist of one, several, or many words.

Some aims of document analysis are to identify the intentions and other characteristics of the communicator; reflect cultural patterns of groups, institutions, or societies, reveal the focus of individual, group, institutional, or societal attention, and describe trends in communication content [59].

Computer-aided content analysis is a very common and growing field. Using computers, documents are analyzed automatically or parts of the material that seem relevant are sorted out for further analysis by the researcher [59]. The first widely used computer system for content analysis is presented in [60].

Some of the different techniques that can be used for content analysis are [61]:

- Document selection and sampling
- Text encoding
- Key-word-in-context lists and concordances
- Word-frequency lists
- Retrievals from coded texts
- Category counts

Content analysis is a popular method in many research fields and disciplines. Content analysis in behavioral and psychological research is discussed in [62, 63]. Historical, political and, social documents and events have also been investigated using these methods [64].

In this thesis, content analysis is used in the form of document analysis by categorizing requirements from different companies into predefined categories. Functional, non-functional (QR), design, process, and hardware requirements are some of these categories. In the QR category, the requirements are further categorized into sub-categories of different quality attributes including robustness. This method was used to help answer RQ3 and RQ4 in papers C and D.

1.4.2 Systematic literature review

Systematic literature review is a method commonly used to evaluate and investigate all available research for a particular research question [65]. The method originates from the field of medical research, but was adapted to software engineering by Kitchenham, and is part of the evidence-based software engineering paradigm [66–68].

The base concept of evidence-based software engineering is objective evaluation and analysis of primary studies relevant to a research question. Torgerson describes the following points as the aims for a systematic review [69]:

- address a specific (well-focused, relevant) question
- search for, locate and collate the results of the research in a systematic way
- reduce bias at all stages of the review (publication, selection and other forms of bias)
- appraise the quality of the research in the light of the research question
- synthesize the results of the review in an explicit way
- make the knowledge base more accessible
- identify gaps, to place new proposals in the context of existing knowledge
- propose a future research agenda

Kitchenham's guidelines divide a systematic literature review into three main phases [66]: planning, conducting, and reporting. Each phase consists of several different stages. Some of the key stages are shown in Figure 1.4.

This method was used to help answer RQ2 and analyze the existing academic literature regarding software robustness. The main research questions in this study involved finding all existing literature about software robustness and categorizing them based on type of contribution, focus area, and focus system.

1.4.3 Design reseach

Purao describes design research as evolutionary and complementary [70]: evolutionary because it represents shifting and changing assumptions, and complementary because this type of research considers both phenomenon and artifact. Design research aims to simplify the phenomenon of research to an understandable level, and in early phases, the artifact only exists in the designer's mind. As the artifact begins to take shape, the complexity of the reality and the studied phenomenon force the designer to create simpler models of the reality to acquire a design.

In the early phases of research, researchers often cite evidence to support the existence of the problem, while later phases focus on validating the proposed design and artifact [71]. This gives rise to a view with multiple models

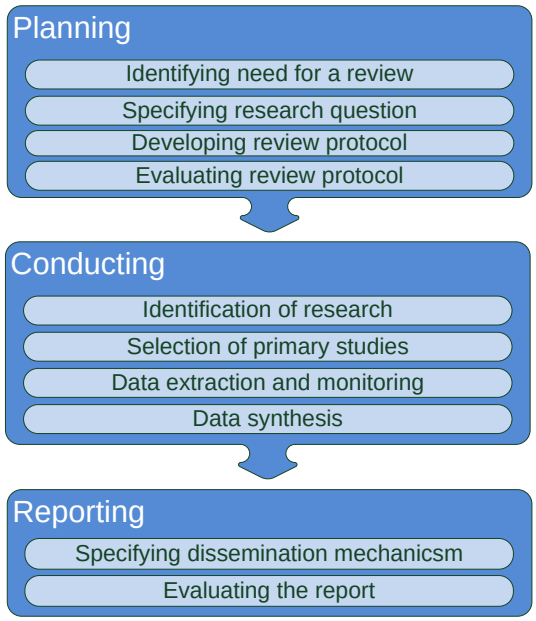

Figure 1.4: Different stages of a systematic review

of reality or the phenomenon, which is confirmed by Greg et al. for software engineering research [72]. The goal of the designer is to refine the perceived realities of the phenomenon and the idea of the artifact until they meet in the form of a design.

The design research process can be classified as a creative process of new thoughts and possibilities [73] characterized as *knowing through making*, in contrast to *knowing through observing or participating* [70, 74]. Purao lists three outputs from design research: the artifact, context-specific knowledge that has led to the artifact, and emergent theories that constitute a more general and underlying type of knowledge [70].

Whilst the "pursuit of truth" is the primary goal for paradigms such as positivist and interpretive paradigms, design research aims to improve our understanding and practice of the phenomenon of interest by creating new knowledge and artifacts [70].

Roozenburg and Cross propose a taxonomy of design models in design research based on earlier academic studies [75]. They propose two main model types: consensus (engineering) models that are prescriptive, and architectural models that are descriptive. Konda disputes Roozenburg and Cross's categorization, and proposes a new taxonomy based on processes rather than the artifact [76]. Konda emphasizes the context of research and argues against universal design models. Konda views design as a collaborative act between disciplines that requires the parties to create shared meanings and memory of

the design artifact. He classifies shared memory into two categories: vertical shared memory for the knowledge of a specific discipline, and horizontal shared memory for knowledge sharing between disciplines, space, time, organization, and culture. Shared memory aims to classify the process and organizational aspects of design innovation and research.

In this thesis we have used design research to create the ROAST and RobusTest frameworks, which respond to RQ4 and RQ5 in papers D and E. To create these frameworks, we gathered data from industry and the academic literature that guided the creation and design of the frameworks. We, later, evaluated the results and the design using different evaluatory methodologies to verify the designed models and frameworks.

1.4.4 Evaluatory methodologies

In a prominent paper on evaluation of research, Kitchenham proposes a framework for evaluating research results, called DESMET [77,78]. DESMET identifies three ways of organizing evaluation exercises:

Formal experiment: subjects are asked to perform a task using the studied methods/tools where results can be analyzed using standard statistical methods.

Case study: the studied methods/tools are tried out using the standard procedures of the evaluating organization.

Survey: investigates the practitioners past experience of methods/tools.

These methods can be applied to quantitative as well as qualitative research. DESMET also includes nine methods to evaluate design research: quantitative experiment, quantitative case study, quantitative survey, qualitative screening, qualitative experiment, qualitative case study, qualitative survey, qualitative effect analysis, and benchmarking.

An evaluation study following DESMET's guidelines executes the following six steps: identification of the context setting, planning and design, preparation, execution, data analysis, and decision making. Kitchenham further identifies three levels of evaluation: *basic level* for evaluations that target the understandability, usability and internal consistency of results, *use level* that explains whether using the proposed method/tool helped in practice, and *gain level* that shows that the results are superior to existing solutions in different respects.

Runeson and Höst add action research to the three above-mentioned methods and classify the purpose and nature of the research methodologies as presented in Table 1.5. In this paper, the authors compare case studies to action research with the distinction that a case study is observational while action research is focused on change processes (e.g. process improvement and technology transfer).

Experiments constitute one method to evaluate research. Basili et al. have introduced a framework for conducting experiments in software engineering

Table 1.5: Overview of research methodology characteristic [79]

Methodology	**Primary objective**	**Primary data**	**Design**
Survey	Descriptive	Quantitative	Fixed
Case study	Exploratory	Qualitative	Flexible
Experiment	Explanatory	Quantitative	Fixed
Action research	Improving	Qualitative	Flexible

with four main steps: definition, planning, operation, and interpretation [80]. The definition phase includes motivation of definition, purpose, perspective, domain, and scope of the study. In the planning phase, the researcher or practitioner designs the experiment and defines measurements and criteria to be measured during the execution phase. The operation phase starts with preparation in the form of pilot studies, which lays the foundation for execution in the form of data collection and validation. The last part of the operation phase is analysis where the data from the execution phase is analyzed to create models and plots. The interpretation phase starts with an analysis of the context through development of statistical frameworks and analysis of the study's purpose. Extrapolation is another part of interpretation that is used to examine sample representativeness. Finally, the last part of the interpretation phase is impact analysis, which measures the visibility, replication, and application of the results. Wohlin et al. have a similar framework but with an extra phase, presentation and packaging, which discusses how the results of the experiment should be presented in order for it to be complete and replicable [81].

Case studies are another method to evaluate research. Runeson and Höst have identified five sources of data for case studies [79]: interviews, observation, archival data, metrics, and checklists. Interviews and content analysis of archival data are used in this thesis.

Triangulation is an important concept in terms of improving the precision of empirical research in case studies [79]. Triangulation is necessary, especially when working with subjective and qualitative data. Four different types of triangulation may be applied [82]: data triangulation (collecting data at different occasions), observer triangulation (collecting data with different observers), methodological triangulation (collecting data using different methods), and theory triangulation (using different theories). We have used several of these triangulation types in the different evaluation studies in the thesis, which we will discuss when presenting each study.

In this thesis we have used the following evaluatory methods:

Paper A: no evaluation

Paper B: qualitative case study (basic level)

Paper C: quantitative case study and qualitative survey (use level)

Paper D: quantitative survey, qualitative experiment, and quantitative case study (use level)

Paper E: quantitative case study (use level)

Paper F: quantitative case study (gain level)

1.5 Research setting

Parts of the evaluation and solution proposal in this thesis have been conducted using open source systems and academic literature. However, a large part of research in the thesis has been conducted in an industrial setting. Four companies have participated in the research; due to confidentiality agreements we can not disclose the names of all the companies. Table 1.6 shows an overview of the companies that have been involved and provides some details about the domain and size of the companies. We present companies briefly in this section to help the reader understand the research setting in the thesis. More thorough description of the companies and the studied projects is available in the thesis where the studies are presented.

Table 1.6: Overview of partner companies

Company	**Size (ap.)**	**Domain**	**Methodology**	**Papers**
Volvo Technology AB	1,000	Telematics	Interviews, content analysis	B, D
Company B	100	Aerospace	Interviews, content analysis, experiment	C, D, F
Company C	100,000	Telecom.	Interviews	D
Company D	250	Telematics	Interviews, content analysis	B, D
Company E	50,000	Transportation & infrastructure	Interviews, content analysis	D

Volvo Technology (VTEC) in Gothenburg, Sweden, was the host company for the research performed during the first half of the project leading to this thesis. VTEC is a technology transfer company that primarily works with Volvo AB and Volvo Cars Corporation. The company is active in research, technology transfer, and development of software and hardware products. Our main collaboration with the company was with the telematics (the integrated use of telecommunications and informatics) division. The academic goal of collaboration was to improve robustness in telematics systems. The industrial goal of the project was to prototype a robust open platform that hosts third party applications. Companies C and D joined the project at a later phase. VTEC was involved in the research presented in paper B for analysis of the state of practice on working with verification and validation, and paper D where VTEC was the host company of the research that led to the design of ROAST.

Company B develops software-intensive systems in the field of aerospace. The company specializes in developing safety-critical software, and has extensive experience from working with established safety standards in the field.

The company has traditionally worked with a few well-established customers, but it is working on a more market-driven approach recently. Company B was a part of the research conducted in paper C for analysis of the state of practice in quality requirements, paper D for evaluating ROAST, and paper F for evaluating RobusTest in an industrial context.

Company C is large company active in the field of telecommunications. The company is an international company developing hardware, software, and services for many customers across the globe. Company C's role in the studies was limited to parts of the interviews and research presented in paper D to evaluate ROAST.

Company D is a mid-size company, specialized in developing and maintaining large telematics systems. The company is young compared with the other companies we have studied, and it works based on a bespoke and customer-oriented model with a few vehicle manufacturers. Company D's role in the studies was limited to parts of the research presented in papers B to investigate the state of practice on verification and validation activities, and paper D to evaluate the ROAST framework.

Company E is a large international company that develops safety-critical systems in the field of transportation & infrastructure. The company develops software-intensive systems in several different countries and location, and it works with many companies and governments on infrastructure and transportation projects. Company E was involved in the document analysis and interviews presented in paper D to evaluate the ROAST framework.

1.6 Overview of appended papers

In this section, we present a summary of the papers included in the thesis. For each paper, we introduce the research questions the paper addresses and the methods used to answer the questions. We also present the results and how they answer the question. An overview of the contributions from different papers is shown in Figure 1.5. The connections in the figure indicate how the results from one paper give motivation and strengthen the results of the next paper.

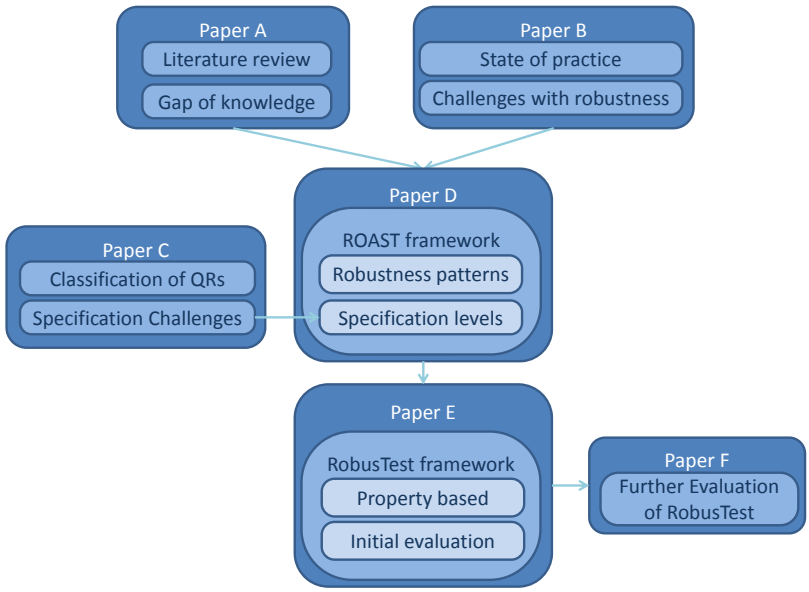

Figure 1.5: The included papers and their contributions and connections

Furthermore, we evaluate the validity of each paper and discuss the potential validity threats to the study and the results. For this purpose, we use a framework to assess the validity of a study [83]. This framework introduces four types of validity threats: conclusion, internal, construct, and external validity. *Conclusion validity* examines the strength of the relationship between the treatment and the outcome. *Internal validity* discusses the causality of the relationship between the treatment and the outcome. *Construct validity* assesses the generalization of the results of the theory underlying the experiment. *External validity* discusses generalization outside the scope of the study.

1.6.1 Paper A

Paper A, titled "A Systematic Review of Software Robustness", presents the results of a systematic literature review to extract the academic contributions in the body of knowledge regarding software robustness (RQ1). We applied the

search phrase "((robust OR robustness) AND software)" to four major research databases. The two authors filtered the 9193 identified manuscripts according to a pre-defined review protocol that resulted in a final set of 144 papers considered related to or as directly investigating software robustness. The 144 included papers were then classified to address RQ2 and its subquestions. As seen in Figure 1.6, the majority of the included papers focus on design, and verification and validation (especially testing) activities.

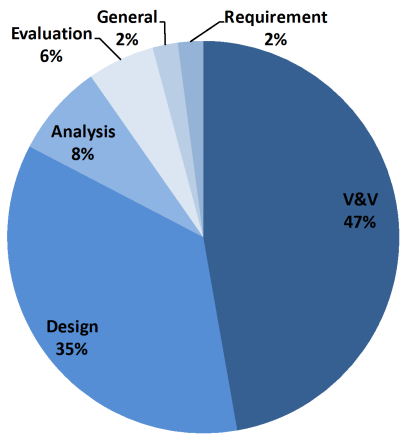

Figure 1.6: Phase focus of the included studies in the systematic review

We classified the quality of the evaluation of each paper and discussed the gaps in the body of knowledge on software robustness (RQ2). Figure 1.7 shows the type of evaluations conducted in each paper, and indicates that more than two-thirds of the studies lacked evaluation or were evaluated in a small academic lab environment. The lack of proper evaluation in these studies limits our ability to draw any conclusions on the applicability of the solutions in large projects.

In addition to the review presented above, another purpose of this paper was to identify gaps in our knowledge on software robustness. The most evident gap was identified in requirements engineering. Although studies have indicated the importance of a complete requirements specification to ensure system robustness [4,5], this area has, according to our investigations, not been explicitly researched in the past. This lack of research is the main motivation for the focus on robustness requirements in the thesis.

From a validity perspective, we have identified a few potential threats in the study. A conclusion validity threat relates to potential bias in the data extraction. We addressed the threat by letting *Author 2* classify a randomly selected 20% of the included studies. The results of the classifications by the two authors were then compared, which showed a statistically satisfactory low number of inconsistencies.

An internal validity threat in the paper was the possibility of omitted research results. We addressed the threat by letting *Author 2* classify 20% of

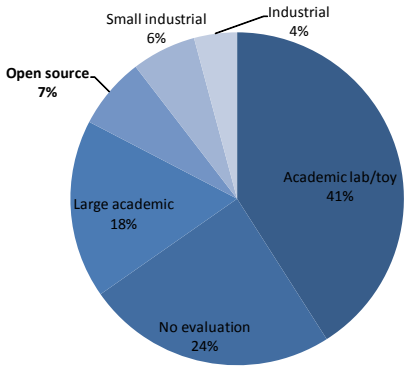

Figure 1.7: System focus of the included studies in the systematic review

all the papers in the search results. The number of inconsistencies was low and the authors discussed the inconsistencies to reach consensus.

An external validity threat in the paper was the search term. The search term was limited to *robust software* and *software robustness*. Since robustness is a well defined term in academic contexts, the risk of important studies about robustness being neglected due to the use of other terminology is minimal. Furthermore, all of the papers in the area previously known to the authors were part of the 9193 identified papers, i.e. covered by the search term. Another reason not to expand the search string was the large number of search results found by the string. According to our investigations, adding more words to the search string in a disjunctive search with *robustness* would have significantly expanded the search results, while having little or no impact on the number of included studies.

1.6.2 Paper B

Paper B, titled "Robustness Verification Challenges in Automotive Telematics Software", aimed to identify challenges related to software quality assurance, and verification and validation practices at the studied companies (RQ3). We performed the case study at two groups at Volvo Technology and Company D, which specializes in developing telematics systems, the term specifically used for telecommunication in vehicles to establish communication with the outside world.

The case study consisted of eight semi-structured interviews with testers, developers, requirements experts, and project managers. The interviews were explorative, and aimed to identify current challenges of developing software in the field of telematics. Another purpose was to find trends in the field and identify potential future problems and discuss how these problems can be addressed. Furthermore, we reviewed the academic literature to find more solutions to the challenges the companies face.

An important finding in the paper was that the studied companies per-

formed most of their testing using costly manual practices. Another finding was that the difficulty in capturing and assuring quality requirements entails a large cost to the projects. We also identified two trends in telematics: introduction of open platforms for third party components and dynamic deployment. These trends are important to observe since they can potentially lead to robustness problems in the systems due to unstable platforms and the third party components.

The paper presents several solutions to these problems and challenges: reviews, certifications, formal methods, and automated testing tools. We introduce the solutions based on the interviews and the academic literature. Despite an abundance of research results in the fields, companies still struggle to adopt the proposed solutions. The main reason is that most of the solutions, e.g. methods, frameworks, and tools, entail large overhead costs in terms of integration with existing processes and training. In our opinion, another reason for the lack of industrial adoption of the research results is that many of the studies are performed in narrow and focused contexts that result in context-dependent solutions that cannot be generalized.

Furthermore, industry usually has high inertia in adopting new tools and methods. Companies need to adapt the solutions to their processes and existing tools, and the resulting cost and risk discourage companies from adopting research results if they have not been properly evaluated. Moreover, these proposed solutions need to show significant improvement of the status quo to make adoption worthwhile.

The lack of easy-to-use automated or semi-automated testing tools for verification of a system's quality attribute adherence was an important finding in the study. This finding has motivated consequent parts of the thesis.

Despite a number of limitations and validity threats of the study, we decided to include the paper in the thesis to give a more clear picture on the questions raised by RQ3 regarding the state of practice on verification and validation activities at the studied companies. An external validity threat in the paper is the generalizability of the results to other contexts and companies. This is a common problem in case studies and we have addressed it by conducting interviews with several companies and by interviewing people with different backgrounds and roles. Furthermore, we have avoided drawing any general conclusions on our findings beyond the boundaries of the company and its working methods. However, the identified trends are more general to the field since they are driven by standards, business, and infrastructure plans.

Another threat is the small sample size of the interviewees. Although the interviewees were selected to represent different roles in the project and the company, it is difficult to eliminate the risk of having an incomplete picture of the project unless we interview all or the majority of the practitioners.

1.6.3 Paper C

Paper C, titled "Challenges with Quality Requirements in Industry: A Case Study", examines the state of the quality requirements elicitation and specification in a case study at a safety-critical software development company,

Company B. The purpose of the study was to ascertain the challenges related to working with quality requirements in industry (RQ3).

To answer this question, we conducted a content analysis study on the documentation from two projects (product specification in one project (P2), and customer (P1 customer) and product specification (P1 product) in the second project) including 980 requirements. We classified the requirements into functional (FR), quality (QR), external (Ext), process (PR), hardware (HW), and design requirements (DR). Since we were especially interested in QRs and FRs, we examined their distribution separately from the other types of requirements. As mentioned earlier, *Company B* develops safety-critical software and has a large focus on quality requirements. In a similar study, Berntsson Svensson et al. examined the state of quality requirements at a non-safety-critical project at Sony Ericsson [84]. However, there is a general lack of these types of quantitative classification studies on quality requirements. We did not find any similar studies for safety-critical systems. To understand the differences that safety criticality introduces in the distribution of QRs and FRs, we decided to compare our results with Berntsson Svensson et al.'s results. Figure 1.8 visualizes this distribution.

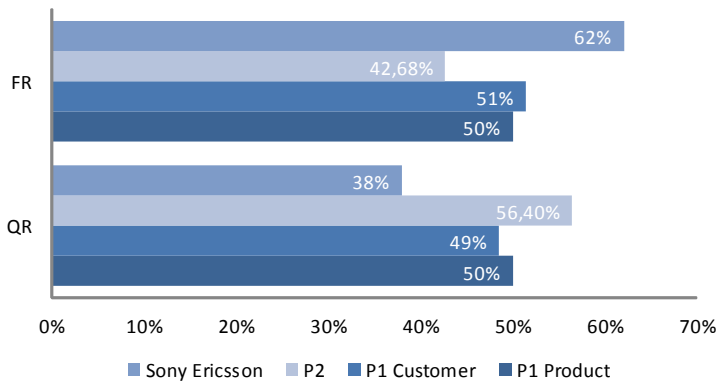

Figure 1.8: Distribution of quality and functional requirements in the studied projects

As the figure suggests, *Company B* focuses more on quality requirements than Sony Ericsson, as expected. The percentage of quality requirements in different projects at *Company B* is between 50 and 60 percent of all the examined requirements, while at Sony Ericsson this figure is lower than 40 percent.

We then studied the quality requirements in greater detail by considering two additional aspects: type and level of quantification. The types of quality requirements come from the ISO 25010 standard. In addition to the requirement types in the standard, we considered *safety*, which captures reliability requirements that deal with the safety of the system. In our judgment, safety is not included in the standard since it does not focus on the system, but rather

on the system's impact on the surrounding environment. In other words, safety requirements are often reliability requirements that focus on eliminating the risks the system can entail for users and the environment. In this paper, we only classify a requirement as a safety requirement if it is explicitly labeled as such by the customer or the company. Figure 1.9 presents the distribution of the requirements based on their type.

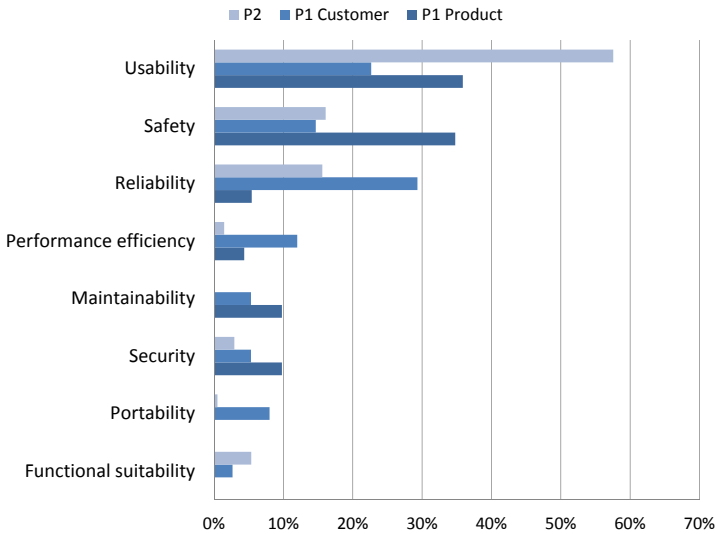

Figure 1.9: Classification of quality requirements based on ISO 25010

The figure suggests that when the customer requirements in P1 are refined into product requirements, the number of safety requirements increases significantly. These requirements are the result of the safety analysis process performed at the company. The other interesting finding is the large number of usability requirements, which can be due to the nature of the products and the large focus on user interaction.

The other classification aspect was the quantification of the requirements, where we divided the quality requirements into two categories: quantified quality requirements (QQR) and non-quantified quality requirements (NQR). QQRs are requirements that include specific quantifications on the events and behavior of the system that they specify. This classification could also be compared with Sony Ericsson's study. Figure 1.9 shows the quantification of the quality requirements in these documents.

The figure suggests that the number of NQRs at Company B is high, which is a potential source of risk due to verifiability issues on some NQRs. One category of NQRs that is intrinsically verifiable is operationalizable NQRs, which are requirements that address a quality attribute but which are also connected to functionality in the system, e.g. the login system. We call the remaining NQRs (excluding operationalizable NQRs) pure NQRs, which are

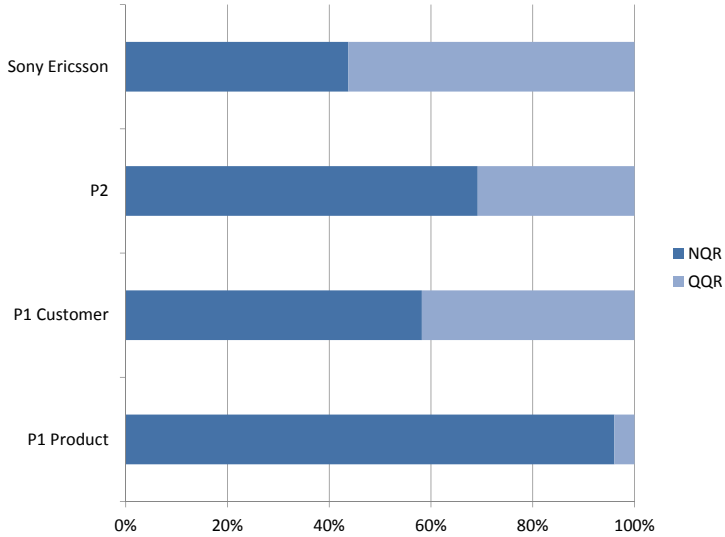

Figure 1.10: Classification of quality requirements based on quantification

NQRs that are not connected to any functionality and which have a low level of quantification. Pure NQRs state a non-verifiable goal on the quality of the system, e.g. "the system should be easy to use". Figure 1.11 shows this distribution.

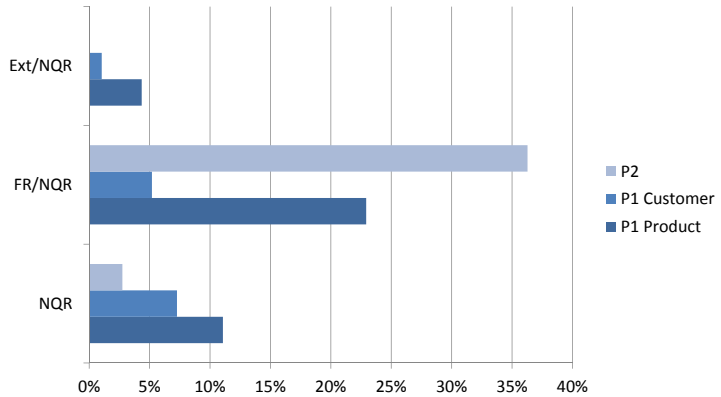

Figure 1.11: Classification of NQRs based on purity

The figure is normalized on all of the functional and quality requirements. From 289 requirements in *P1 product*, more than 10% were pure NQRs. To investigate the effect of the presence of their pure NQRs on the projects, we conducted five interviews with development and safety managers working in the projects. As expected, the NQRs received from the customer had been

a source of financial problems in the projects due to their ambiguity and unverifiability. These results strengthen our assumptions on the importance of requirements quantification, in particular when it comes to non-functional requirements such as safety and robustness.

1.6.4 Paper D

Paper D, titled"ROAST: A Framework for Specifying Software Robustness Requirements based on Patterns" addresses RQ1 and RQ4. Based on our interviews in Paper A, we realized that robustness is not a clearly defined term in industry since the standard academic definition needs to be interpreted to fit the industrial context. One research question Paper D aims to answer is the definition of robustness and what it explicitly means in practice (RQ1). Another question is how robustness requirements can be elicited and specified in a systematic way to address the gap we identified in Paper A.

Robustness has been defined in the IEEE standard for software engineering terminology [1]. However, practitioners need a clear understanding of the term in different contexts as well as the implications when there is a lack of robustness to be able to develop robust software systems. We originally discussed this issue in Paper D, although some of the results are also presented in Section 1.2.2, where we investigated different uses of the word robustness and defined robustness as [1]:

> The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

Paper D gives a deeper analysis of this definition based on the context and proposes *input stability* and *execution stability* as the two main parts of software robustness. In this definition, input stability refers to the stability of the system while receiving inputs, regardless of the validity and correctness of said input or its timing. Execution stability refers to the ability of the system, component, or service to function correctly in the presence of other systems, components, or services that may not function in an expected manner. As suggested in Paper B, execution stability is an important factor for open platforms and in Paper D we discuss what practitioners should consider to increase the execution stability of the platform, services, and software systems in general.

Based on this definition, we analyzed these types of stability further to clarify how robustness can be specified. When analyzing the existing requirements documents at our partner companies and the academic literature, we concluded that robustness characteristics follow certain patterns. To extract and describe these patterns, we needed to better understand the root causes and symptoms of robustness failures. We investigated the literature and requirements documents available to us in order to create the root cause and symptom analysis models. Input instability root causes can logically be divided into four groups as shown in Figure 1.12.

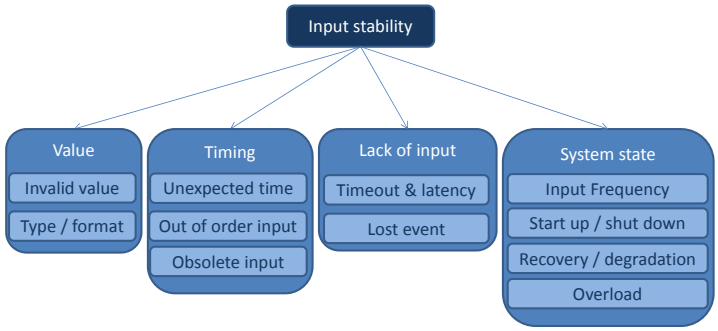

Figure 1.12: Root causes of input instability

According to our investigations, shared resources in the execution environment are the main cause of execution instability. Figure 1.13 shows some of the root causes we identified for execution instability issues. Of note is the category *other resources*, which includes resources not included in the first four categories such as sensors and actuators.

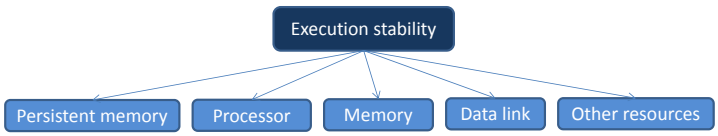

Figure 1.13: Root causes of execution instability

In the process of creating the root cause analysis model, we found some common means to achieve robustness (e.g. graceful degradation and wrapping). Furthermore, Paper D investigates the symptoms that follow a robustness failure. Figure 1.14 shows the common symptoms of robustness problems, categorized according to severity and type of response from the system.

These findings led us to create a framework for the robustness requirements elicitation and specification, called ROAST. ROAST consists of two parts: robustness requirement patterns and specification levels.

Based on the root cause analysis model, our literature review, and our investigations with the partner companies, we identified 19 patterns to specify robustness of a system:

1. Out of range or invalid value of input
2. Input with erroneous format or type
3. Input with unexpected timing
4. Input received in an unexpected order

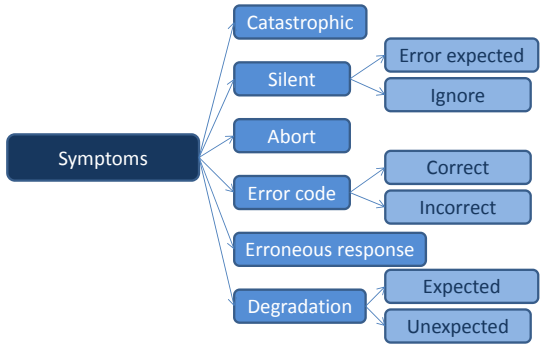

Figure 1.14: Symptoms of robustness issues

5. Obsolete input
6. Timeout
7. Lost event
8. High input frequency
9. Input during transitional state
10. Input during start up or shut down
11. Input during alternative operational mode
12. Input during system overload
13. Graceful degradation
14. Encapsulation
15. Shared run-time memory
16. Shared processor
17. Shared persistent memory
18. Shared network
19. Other shared resources

Paper D provides more details about the patterns and how they can be used as guidelines and checklists for practitioners to analyze, elicit, specify, and verify robustness requirements in a system. Application of these patterns together with the high level requirements and the system architecture, enables practitioners to perform a robustness analysis of the system that can result in concrete robustness requirements. However, similar to many other quality requirements, robustness requirements tend to be specified on a high level,

which decreases their verifiability and the focus and prioritization they receive during implementation and design.

Based on the results from Paper C and the academic literature, we suggest a model that measures the level of quantification and verifiability of each included requirement. The aim of the *specification level model* is to analyze and improve the quantification and details of robustness requirements, although it can be used for other types of requirements as well. The model includes five levels of requirements specification. Furthermore, the model is based on the requirements we have examined and, in its simplicity, captures the characteristics of requirements such as quantification, scope, and verifiability. While levels 1 and 2 are qualitative and unverifiable requirements, level 3 and 4 add more quantification and measurement, and level 5 adds more realism and data on how the system should be used and verified (Figure 1.15).

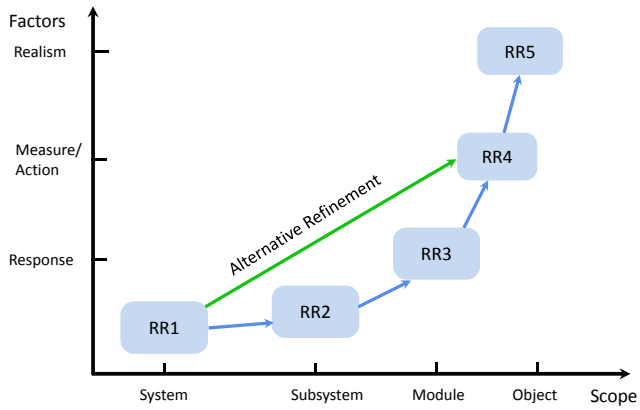

Figure 1.15: The levels of abstraction for requirements specifications based on factors and scope

Figure 1.16 visualizes ROAST in the context of its supporting models from top down. From left to right (bottom of the figure), the figure presents the process of how to apply ROAST, including the process inputs and the resulting output, which is a set of requirements for robustness.

We performed three empirical evaluations on ROAST, and the results indicate that ROAST mainly improves completeness of the set of requirements and reusability of requirements between projects; however, further evaluation of ROAST is required. A subject for future, therefor, is to find evidence for how ROAST affects the characteristics of other requirements.

Due to the small number of participants, the evaluations do not prove that ROAST provides complete, unambiguous, and reusable requirements from a construct validity perspective. However, the conducted evaluations demonstrate that applying ROAST can improve these qualities in the requirements documents.

Generally, using systematic methods to elicit and specify requirements often increases the quality and completeness of the requirements documents.

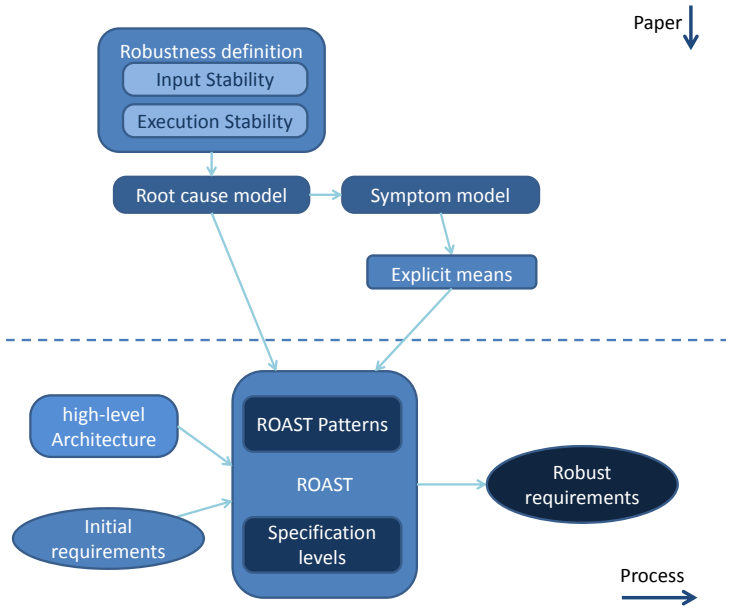

Figure 1.16: The structure of Paper D

According to our evaluations, ROAST is no exception to this rule. However, more evaluation can strengthen the foundation and validity of ROAST.

The main validity threat to this study concerned ROAST's external validity, i.e. that it would not be generalizable to other contexts than where it was created. However, since ROAST was evaluated at several companies, with comparable results, this threat is considered minor.

1.6.5 Paper E

The title of Paper E is "RobusTest: A Framework for Automated Testing of Robustness in Software". The paper explores how automatically or semi-automatically to test robustness requirements (RQ5). To address the research question, we created a testing framework called RobusTest.

Due to the intrinsic connection between requirements and verification, we created RobusTest-based robustness requirements in contrast to existing robustness testing frameworks that disregard the requirements and generate automatic tests solely based on the interfaces of the system under test to find bugs, e.g. Ballista and JCrasher [6, 7]. As discussed earlier, practitioners often neglect to specify robustness requirements, although they implicitly expect the system to be robust; this presents a challenge to our approach of requirements-based testing of robustness. To address the challenge, practitioners can use ROAST patterns that are also built into RobusTest to identify the possible types of robustness failures and specify the implicit robustness requirements

that should be tested.

Requirements that follow different ROAST patterns should be tested differently; however, similarities between requirements elicited from the same pattern enabled us to create a generic framework for semi-automated testing. RobusTest capitalizes on these similarities to specify behaviors or properties that the system needs to fulfill. In other words, these properties are extracted from the patterns in ROAST and from the implicit and explicit requirements of the system under test. We use generic parameters to represent different data types and classes in the generic test case specification. The generic parameters allow us to generate many test cases from one property.

When conducting automated tests, having an automated test oracle to analyze the results from test executions is a key factor in addition to automated test case generation. Therefore, practitioners and users should specify the expected behavior of the system alongside the test case description. Practitioners can use generic parameters to match the expected behavior with the generated test case. Furthermore, we have incorporated the robustness oracle called CRASH as the default oracle in our framework. CRASH stands for: Catastrophic, Restart, Abort, Silent, and Hindering failures that represent the different types of symptoms to which a failure leads [85]. However, CRASH only captures the most obvious and apparent failures in the system and does not cover the whole spectrum of possible responses as described and detailed in our symptom model presented in Paper D. Figure 1.17 shows the overall structure of the RobusTest framework.

In this paper, we evaluate RobusTest by applying a rather complex property, based on the *Input received in an unexpected order* pattern in ROAST, to test the robustness of two open source instant messaging applications, Vysper[1] and Ejabberd[2], which are based on the XMPP standard[3]. Our tests found three catastrophic failures, eight aborts and 15 silent (non-conformance) failures in the applications by running a total of 400 automatically generated test cases (the same 200 tests were run on each implementation).

1.6.6 Paper F

The title of Paper F is "Semi-automated Robustness Testing in Industrial Context Using RobusTest". In this paper, we evaluate RobusTest in an industrial context at Company B. The purpose of the paper was to further evaluate the RobusTest framework and assess its application in an industrial setting as well as assess its ability to cover ROAST patterns other than *Input received in an unexpected order*.

The paper uses the same design of RobusTest as the previous paper as shown in Figure 1.17. The evaluation of RobusTest in this paper was performed on the alarm module of a distributed safety-critical system that informs operators about the connectivity of the modules, and gathers data and

[1] http://mina.apache.org/downloads-vysper.html
[2] http://www.process-one.net/en/ejabberd/
[3] http://xmpp.org/

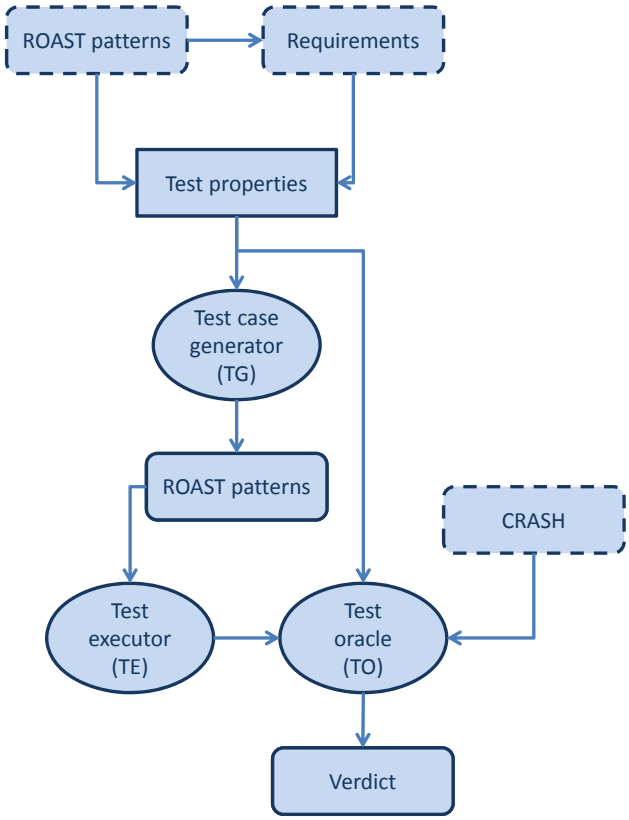

Figure 1.17: The structure of the RobusTest framework

parameters to generate alarms based on the state of the system and its modules. Figure 1.18 shows the overall design of the evaluation. On one side, the RobusTest framework generates test cases using four generic specifications that we provide. On the oracle side, RobusTest retrieves information about the generated alarms and checks them against the expected behavior of the system that is also specified in those properties.

In the alarm module, a configuration file specifies what data should be collected and from which modules the data should be collected. This configuration file also specifies the frequency of data retrieval. The collector part of the alarm module pulls the data based on the configuration file. The analyzer part of the alarm module analyzes the received data to generate alarms. We performed the fault injection and alarm retrieval using XML-based protocols that the alarm service supports.

Consequently, we used the four specified properties to generate 4,000 test cases to test the alarm module. The four properties were based on four ROAST properties: two properties on input with invalid value, one property on a

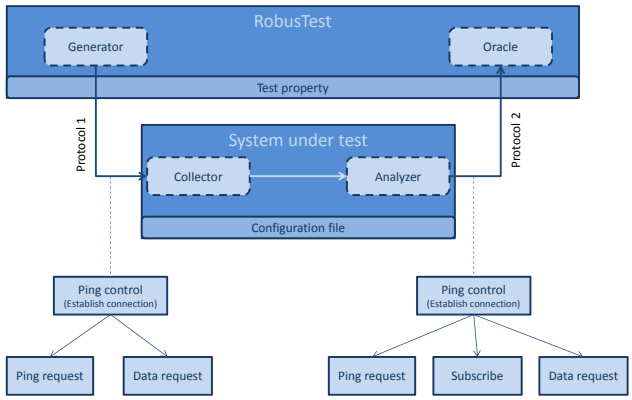

Figure 1.18: Design of the evaluation applied in Paper F.

combination of timeout and input during startup and shut down, and one property on input with unexpected timing. The test execution detected a total of four errors. Two of the errors were due to inconsistencies between the response and the specified protocol, and were due to out-of-date protocol specifications. Another error occurred while establishing a connection on the communication socket between the fault injection and data collector modules. When the response to connection initiation from the collector module was not received within the expected time period, the socket was shut down and the connection could not be reestablished without restarting both of the modules. The last error was detected on the last property, and was due to an internal design weakness in the alarm module. In this case, two contradicting alarms were set at the same time. Our investigations showed that the problem was fixed upon the next data collection round and the problem arises when the list of alarms is received by the oracle before the data collected by the collector has been completely processed by the analyzer.

The evaluation in this paper presents further evidence for the applicability of the RobusTest philosophy of semi-automated and directed testing for assessing robustness in software systems. Our study suggests that this type of testing is more effective than the traditional manual and unit testing, and more directed and effective than the more generic automated random robustness testing tools such as Ballista and JCrasher.

1.7 Discussion

In this section, we discuss our findings as regards achieving our research goal and answering the stated research questions. The subsections are divided based on our research questions, where we discuss the major findings for each question and how they fit into the existing body of knowledge.

1.7.1 How do we define robustness in different contexts and for different types of systems? *(RQ1)*

Quality and functional requirements have fundamental differences when it comes to requirements, implementation, and testing. While functional requirements are usually very focused on a specific part of the system and have a limited scope, quality requirements tend to have a more general scope, cover a larger part of the system, and stretch over several modules and functions. Robustness is no exception to this tendency.

Robustness failures often emerge from one part of the system, but the root cause usually comes from another part of the system. Depending on the affected module, it can spread to other parts of the system. Therefore, lack of robustness usually results in degradation of functionality or quality of the system, and the user usually experiences this as functional and quality degradation rather than lack of robustness. For this reason, we classify robustness as a second degree quality attribute. This issue might also be one reason that robustness was excluded in the quality attribute classifications presented earlier in the paper.

In our opinion, the exclusion of robustness as a main quality attribute in standard models has often led to misinterpretation of the term by practitioners and researchers. While our studies indicate an excessively general view of robustness as an overall quality or dependability in industry, researchers usually view robustness as the ability of the system to withstand input with a faulty value. This has resulted in robustness being neglected as an important quality attribute by industry and being viewed as an abstract and high level concept. There are, however, strong academic results on the ability of systems to withstand faulty input in the field of automated robustness testing. These results have emphasized a specific type of robustness testing that solely focuses on fault injection at the external interfaces of the system, which has had a strong influence on the use of the term robustness.

Therefore, to make a significant contribution in the field of software robustness, we first need to clarify the term robustness. For this purpose, we went back to the standard definition of robustness and extracted two main concepts in the definition: stability in presence of faulty inputs and stability in presence of stressful external environmental conditions.

1.7.2 What is the state of knowledge on software robustness? *(RQ2)*

To answer RQ2, regarding the state of the art in the field of software robustness, we conducted the systematic literature review study presented in Paper A. According to the results extracted from 144 relevant papers, many of the existing studies focus on random interface testing and fault injection to the interfaces of the system. The main purpose of this type of testing is to randomly find robustness errors at a low cost. The prominent tools in the field are Ballista [6] and JCrasher [7]. Given the external interfaces of the system under test, the tools generate random data that they send to the system's interfaces and diagnose the responsiveness of the system after each test. The tools have been used to test robustness of large open source, operating systems, and commercial-off-the-shelf systems(COTS). A limitation of the tools is that they fail to consider the internal state of the system before each test execution, which prevents the tools from testing more complex and relevant execution traces. This type of research has been the primary focus of the research community.

Design and implementation techniques to enhance robustness are the focus of another group of studies. Encapsulation (wrapping) of the interfaces, to filter invalid inputs, especially for COTS and graceful degradation of the system's functionality are the two most researched areas among these techniques. Defensive programming is another set of widely proposed techniques initially created to address security issues [86], that can potentially improve robustness.

The research results connected with the other phases of software development are very limited. Although researchers have shown the importance of completeness of requirements in increasing the quality and robustness of a system, our systematic review on software robustness, presented in Paper A, did not find any results and guidelines for the practitioners on how to achieve this. Our research has shown that neglecting proper specification and management of robustness requirements also occurs in the studied companies. In the interviews, financial considerations and lack of expertise on how to specify quality requirements were stated as being the factors limiting improvement in the robustness requirements elicitation and specification phases. However, it is important for practitioners to be aware of the risks when neglecting quality requirements during system design and development, in terms of assessing and prioritizing risk management in the requirements specification phase.

Another finding of the thesis is that the research community has solely focused on robustness in the presence of input with faulty value. Other aspects such as the timing of input, which also affects robustness, is a neglected research area in terms of software robustness [87]. Although there are studies that focus on timing of input, they usually lack a robustness perspective and focus mostly on timeout properties of the system, while disregarding lost inputs and the state of the system under test, for instance.

Research studies on robustness that study the behavior of the system under stressful environmental conditions is another area that is underdeveloped in academic research. However, there is extensive research on operating systems

and on creating a stable environment in which applications and services can run, and which aims to prevent the occurrence of stressful environmental conditions. Research is less extensive on how applications and services running on a platform should behave in case of critical and unintended environmental conditions.

From the system domain perspective, there are not many studies that directly address robustness in embedded systems. A concrete observation in this regard is that, although many of the results are general and applicable to different types of systems, there are very few studies that have specifically evaluated or contributed to the body of knowledge on robustness for embedded systems.

From an evaluation perspective, our systematic review shows that, two-thirds of the existing studies on software robustness lack strong empirical evaluation or evaluation of large systems. Ballista is one of the few exceptions in this regard, since it has been evaluated on large open source and commercial systems [88–90]. In conclusion, our interviews show that the lack of evaluation and the very specific and narrow scope in many studies discourage industry from adopting the academic solution proposals and contributions.

1.7.3 How do practitioners work with requirements, and verification and validation, especially on robustness, at different phases of development? *(RQ3)*

The results of the thesis presented for RQ3, regarding the state of practice on software robustness in industry, are mostly based on the requirements analyses and the information collected from interviews with our partner companies. Based on the interviews, industry is still highly dependent on manual testing for verifying both FRs and QRs.

Another issue that practitioners face is the difficulty of specifying quality requirements at an early phase of development. Our analysis indicates that a lack of QR specification often results in customer dissatisfaction, high costs, and extra work at the later phases of projects. Practitioners often neglect to specify quality requirements and lack methods to work specifically with quality requirements during the design and implementation phases. Furthermore, quality requirements are sometimes specified on a high level of abstraction, and are therefore neglected during development and testing due to their unverifiable nature and because of time pressure during the final phases of the project.

Some types of quality attributes such as performance and availability are easier to specify, but harder to ensure during development, e.g. *the system should always respond in less than 0.5 seconds* or *the system should be available 99% of the time*. These attributes are general properties of the whole system, and underlying problems relate to robustness for instance can result in degradation of the attributes.

We conducted a requirements analysis study (presented in Paper C) at Company B, which develops safety-critical software, in order to understand

the challenges practitioners face when dealing with quality requirements. As expected, the number of quality requirements in the company's systems was high (between 50 and 60 percent of the total number of software requirements depending on the document and project). We analyzed the customer requirements and the refined requirements in the form of product specifications. The interviews showed that subjecting the customer requirements to a rigorous safety analysis process led to refinement and elicitation of many reliability and safety requirements on the product specification level. Furthermore, we found around 10% non-quantified quality requirements that were unverifiable. Our interviews showed that these requirements are a major source of delay and late costs in the project since they are high level and qualitative, which forces the development team to interpret them subjectively.

Further, the analyzed requirements included very few robustness requirements from the customer. Therefore, the documents fail to specify the behavior of the system in unexpected situations and mostly focus on defining the main features of the system. In cases where unexpected situations are captured in later phases of development, the expected behavior assumed by the development team does not necessarily correspond to the expectations of the customer. It is important to note that the customers validated the products at the company in long acceptance tests lasting several months before final acceptance. Any changes at this stage create huge development costs and can trigger further development and the need to restart the validation process.

In comparison to the Sony Ericsson study [84], the products at Company B included more quality requirements. There was in particular more focus on safety, reliability, and usability requirements. The relatively larger number of quality requirements shows more focus on the quality aspects in the system. However, less quantification and less focus on verifiability of these requirements should be addressed to enable the customers to verify this higher level of quality that is expected.

Another observation on the differences in requirements management between the companies is the bespoke nature of the working process at Company B and the market-driven approach in the Sony Ericsson project. This difference in the source of requirements is another potential reason underlying the observed difference. Working in a market-driven context means that requirements are created internally at the company, which makes it easier to change and clarify them at later stages of development. In contrast, in bespoke projects following a linear or waterfall model, communication with the customer is more intensive in the requirements phase, and adding or changing requirements at later stages is cumbersome; therefore, it is essential to specify and verify the requirements at the early stages of requirements management.

Generally, the results show that quality requirements often are neglected in requirements engineering phase and in cases where they are specified, they are not specified in a quantified and verifiable manner. Combining this finding with the observation that the companies conduct manual testing, not suitable for verifying high-level and summative requirements, introduces threats of increasing development costs or lowering quality of the resulting product.

1.7.4 How can we elicit and specify robustness requirements in a systematic way? *(RQ4)*

To answer RQ4, on elicitation and specification of robustness requirements, we introduced ROAST, a framework for the robustness requirements elicitation and specification. We constructed ROAST by investigating root causes and situations that lead to robustness failures. Furthermore, we have studied the consequences and symptoms of these failures and how they manifest. By dividing robustness into input stability and execution stability, we found four major categories for the root causes of input stability. The main identified root causes are: erroneous value, unexpected timing, lack or loss of expected input, and the system's state when receiving the input. For execution stability, the main root cause was identified as resource sharing with other services and applications.

As discussed above, robustness requirements that are similar to other quality attributes often follow certain high level patterns in different systems and contexts. In order to elicit robustness requirements and ensure completeness of the requirements, ROAST includes 19 robustness patterns. Besides the patterns addressing the root causes of input and execution stability, ROAST also includes two design patterns. Although these patterns introduce restrictions and expectations to the design phase of development, it is sometimes essential to specify the expected parts and mechanisms in the design of the system during the requirements phase to ensure a certain level of quality for the system. A clear example in this regard is including a login system to achieve higher security and integrity of data. Practitioners can add more patterns to this list if they find more general or domain-specific patterns.

ROAST also presents a model to classify the level of specification in the elicited requirements. The model does not provide guidelines on the structure or syntax of specification; rather, it indicates the level of quantification and the measurements a requirement needs to maintain to be verifiable. We have identified five levels of specification in the robustness requirements we analyzed. The five levels are placed at coordinates based on two axes: scope and factors. We observed other levels of specification in for instance performance requirements ("The system should respond in less than 0.5 seconds", which is a requirement with global scope and specific measures). However, we did not observe any robustness requirements specified on these levels and therefore did not include the levels in the specification level model. For robustness requirements, we encourage practitioners to refine the high level requirements on levels 1 and 2 further, preferably to levels 4 and 5. This addresses the existing problems in the industry in terms of quantification of requirements.

We have conducted three evaluation studies on ROAST. The first study was a requirements analysis evaluation that investigated the completeness of the set of patterns. The other two evaluation studies used qualitative methods in the form of surveys and workshops to compare state of practice approaches to elicit requirements with the more systematic approach provided by ROAST. The results suggest that elicitation using ROAST improves requirement unambiguity, verifiability and the completeness of requirements. Interviews with

requirements practitioners confirm these benefits. Some interviewees have suggested using ROAST as a checklist during the requirements and verification phases. Other interviewees prefer to utilize the framework as a whole to elicit and verify requirements.

1.7.5 How can robustness testing and assurance activities be aligned with the requirements? *(RQ5)*

The results the thesis presents on this research question are based on applying the ideas in the RobusTest framework at open-source in industrial projects, as well as evaluating the results and comparing them with the results from manual or specified test case testing methodologies. Papers E and F discuss this question in more detail.

The RobusTest framework enables the user to write generic properties from which multiple test cases can be automatically generated, and the results can be automatically evaluated against the expected behavior of the system under test. RobusTest not only provides support to test input with invalid value, but also other patterns in ROAST that deal with input stability in presence of timing and system state issues.

RobusTest has been evaluated on two open source systems and a safety-critical industrial module. The results showed non-conformance with the protocol and errors due to input timing and values. We conducted these studies to evaluate the ability of RobusTest to find robustness errors and also to compare it with the manual and unit tests already performed on the systems we have tested.

We have currently implemented parts of RobusTest in Java. We used the implementation to test systems written in other programming languages such as C++ and C# as well. Due to the black-box nature of the tests and our communication with the system under test (SUT) through well-specified protocols, our implementation was independent of the SUT's development language. However, testing in the native language of the SUT might be necessary in other systems and this can limit the possibility of using our current implementation on those systems. The principles and patterns introduced in RobusTest and ROAST are general, though, and there are tools and libraries for automatic data generation for most programming languages. Therefore, testing with the RobusTest framework is not limited to Java or any other programming language.

An important property of RobusTest, in addition to semi-automated robustness testing, is that RobusTest provides clear links between the requirements and test cases. Despite this desired link, practitioners using RobusTest are not limited to the requirements documents. Practitioners can use the ROAST framework to perform robustness analyses on the requirements or the design to define new RobusTest properties. Performing this analysis is especially important due to our previously-discussed findings on lack of robustness requirements in the specification documents.

Our experience in the industry indicates that manual or limited unit tests for regression testing are still the dominant technique for testing software sys-

tems. Manual testing is a controlled and directed way of testing specific parts of the system using specific test cases and scenarios. Due to budget and time issues and lack of generic and easy to use automated testing platforms, companies often neglect to test the system's behavior in unusual cases that can lead to robustness problems. Quality attributes such as robustness, performance, and safety have a cumulative nature and testing a few controlled cases does not verify the presence of a specific quality in the system. Therefore, we argue for more generic and automated approaches for testing these qualities.

As regards robustness, the existing automated solutions known to us offer randomized testing with simple oracles that do not assess the correctness of the system's behavior and responses in the tested situations; instead they examine the responsiveness of the system after each test case execution [7, 88]. Although these methods and tools are able to find errors in the SUT at a relatively low cost [7,88], their undirected and unsystematic approaches prevent us from judging the robustness of the SUT. RobusTest offers an alternative to these approaches that is more automated than the manual approach and more directed and controllable than random robustness testing.

In this thesis, we did not compare RobusTest with other robustness tools such as Ballista and JCrasher. The reason was partly that the other tools are unable to capture the timing of input and partly because the systems we tested had more advanced types of inputs than what the other tools could handle. The other tools are, however, easier to use to find errors arising from oncorrect formatting and to find the value of the input in systems with simpler interfaces.

In summary, to answer RQ5, we introduced the RobusTest framework to create a link between the ROAST framework and the testing activities. Although we have captured our ideas in a testing framework, practitioners can use the practices and principles that constitute RobusTest separately to adapt it to their systems. Another important consideration is that given the direct link between ROAST and RobusTest, in addition to the requirements specifications, practitioners can use RobusTest to analyze the SUT and elicit robustness vulnerabilities and test them independently of the requirements documents. In addition to this link, RobusTest also provides a more efficient way of robustness testing than traditional and manual testing, and it provides more controlled and directed testing than older automated robustness testing tools.

1.8 Future research

We suggest three paths for future research. Two of the paths are aimed at industrial practitioners and one at the research community.

1. **Investigate design methods to improve robustness:** As already discussed, there are existing solutions for the design phase that address improvement of a system's robustness. A natural step after this thesis is to investigate the existing design solutions and possibly connect them to the patterns in ROAST. This step helps developers more efficiently fulfill the requirements set by ROAST.

2. **Industrial use of ROAST and RobusTest:** The thesis provides empirical data to support the possibility of using ROAST and RobusTest in industry. Practitioners can use ROAST and RobusTest in their current state as guidelines to improve the robustness of their systems. We encourage practitioners to customize ROAST to their scope and field and share their knowledge with the author. This will help improve and strengthen the validity of the frameworks.

3. **Expand to other quality attributes:** As discussed in the thesis, many quality attributes have characteristics that can be captured as patterns. However, the abstraction level varies for different quality attributes. Researchers have used pattern-based approaches for quality attributes such as performance. In our interviews, industrial practitioners expressed a desire to extend such frameworks to other qualities, such as security, reliability, and safety. For instance, safety patterns are generally on a higher level and based on types of hazards. These patterns can be further refined in the same way as ROAST. For other quality attributes, the patterns can be ascertained by the development team performing reviews or drafting document specifications instead of making tangible changes in the design of the system. Some of these solutions already exist in the form of safety standards and safety analysis frameworks or guidelines. However, we encourage researchers to focus on the different attributes and create requirements and analysis frameworks similar to ROAST to increase completeness and decrease ambiguity when specifying QRs.

1.9 Conclusion

The main goal of this thesis is to contribute to the body of knowledge on software robustness. To achieve this goal, we have stated the following research questions:

RQ1 How do we define robustness in different contexts and for different types of systems?

RQ2 What is the state of knowledge on software robustness in academic publications?

RQ3 How do practitioners work with requirements, and verification and validation, especially on robustness, at different phases of development?

RQ4 How can we systematically elicit and specify requirements to achieve a high level of robustness in software systems?

RQ5 How can robustness testing and assurance activities be aligned with the requirements?

To answer these questions we performed a series of empirical studies at four companies. We acquired the results in this thesis by conducting content analyses, semi-structured interviews, design research, evaluatory experiments, case studies, and systematic literature reviews. Our studies resulted in six research papers that are presented in the following chapters.

The first step of the thesis was to define robustness into a unified term that is applicable regardless of the industrial context. Based on this definition we further defined characteristics that a system should fulfill in order to be considered robust (*RQ1*). We refined the definition of robustness based on the definition in the IEEE Standard Glossary of Software Engineering Terminology [1] as stability in presence of unexpected or faulty input and execution stability in the presence of unpredicted, stressful, and unexpected environmental conditions.

Thereafter we conducted a systematic literature review on software robustness, where we found and classified 144 relevant papers that together make up the current state of knowledge on software robustness (*RQ2*). We found that existing studies mostly focus on random testing of systems on interface level. Other studies focused on the design phase, and how to improve robustness by encapsulating or wrapping the system against unexpected or faulty inputs. We also identified gaps in the knowledge, most clearly during the requirements phase.

Furthermore, the thesis investigates the state of practice on managing quality requirements and attributes, specifically robustness. We conducted interviews, requirements analyses, and workshops at five companies to identify the challenges that arise when working with quality requirements and robustness. Results from the studies suggest that the companies emphasize manual testing and often lack systematic methods for working with the quality attributes of the system (*RQ3*). We also performed document analyses of the requirements specifications at a company that develops safety critical system with

the objective of identifying and classifying requirements based on type (functional, quality, etc.) and their level of quantification. The results suggested by the study showed that the company had many QRs, but some of them were unquantified and unverifiable due to logical ambiguity, which had introduced additional costs to the project (*RQ3*).

To address the identified problems and gaps in the state of practice and knowledge, we introduced ROAST, a framework for software robustness requirements elicitation and specification. We have identified 19 patterns to help practitioners specify robustness requirements and five levels of specification to address the identified issues related to requirements quantification (*RQ4*). In four evaluation studies we evaluated ROAST using requirements analysis, practitioner interviews and surveys. The results suggested that ROAST improves robustness requirements with regards to unambiguity, verifiability, and completeness. The practitioners also found ROAST to be a valuable guideline for eliciting and refining robustness requirements. A static requirements analysis found that all robustness requirements in four requirements documents, concerning safety-critical systems, were covered by ROAST patterns. Finally, a dynamic analysis of requirements for a highly safety-critical subsystem showed that ROAST could find additional robustness requirements that were previously not specified for the system.

We also introduced RobusTest, a property-based robustness testing framework whose given properties (based on ROAST patterns and the context of the system) generate test cases for automated robustness testing (*RQ5*). In two different studies, we evaluated the ability of RobusTest to find robustness issues in two open source and one industrial system. We found eleven robustness issues in the two open source systems by running 400 semi-automatically generated test cases. Additionally, using the guidelines provided by RobusTest, we generated 4.000 test cases to test a safety-critical industrial module that had previously been through rigorous testing. The tests found two robustness issues in the system, which we reported to the company.

The main future direction of the research is to investigate software robustness in the design and implementation phases of the software development process and to incorporate the methods and frameworks presented in the thesis, with the industrial process.