Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Requirements Capture for Parallel Applications
D5.1

Due date of deliverable: 30th, June 2016

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP5

Responsible institution: Software Competence Center Hagenberg
Editor and editor's address: Bernhard Dorninger, Software Competence Center Hagenberg

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	30/06/16	Michael Rossbory	SCCH	Initial version
2	06/10/16	Lorena Paoletti	SCCH	Review and corrections (grammar, typos, formulation)
3	07/10/16	Michael Rossbory	SCCH	Chapter references

Executive Summary

This deliverable examines the process of requirements engineering (RE) in context with parallel programming in common and the Rephrase vision in particular. Upfront stands a glance at the basics of RE and elaborate on important terminology. The document continues with a brief overview of two of RE's vital aspects, elicitation and documentation.

In another section we briefly highlight alternative approaches to RE, before we bridge from RE to system/application design. The research of essential literature together with the results of a short survey is put into context with the vision of a design process pursued in the RePhrase project, leading to the conclusions in chapter 7.

Contents

1 Introduction

In this document, we will address the process of requirements capture for parallel applications. We will take a look on the goals and tasks of requirements engineering (RE) in general before considering the specifics of RE in context of the development of parallel applications. We will also examine typical design processes for parallel apps and their intertwining with RE. This task will take the input from T6.1. The result of this task will be a report (D5.1) describing issues concerning requirements engineering for data-intensive parallel software development. In particular, this document will contribute to answer the following questions:

1. How concrete does a Software Requirement Specification has to be in order to achieve a quick and reliable match to patterns?
2. How can the pattern description be organized and structured to support quick and reliable matching?
3. Which keywords/phrases are useful for pointing out specific problem solution strategies?
4. Which factors influence the selection of one (or more) patterns for a certain problem and how can these factors be measured?
5. Can we develop heuristics for selecting a specific pattern for a given concrete problem?

The remainder of this document is structured as follows:

The next chapter introduces to the basics of requirements engineering (RE) and features a taxonomy of notions. The subsequent chapters 3 and 4 elaborate on the aspects of requirements elicitation and documentation, chapter 5 provides an overview of Goal Oriented RE as an alternative or supplement to traditional RE.

In chapter 6 we briefly illustrate the impact of requirements on design in general and on design of parallel applications in particular, including pattern based approaches.

Chapter 7 explicitly puts it all into the RePhrase context as it tries to answer the questions asked above and provides some recommendations for requirements elicitation and documentation adapted to the envisioned RePhrase design process.

Chapter 8 is a short summary of the survey that have been carried out, before finishing with a short conclusion.

2 Introduction to Requirements Engineering

In system development and software development alike, requirements engineering (RE) is a crucial set of processes spreading different levels - organizational, product and project levels - and depending on the selected software engineering approach even different project phases. Its primary purpose is to determine and specify the functionality and the characteristics of the system to be developed.

Classical software process approaches (like, e.g. the waterfall model [57]) concentrate RE efforts in the early phases of a project, whilst newer, agile process definitions [29] suggest a more continuous and dynamic role of requirements engineering. But even classical approaches define RE as a continuous process with recurring activities during a software project.

2.1 The Goals of Requirements Engineering

The purpose of requirements engineering is not only to get a specification of the system to implement, it seeks to establish and maintain a common view of the system's stakeholder needs and demands. It also serves as a base for effort and cost estimation. According to Pohl [69], the three main goals of requirements engineering are:

- Improving an opaque system comprehension into a complete system specification: addresses the progression of requirements during the RE process from rather vague needs and desires in the beginning to a more or less complete specification.

- Transforming informal knowledge into formal representations: addresses the different means of representing or documenting requirements. Such representations may range from informal, natural language, semi-formal models up to rigorous, formal specification languages. From the goal of having a fully consistent and unambiguous specification, a formal representation would be preferable. Having a complete formal spec, it would theoretically be possible to fully generate the specified system. However, there are other

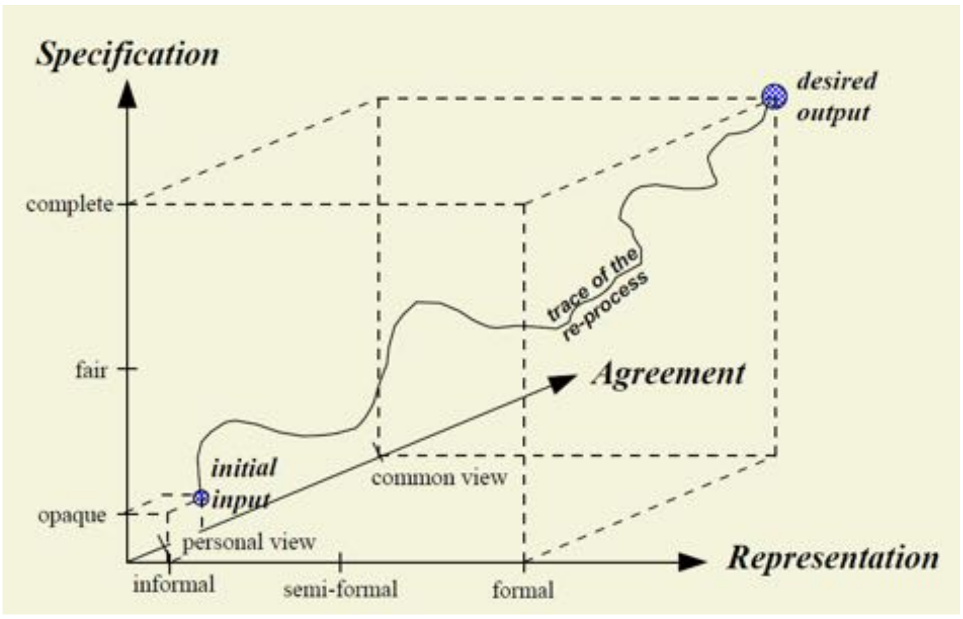


Figure 2.1: Three dimensions of RE (Pohl)

factors relevant when choosing a requirements representation, such as understandability or personal preferences, so having a formal spec is NOT always the desired outcome of the RE process.

- Gaining a common agreement on the specification out of the personal views: deals with the level of agreement between the stakeholders of a system concerning the specified requirements. In the beginning of the RE process, there might be very different, personal views concerning the planned system, which is only natural and usual has positive effects. As the RE progresses, these individual views shall be integrated into a common view.

Pohl argues that these three goals constitute the three dimensions of requirements engineering, which confine the transformation of original requirements input to the desired output of a RE process (See figure 2.1).

2.2 Activities in Requirements Engineering

Requirements capture is a somewhat fuzzy term, which is considered a not well-defined technique but rather a colloquial phase in requirements research and practice.

Capturing requirements means to complete a number of activities in an often continuous process (short: RE process). There is a number of more or less similar definition of the RE process [65] [73] [68] [18] [71], which state a varying number and organization of requirement engineering responsibilities or activities.

Aurum [18] lists five distinct activities with Elicitation, Specification and Modeling, Priorization, Analysis, Negotiation and Quality Assurance.

Pohl [68] defines a framework of 3 core activities (Elicitation, Documentation and Agreement) plus 2 cross-sectional activities (Validation and Management).

A more extensive suggestion for a RE process is delivered by Robertson and Robertson [71], which features more or less the same core activities, but also tries to emphasize on the continuous nature of RE.

However, as a consensus we may identify following important activities:

- Elicitation: is the process of gathering or developing requirements. Sources of requirements are identified during stakeholder analysis. Stakeholders can be human or even organizations. Additional sources of requirements may include documents or existing (software) systems, the latter not necessarily being a legacy system to be replaced by the system under development.

- Analysis and Documentation: After eliciting requirements, they need to be documented. Depending on the pursued development process paradigm and project or domain related factors, the extent of documentation may vary. Possible representation forms range from textual specifications to more or less formalized models.

  Specification and modeling frequently involves analysis to detect implications and dependencies between requirements.

- Consolidation and Validation: Especially in larger projects with a high number of stakeholders, disagreement and conflicts may arise concerning goals and requirements. Consolidation addresses to identify and resolve conflicts and thus has to support establishing one consistent set of requirements. Validation aims at assuring appropriate quality of the requirements artifacts themselves.

- Management: Requirements management (RM) is the set of measures that supports the other activities like elicitation, analysis, documentation and validation in an optimal way. Its four core tasks are to support information interchange concerning requirements. RM has to assure that gathered requirements are available and traceable. In addition, it shall support the definition and control of RE processes. Another key task is to maintain dependencies between requirements and other artifacts. Finally RM is responsible for providing reporting and controlling of the whole RE process.

In the context of this work we will concentrate mostly on Elicitation and Documentation.

2.3 Classification of Requirements

There are many possible ways of requirement classification. Aurum and Wohlin [18] sum up possible categories. A quite common distinction of requirements splits

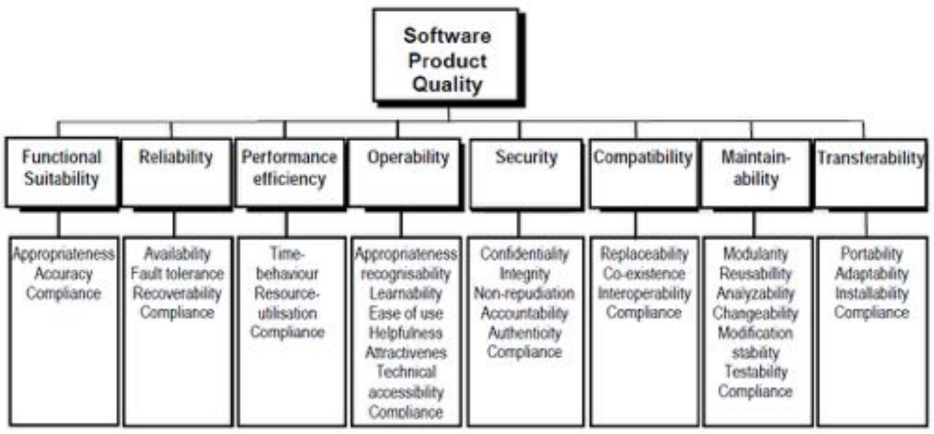

Software Product Quality
Functional Suitability
Reliability
Performance efficiency
Operability
Security
Compatibility
Maintain-ability
Transferability
Appropriateness Accuracy Compliance
Availability Fault tolerance Recoverability Compliance
Time-behaviour Resource-utilisation Compliance
Appropriateness recognisability Learnability Ease of use Helpfulness Attractivenes Technical accessibility Compliance
Confidentiality Integrity Non-repudiation Accountability Authenticity Compliance
Replaceability Co-existence Interoperability Compliance
Modularity Reusability Analyzability Changeability Modification stability Testability Compliance
Portability Adaptability Installability Compliance

Figure 2.2: ISO 250xx software quality standards

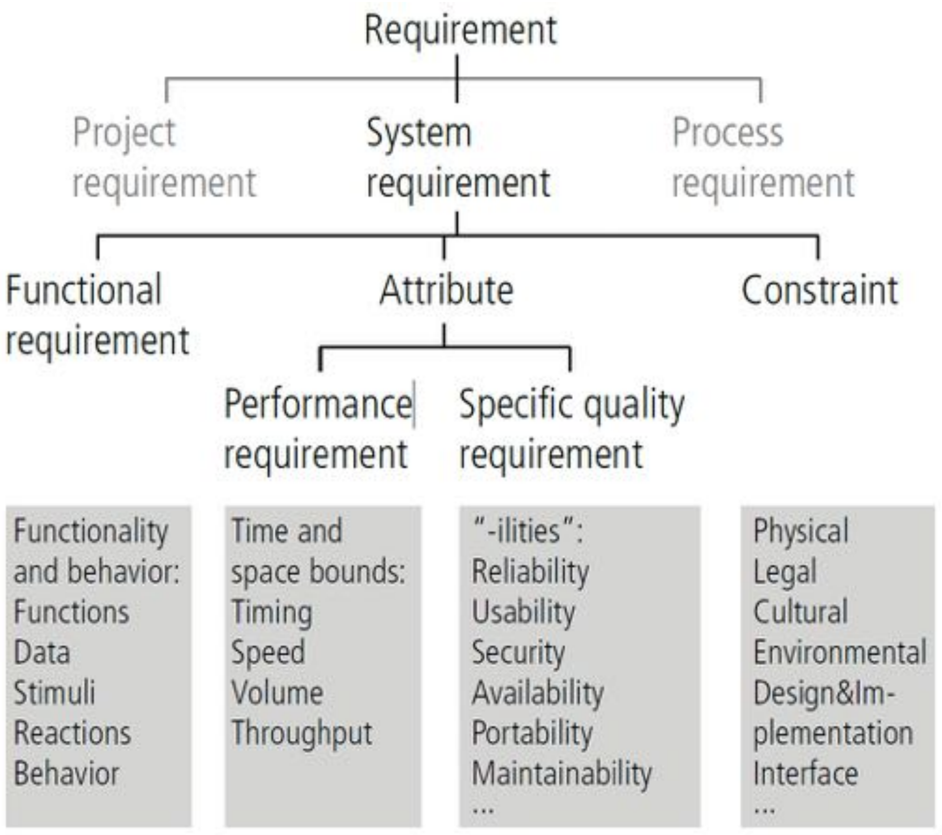
them into *functional (FR) and non-functional requirements (NFR)*. FRs describe *what* the system has to do and on the other hand NFRs specify *how* to do it by placing constraints on specified functionality.

While there is a broad consensus what a FR is, there are different views concerning the scope and classification of NFRs. Common to all these views is that NFRs (or at least a large portion thereof) address the quality of a system and its desired functionality. Well known and accepted classification schemes include FURPS [45] and the ISO 25000 [4] series of standards, the latter being formerly known as ISO 9126. They provide a number of quality attributes (QA) for being used for writing up NFRs. See figure 2.2 for an overview of ISO software QA.

[68] defines an NFR to be either such a quality attribute or nothing but an underspecified FR, which needs further clarification.

In turn, the superseded IEEE 830 standard [3] of Recommended Practice for Software Requirements Specifications (SRS) avoids the term *non-functional requirement* at all, but names different types of requirements each SRS may contain. The main focus of course lies on functions of the software (being the FRs). However, many of the QA categories defined in ISO25000 can be found in IEEE830 as well, with performance being the most prominent one. Other specific software system attributes include reliability, availability, security, maintainability and portability. The successor of IEEE830 valid to date is the IEEE/ISO/IEC 29148 standard [6], which lists NFRs as one of possible requirement types. It suggests two subtypes of NFRs, the one being "quality requirements" more or less mirroring the ones from ISO25000. The other subtype is called 'Human Factor Requirements' and refers mostly to usability issues. Both IEEE830 and IEEE29148 also introduce the term '(design) constraint', which more or less denotes requirements that restrict the degree of freedom concerning design and implementation of the specified system.

This view is backed by GLinz [43], who provides a good overview of defini-



Requirement
Project requirement
System requirement
Process requirement
Functional requirement
Attribute
Constraint
Performance requirement
Specific quality requirement
Functionality and behavior: Functions Data Stimuli Reactions Behavior
Time and space bounds: Timing Speed Volume Throughput
"-ilities": Reliability Usability Security Availability Portability Maintainability ...
Physical Legal Cultural Environmental Design&Implementation Interface ...

Figure 2.3: A concern-based taxonomy of requirements (Glinz)

tions of the term 'non-functional requirement' and provides his own concern-based taxonomy of requirements (see figure 2.3). According to this taxonomy, system requirements can be distinguished in functional requirements, (quality) attributes and constraints, with attributes being further separated in performance requirements and other specific qualities.

Glinz defines a NFR being a *performance requirement* or a *specific quality requirement*. The prominent position of performance on its own category is based on the wide importance performance requirements have in practice and the comparatively straightforward measuring of performance issues. Specific QA refer to the categories defined by the ISO 25010 model depicted in figure 2.2 - minus the performance category.

Note that this definition does not introduce constraints as NFRs. In requirements engineering, the term constraints is used in various meanings and contexts. Here, *a constraint is a requirement that limits the solution space beyond what is necessary for meeting the given functional, performance, and specific quality requirements*.

To allow classification of a specific requirement, Glinz provides a few simple

No	**Question** *What does this requirement state because we needed to specify...*	**Category**
1	... some of the system's behavior, data, input, or reaction to input stimuli - regardless of the way it is done?	Functional
2	... restrictions about timing, processing or reaction speed, data volume, or throughput?	Performance
3	... a specific quality that the system or a component shall have?	Specific Quality
4	... any other restriction about what the system shall do, how it shall do it, or any prescribed solution or solution element?	Constraint

Table 2.1: Requirements classification rules

rules backed by questions, which have to be asked in the numbered order (see table 2.1).

To get a more detailed view on classification of requirements, especially the different views on non-functional requirements consider reading [26]. For our purposes, we tend to follow Glinz' taxonomy, as it seems to suit the view on (NF)-requirements best when it comes to the design of parallel applications. This is due to the fact that design processes for parallel software treat performance as the dominant non-functional factor. Before we elaborate on that, we take a brief look at the interesting steps in the requirements engineering process.

3 Requirements Elicitation

As pointed out above, elicitation is the process of gathering requirements from various sources. Requirements of a system to be developed depend on the problem domain, the context and the specific tasks the application is to solve.

Before actual elicitation activities can start, there are some mandatory and important preliminary steps to perform upfront. These steps lay the foundations for the actual requirements elicitation and subsequent tasks.

3.1 Initial Analysis and Preparation

3.1.1 Understanding of the Domain

A prerequisite to requirements elicitation is an understanding of the subject specific domains being concerned. A requirements engineer needs to have at least a basic knowledge about the domain the system to be implemented will work in. Of course, in most cases the engineer cannot be a fully qualified domain expert, but he/she should develop a basic understanding of the most important artifacts and processes. This also includes the expert language and technical terms being used. A glossary can help to reduce the communicative gap and avoid misunderstandings.

3.1.2 Goal Analysis

Goals are the starting point of every development project. In requirements engineering elicitation, analysis and documentation of goals is an essential task, which provides the base for all following steps. In addition, goals have a high communicative value - at least top goals are frequently used as project headnotes.

How to develop goals of course depends on a high number of factors, e.g. the character of the project: Do we develop a completely new innovative, product? Do we have to supersede a legacy system by a modern successor? Depending on this factor and other factors, the elicitation methods for defining and refining goals have to be chosen.

’Traditional’ RE sees goal analysis more as a preliminary albeit important step. There is a RE paradigm that pursues a goal centric approach on RE called ’Goal Oriented Requirements Engineering (GORE)’. GORE approaches also emphasize

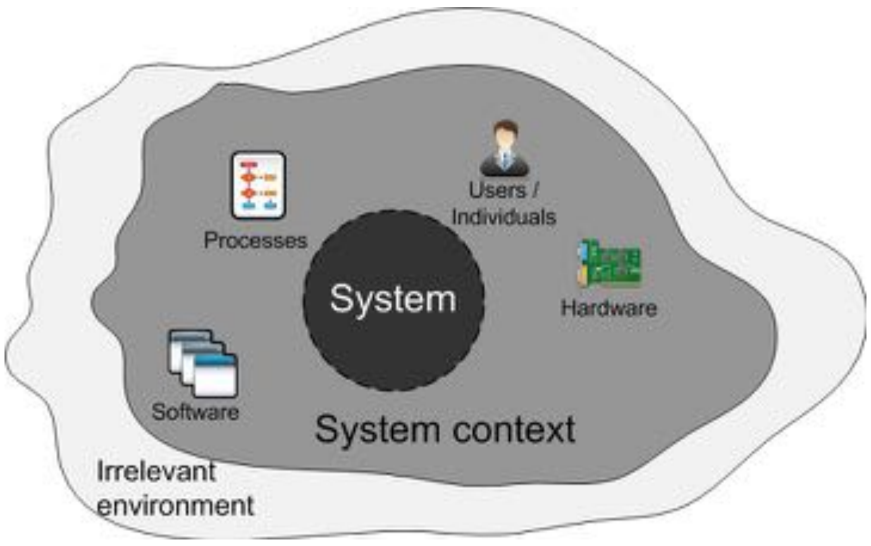


Figure 3.1: System Context

the influence of goals on architectural design. In chapter 5, we will take a look of some of these approaches.

3.1.3 Analysis of the System Context

There is barely any system developed from scratch and in an airless environment. If there is a need to solve some kind of existing problem or support an existing (manual) process, the initial situation is the point to start. This is not always in the scope of work of the requirements engineer but rather for a business analyst for complex business processes or a technical expert, if the domain problem centers on complex technical processes. Whoever gathers and analyzes which knowledge whatsoever is dependent on project and domain characteristics. Doing an analysis of the initial situation or studying the outcome of such an analysis also contributes to understanding the domain and may reveal additional goals or provide rationale for existing ones.

In another early step, the scope and the context of the system to be developed have to be determined and analyzed. The goal is to identify all material and immaterial objects, which have a relationship to the system to be developed. Objects of interest include living individuals (e.g. users and other concerned persons), neighboring systems (HardWare (HW) and/or SoftWare (SW)), processes the system is participating in or other important artifacts. Obviously the results from an initial analysis are of great use here (Figure 3.1).

- System border: Which aspects of the real world shall be covered by the system to be developed? Which aspects are parts of the system's environment?

- Context border: Which aspects of the environment need to be analyzed? Which parts of the environment remain irrelevant?

The system context is the part of a system environment relevant for the definition and understanding of the requirements. Note that a prospective point of view shall be taken, since the context shall focus the - not yet existing - system to be developed. Apart from identifying all the relevant objects, context analysis shall also investigate the system's relationships with these objects. Of course, at this point of the requirements process, there shall be no detail analysis (e.g. analyzing the inner details of a transport protocol). The goal is to provide a very brief description and classification of the relationship (e.g. data flow between system and neighbor A is image data).

Documentation of the context can be either in textual form or with the help of graphical representations, either proprietary block diagrams or established, such as UML Deployment Diagrams [10] or Data Flow diagrams.

During the development of a system, its context may change. This of course has to be addressed by the requirements process. In fact, it is unlikely - especially for large projects - that an initial context remains unchanged.

3.1.4 Stakeholder Analysis

Another important step is to identify the stakeholders of the system, i.e. persons or groups who are or will be affected by the system and thus are potential sources of requirements. But there can also be immaterial or organizational stakeholders, which may be significant to the system, such as standardization bodies.

The result of the context analysis may serve as a basis, since important users and other affected persons or groups should have been already identified there. Here, the task is to gather detailed information about each stakeholder including contact, expertise and availability information as well as documenting useful characteristics such as the stakeholder's motivation and interests. Stakeholders are usually documented in tabular form - most common is the use of spreadsheet applications.

3.1.5 Other Sources of Requirements

Intertwined with analyzing stakeholders is the identification of sources of requirements. Obviously, the various stakeholders are most important, but there are other sources, which may provide valuable information, such as legacy systems, existing documents or even solutions of competitors.

Requirements sources are also mostly documented in spreadsheets.

3.2 Selection of Elicitation Methods

Depending on the stakeholders, there is a more or less clear notion what the software shall do. There is a number of established methods to elicit, document and validate functional requirements.

The selection of the right methods depends on various factors reflecting the context of and the situation in the project. Rupp [73] offers a table mapping diverse factors to some of the more important elicitation techniques (see figure 3.2). It is important to note, that tables like these are no dogma rather than experience based characterizations from real-world projects. Roughly, elicitation techniques can be divided into creativity techniques, observation techniques, questioning techniques and artifact based techniques.

There are far more methods than depicted in figure 3.2. Prototyping, for instance, is another prominently used method. Although mainly used for requirements verification and validation, it is also useful in detecting new requirements or refining existing ones. This is especially the case for interactive systems, where UI prototypes may be sketched in workshops or used in scenario walkthroughs.

More or less detailed descriptions of elicitation processes and methods can be found in [73], [68], [71] and [89].

3.3 Conducting Requirements Elicitation

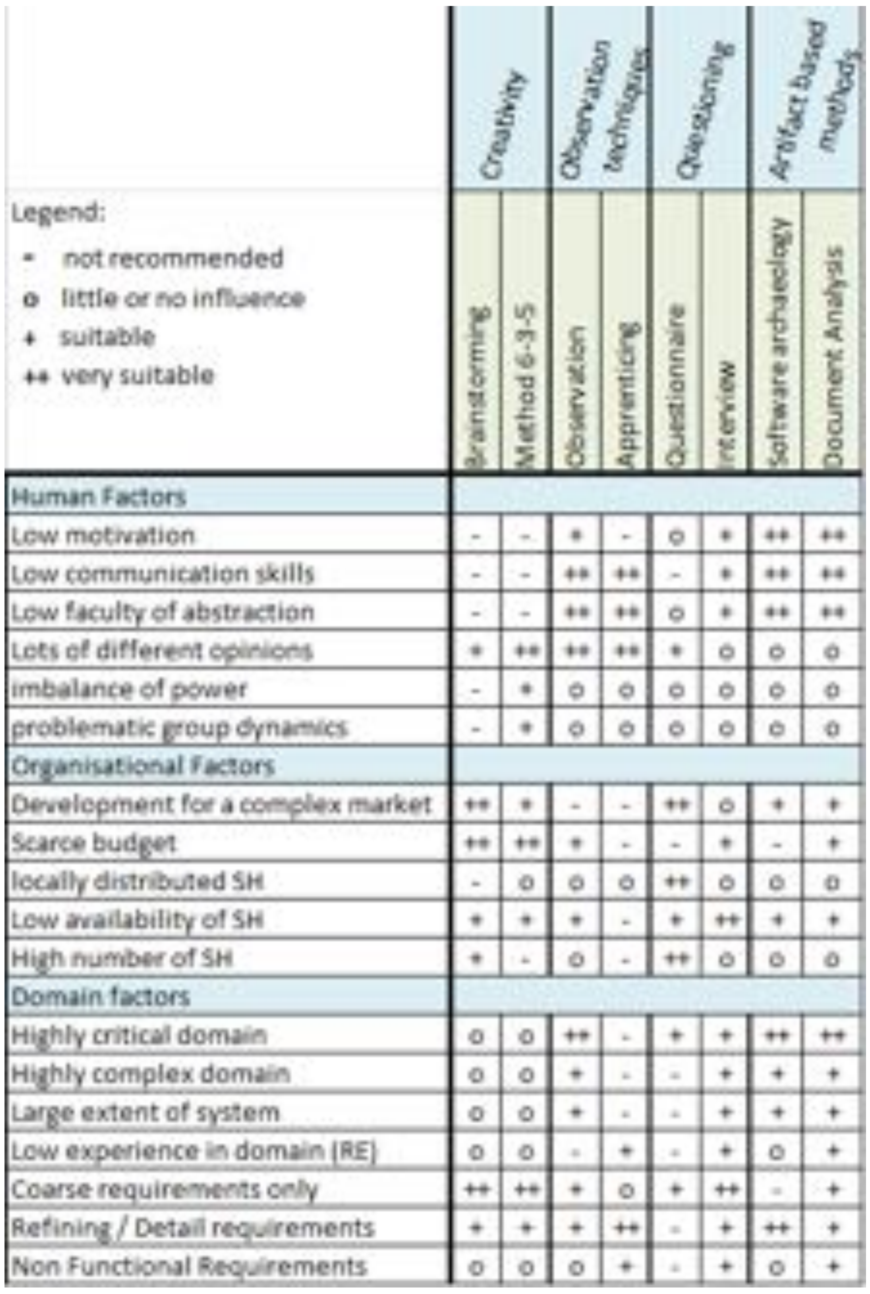
Once methods have been selected the elicitation activities may commence. Usually, requirements are developed top down starting with coarse grain requirements and subsequent refining. To what extent a requirement has to be refined highly depends on the objected functionality, its importance and even the underlying design process used.

For instance, agile design processes like Scrum tend to have a more relaxed approach to requirements analysis than the German V-Modell-XT. In practice, designers and developers are quite regularly confronted with ambiguous and incomplete requirements.

There are some important criteria when it comes to eliciting and documenting requirements. The ISO standard for RE [6], in Section 5.2.5, defines such quality criteria for individual requirements or a set of requirements.

One of these criteria is completeness: The standard states a requirement being complete, when *"the requirement needs no further amplification because it is measurable and sufficiently describes the capability and characteristics to meet the stakeholder's need"*. Interpreted in a simple yet practicable way requirements are complete, if the relevant stakeholders are satisfied and see all their required functionality being reflected in the specification. This implies the continuous consolidation of elicited requirements with the relevant stakeholders. The advice is also valid when eliciting requirements in a re-engineering project, which involves a fair lot of artifact based methods, e.g. analyzing the functionality of a legacy system. Here, the requirements engineer is also to seek consolidation of the retrieved information from system experts.

According to Zowghi and Gervasi [90], there are two dimensions to completeness. Internal completeness refers to the completeness of individual require-



Legend:
- `-` not recommended
- `o` little or no influence
- `+` suitable
- `++` very suitable

	Creativity		Observation techniques		Questioning		Artifact based methods	
	Brainstorming	Method 6-3-5	Observation	Apprenticing	Questionnaire	Interview	Software archaeology	Document Analysis
Human Factors								
Low motivation	-	-	+	-	o	+	++	++
Low communication skills	-	-	++	++	-	+	++	++
Low faculty of abstraction	-	-	++	++	o	+	++	++
Lots of different opinions	+	++	++	++	+	o	o	o
imbalance of power	-	+	o	o	o	o	o	o
problematic group dynamics	-	+	o	o	o	o	o	o
Organisational Factors								
Development for a complex market	++	+	-	-	++	o	+	+
Scarce budget	++	++	+	-	-	+	-	+
locally distributed SH	-	o	o	o	++	o	o	o
Low availability of SH	+	+	+	-	+	++	+	+
High number of SH	+	-	o	-	++	o	o	o
Domain factors								
Highly critical domain	o	o	++	-	+	+	++	++
Highly complex domain	o	o	+	-	-	+	+	+
Large extent of system	o	o	+	-	-	+	+	+
Low experience in domain (RE)	o	o	-	+	-	+	o	+
Coarse requirements only	++	++	+	o	+	++	-	+
Refining / Detail requirements	+	+	+	++	-	+	++	+
Non Functional Requirements	o	o	o	+	-	+	o	+

Figure 3.2: Some important elicitation methods and their selection criteria (Rupp)

ments. External completeness is addressing the phenomena of missing requirements: Achieving external completeness of a requirements specification is in most cases (for larger projects) a big challenge. There is always the possibility to miss out on certain requirements, e.g. if stakeholders take a requirement for granted but are not explicitly stating it - and on the other hand the requirements engineer might not know about this demand at all.

Apart from completeness, there is also the question of the detail level of a requirement or a set thereof. It is not possible to give exact figures or rules on how detailed a single requirement or a full specification has to be. The coarser a requirement or a set thereof is, the greater is the degree of freedom - which may be explicitly desired though - or possible misinterpretation. ISO 29148 specifies the quality attribute of *'Unambiguousness'*, which mandates a requirement to be only interpreted in one possible way. But ambiguity may not just arise from the lack of detail. It is quite common that detailed wordings are becoming very difficult to understand. To avoid this, requirements need to be carefully formulated (see chapter 4). Again, consolidation techniques offer great support here.

Espana [36] have done a scientific reflection on the completeness and granularity of conceptual models by demonstrating their findings on experiments with two RE methods (use cases and communications analysis).

A more practical approach to tackling requirements detail level can be found in [73], who introduces five specification detail layers: While *Level 0* has coarse-grained requirements typically interesting for stakeholders from the management level (e.g. visions, strategic goals, very coarse features,...), the highest *Level 4* targets system architects and developers. This level might contain detailed technical requirements, component definitions and their interfaces, test cases for components. However, even Rupp's distinction of layers remains rather vague and is not free of overlaps. There are no hard characteristics or constraints putting a requirement on a specific detail level.

3.4 Elicitation of NFRs

Ostensibly, it is the desired functionality that determines and drives the design of a solution. However, as suggested by Bass et.al. [19] and others, NFRs and constraints usually have a huge impact on the architecture and design of a solution. Contradicting that, in many projects or even design processes NFRs are neglected.

However, there is consensus throughout the practical and academic RE community, that NFRs play a special role in software development. Typically, NFRs are initially expressed on a very high level only and have to be refined during the RE process. Although there are some 'global' NFRs, in most cases a NFR is tied to a specific functionality, capability or a task a system has to solve. But it is not only functional requirements, a NFR may be related to. Other NFRs and constraints may be intertwined with an objected NFR. Also it has to be considered that different stakeholders might have different views on certain quality attributes (e.g.

reliability may be interpreted differently).

Thus, elicitation of NFRs usually involves iterations of elicitation and analysis. Elicitation and subsequent refining may be supported by a catalog or checklist customized for the objected project and/or domain. This may be a full or partial catalog based on the aforementioned ISO25010 categories.

An important aspect concerning NFRs is that they should be measurable (e.g. [55]). Quite frequently, NFRs are expressed by stakeholders like *"The computations shall be performed as fast as possible..."*, *"The system shall be secure......"* or similar phrases. Such verbalization offers plenty of room for unsuitable interpretation. Not only because NFRs frequently are used for acceptance criteria the need for having measurable and verifiable requirements becomes comprehensible.

In this light, Pohl [68] argues that quite frequently, (imprecisely formulated) NFRs are nothing but underspecified functional requirements. Therefore refining should also clarify if a requirement is really non-functional or can be further developed into a set of functional requirements. It is the task of the requirements engineer to resolve vague specifications of NFRs regardless of decomposing it to FRs or making them verifiable NFRs.

Having measurable criteria is still not enough, since the outcome of a benchmarking or measurement usually depends on the context and environment of that measurement. This is especially the case for external quality attributes like Performance. Quite frequently, requirement constraints (such as "hardware platform to be supported") may be sources of context and environment information. Thus, refining should seek to fully clarify the desired value of an important NFR as well as to achieve consensus under which circumstances and in which context this value shall be reached by the system.

In practice, elicitation is often done completely unspecific to the type of requirements, i.e. FRs and NFRs are elicited by applying methods from the same basic method set (see also section 3.2). It has been criticized that the state of the practice often does not consider the link between FRs and NRFs on the one hand and on the other hand does not give advice on ending the elicitation process. This is addressed by the IVENA method [73], which incorporates suggestions made by Dörr in his "Task and Object Oriented RE" approach (TORE) [13] and propagates reuse of RE artifacts.

Here, the elicitation is guided by a catalog of interesting NFR categories (e.g. based on the ISO25000 standard), a set of usual constraint categories and tailored checklists for both sets. The influence of Dörr's work is manifested in closely relating NFR elicitation to the available functional requirements. By considering the developed checklists, stakeholders are queried for NFRs along already specified system functions (e.g. use cases) or already defined system components. With this functional oriented elicitation approach, it is more likely to get a more complete set of NFRs than with an approach decoupled from system functionality.

3.5 Requirements Elicitation in Agile Approaches

Nowadays, as agile approaches to software engineering have become more popular than conservative development processes, exuberant specifications tend to be less common, especially in smaller scale projects. Other than the traditional approach, RE in agile processes emphasizes the evolution of requirements as the project progresses. However, although there are significant differences on how requirements are analyzed, documented and managed, the aspect of elicitation - and the methods used for it - is quite similar according to Leffingwell [53]. A major difference can be found in the emphasis of the continuous nature of requirements elicitation and refinement, which is owed to the iterative nature of most agile project approaches.

However, compared to projects following strict and heavyweight process models such as the V-Model XT [21], elicitation (and subsequent documentation) of requirements in agile projects tends to aim at a more coarse level of requirements. This of course leads to a higher degree of freedom for developers and on the other hand may be the source of misunderstanding and misinterpretation. There have been many debates and criticism regarding the handling of requirements in agile projects, especially the neglect of dealing with NFRs (e.g. see [34], [24]). Hence, this has been addressed by the agile community, e.g. by introducing backlog constraints [53], chapter 17.

3.6 Requirements Elicitation for Parallel Applications

There is no sign of publications suggesting special elicitation methods for parallel applications, either functional or nonfunctional ones. Especially at the beginning of the requirements engineering process, there is little knowledge about the problem and its context. In fact it is likely, that stakeholders do not care whether a solution to their problem is implemented using a parallel programming approach. They will rather be expressing more or less specific performance goals. It is also quite frequent that stakeholders are even not aware of the possibility of having a problem solved by a parallel implementation, which is especially the case for non-embarrassingly-parallel problems.

These views are backed by the results of a short survey (see chapter 8) examining the role of requirements engineering in a few small to medium sized projects.

With these facts in mind, there is no specific difference in how stakeholder requirements are initially gathered. However after having taken the initial requirement process steps, especially goal definition and context analysis, there might already be enough information suggesting a parallel solution and thus potentially influencing the selection of elicitation methods.

Quite regularly, it will be rather the case that parallelization is looming once the specified problem is given a closer examination by an expert in parallel programming - which is not necessarily the same person as the requirements engineer. This problem analysis phase is located in the playground of requirements elicitation, analysis and early high-level design.

4 Requirements Documentation

Requirements need to be documented. Apart from having a common base of information allowing discussions, reasoning and other forms of communications, there are other reasons, which have been widely described in publications, e.g. in [77]. Even more like with elicitation, documentation and specification has to be done more or less in detail and adherence to strict rules and regulations, depending on the size, complexity and importance of the project as well as the underlying development process.

In traditional RE, documentation starts with documenting the results of the steps preceding elicitation including the system context, stakeholder analysis and requirements source analysis. There are established documentation methods for these aspects (see [73], [71]), such as UML deployment or use case diagrams for visualizing the system context. For stakeholder management and keeping record of requirements sources, spreadsheet lists have proved to be useful.

An important thing is to create and continuously maintain a glossary covering all important domain and technical terms. This eases the communication between requirement engineers, stakeholders and developers.

4.1 Challenges

Raw, elicited data is seldom ready for immediate documentation. When having a large number of stakeholders and/or a complex system-to-be and/or underlying domain, a lot of elicited raw information and data needs to be thoroughly analyzed, refined, filtered and consolidated before being documented. Apart from seeking agreement among stakeholders (e.g. conflict resolution), this usually requires recurring activities of elicitation and document analysis and even might involve activities and decisions on the brink of system design when the required information cannot be delivered fully by the stakeholders or other requirements sources.

Traditional RE literature keeps a little bit quiet on this issue or provides only some very basic rules on refining ([73], chapter 9.11.)

Scientific efforts include the classic of the SEI [19] or try to provide partial solutions, e.g. Machado et.al [54] who describe the transfer of user requirements to architectural requirements with their "4 step rule set". Also, goal oriented approaches to RE (see chapter 5) provide some guiding in analysis refining.

However, even these methods cannot provide automated metamorphosis from raw information to a system specification. It is the task of the requirements engineer to accomplish this task, which of course also demands some creativity.

4.2 Selecting Documentation Methods

For documenting actual requirements, there is a broad range of documentation methods (and tools). There is no general rule which documentation is best for a specific kind of project. Literature suggests some guidelines and rules for making such a selection. Apart from general parameters, such as project size and complexity, the following aspects could be taken into account when deciding upon appropriate documentation methods:

- Intended Purpose / Targeted Audience: A major influence on selection of the form of documentation is the purpose the specification is authored for. If it is mainly directed at the human stakeholders it should emphasize on easy readability and comprehensibility, depending on the audience's capabilities. If automatic processing (e.g. code generation) is on the cards, a formal approach to a specification is more likely to be helpful.

- Acceptance: If the targeted audience of a requirements documentation consists (mainly) of humans, their acceptance of the selected documentation methods is a significant factor for selection of a documentation method as well.

- Acquaintance with notation: A direct influence to acceptance usually is the acquaintance of the subjected persons with the chosen notation. For instance, formal specification languages might not be suitable to document requirements for uneducated users, who might prefer natural language.

- Specification Level: Some documentation methods are reasonable for high level issues only (e.g. use case diagrams), whereas others are more suitable to capture fine details.

- Domain and Complexity of Issue: If a complex issue needs to be described, a mix of documentation methods might be appropriate. It is likely, that describing highly complex and/or technical issues with natural language only might lead to verbose and incomprehensible texts. More or less detailed (graphical) models may complement written text and thus increase comprehensibility.

- Effort: As cost is nearly always a constraining factor it has to carefully evaluated if the benefit of employing a certain method outweighs the expected effort. The question is always if there is a less costly method to reach the required documentation level.

- Securing Consistency: Another important issue when selecting a documentation method is the need of securing consistency. In principle, specifications based solely on natural language might be more difficult to keep consistent compared to modeled information. But consistence is not just about selecting a method, but is especially of significance for picking tool support. As the RE process is somewhat continuous, requirements changes may occur at any time during the lifetime of a project. These changes may affect consistency of the requirements information base. Especially in large projects with numerous interdependent requirements it is vital to have an adequate tool support helping to secure consistency.

- Unambiguity: For some purpose, a highly precise and unambiguous specification is required. For instance, this is the case for safety-critical systems, where any inconclusive requirements can have fatal consequences.

4.3 Documentation of Functional Requirements

Generally, FRs may be documented using natural language as well as in textual or graphical models.

4.3.1 Natural Language

A very common way of documenting requirements is the use of natural language, i.e. one forms FRs in sentences or in paragraphs. It is recommended to assign unique IDs to each single requirement, as it eases communication and aids maintaining consistency.

Basically, FRs may be formulated in an arbitrary way. However, it is essential not to become lost in complicated and potentially ambiguous verbalization and meaningless phrases. Rather than that, one should maintain a clear and consistent wording. This includes the careful use of words such as "must", "will", "should", etc. as these might induce legal consequences. Rupp [74] proposes a simple set of requirements (sentence) templates (Figure 4.1) based on linguistic patterns (first version dating back to 2004).

A similar approach extending the templates to other usage scenarios (such as business rules or use case steps) is suggested by de Almeida Ferreira and da Silva [32].

An older effort is the work of Duran [33], which introduced a table like structure similar to use case descriptions.

In their PABRE project, Franch et.al [39] [38] introduce their requirements patterns which target the reuse of requirements based on a requirements pattern catalog. In fact, their method raises the claim to reduce requirements elicitation to a mere search for a proper requirement in the catalog. This seems reasonable when it comes to NFRs (see below), but reusing FRs this way either requires a very

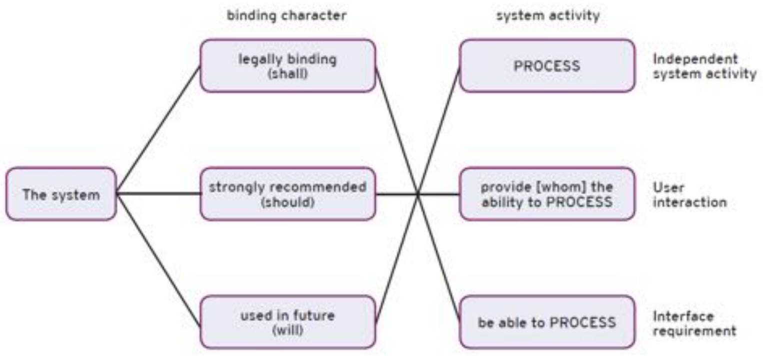


Figure 4.1: Requirements Template (Rupp)

extensive pattern collection or restricts the use of this method to similar domains or classes of software systems.

4.3.2 Conceptual Models

As an alternative to or complementing natural language, FRs may be documented in models. In RE, the most prevalent method is the Unified Modeling Language (UML) [10], which provides graphical diagram types covering static and also dynamic aspects of a software centric system. UML is extensible: SysML [7] for instance extends UML to support specification, analysis and design of systems in general, including e.g. hardware aspects.

In RE, nearly all of UMLs' 14 diagram types can be employed, though mostly used are Use Case diagrams and Class diagrams covering static aspects, while activity diagrams, state charts and sequence diagrams are to be used for dynamic aspects.

For documenting activities and processes, a dominant role is played by Use Case Diagrams and/or descriptions. Use case diagrams only provide a very abstract view on a system briefly depicting activities and interactions. Depending on the intended level of detail use cases are described verbally (e.g. by using a template) and may also be enriched with activity diagrams. Of course, other modeling techniques can be used as well: e.g. Event Driven Process chains (EPCs) [47], the Business Process Modeling Notation (BPMN) [8] or even Petri Nets for documentation of processes.

UML sequence diagrams or Message Sequence Charts may be used to depict protocol and interface requirements. For documenting important needs concerning states and state transitions of objects UML state diagrams are a popular choice.

A central role in RE documentation by modeling is played by UML class diagrams, especially when it comes to document the data model of the domain the system or software is supposed to work in. Class diagrams depict the static aspect of a system, as it shows the domain entities and their relationships. Identification of domain objects often requires some creativity and interpretation skills by the persons in charge with RE as ready-to-document domain models are usually not available.

A more abstract view is provided with UML deployment and component diagrams. These may be used for providing a higher level view of the system to realize. For instance a deployment diagram may be used to document the system context.

4.3.3 Formal Methods

For some purposes, natural language and modeling languages like UML may be to semantically "weak" for describing the requirements of a (software) system. This is the case with e.g. safety critical systems, like in health services or aviation, which demand a rigorous approach to software engineering. For these domains, formal languages are used to specify system requirements. Formal methods include Petri nets, special languages like *KAOS* (see also 5), *B* or the concept of Abstract State Machines (ASMs). For an overview, see [85].

Formal specifications provide clear advantages in being predisposed for consistency checks as well as validation and verification. Furthermore, they are an ideal base for automatic processing, such as code generation. However, there are also a lot of disadvantages. Apart from comprehensibility for non-professionals, specifying large systems formally is in itself error prone and tends to be costly. In addition formal specs are absolutely inappropriate for certain aspects of a software system (e.g. user interface requirements). See [76] for a discussion.

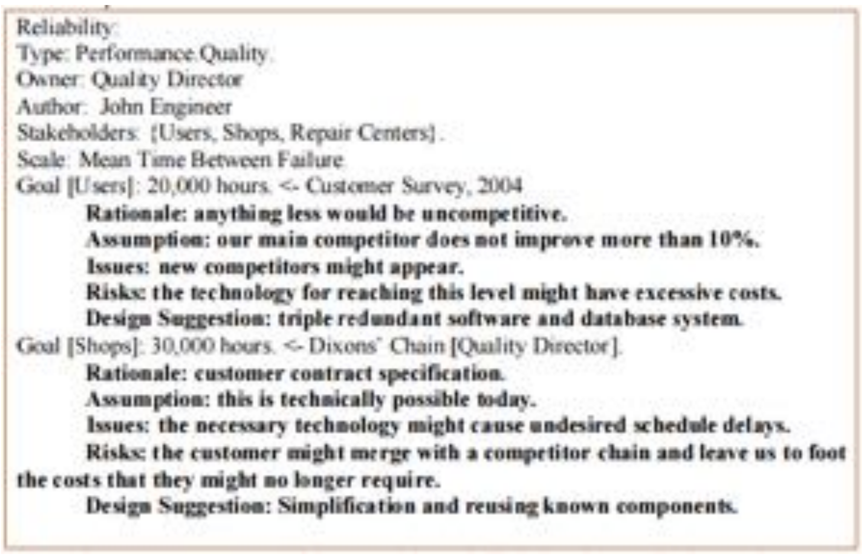
4.4 Documentation of Non-Functional Requirements

Like functional requirements, NFRs need to be documented, too. In practice it is quite common to formulate NFRs and constraints in textual form. This may be supported by the reuse of NFRs from a repository, as suggested by Rupp's IVENA approach [73], chapter 21.5.1 or the use of NFR templates and patterns as proposed by Franch et.al [39] in the PABRE project.

NFRs may also be incorporated into models. SysML for instance, has its own requirements diagram type, which may also be used to depict NFRs, constraints and their relationships. UML has no such diagram, here NFRs can often be found as annotations to functional models (use cases, activity diagrams,..).

Another means is "Planguage", a semi-formal language intended to be used for various planning artifacts during the software development process. One of its most advocated uses is the documentation of NFRs [41] [42] [55]. Figure 4.2 shows a sample of Planguage usage.

There is also the question of where to specify NFRs. If NFRs are specified separately or in context of a function depends on the process model followed and a possible specification template being used (see section 4.5). Some templates require NFRs being in their own structural section. However, especially when


Reliability:
Type: Performance.Quality.
Owner: Quality Director
Author: John Engineer
Stakeholders: {Users, Shops, Repair Centers}.
Scale: Mean Time Between Failure
Goal [Users]: 20,000 hours. <- Customer Survey, 2004

Rationale: anything less would be uncompetitive.
Assumption: our main competitor does not improve more than 10%.
Issues: new competitors might appear.
Risks: the technology for reaching this level might have excessive costs.
Design Suggestion: triple redundant software and database system.

Goal [Shops]: 30,000 hours. <- Dixons' Chain [Quality Director].

Rationale: customer contract specification.
Assumption: this is technically possible today.
Issues: the necessary technology might cause undesired schedule delays.
Risks: the customer might merge with a competitor chain and leave us to foot the costs that they might no longer require.
Design Suggestion: Simplification and reusing known components.

Figure 4.2: A sample of Planguage usage for a NFR (Gilb)

following approaches like IVENA, it is preferred to specify NFRs with "their" functional requirements.

4.5 Specification Templates

When specifying software systems of a reasonable size, the requirements documentation usually contains requirements in natural language and in form of conceptual models at least, if not formally specified requirements. Of course, it is possible to have one's own structure of a requirements document which in fact a lot of companies and institutions have. There is a number of established document templates and recommended guidelines for structuring specification documents.

A well-known guideline originating from the (meanwhile obsolete) IEEE 830 recommendation [3] is now an ISO standard (ISO/IEC29148) [6]. Amongst other things such as requirements concepts and processes, it describes the recommended structure of a software requirements specification and briefly outlines the contents of each section. However, the whole standard document stays a little bit superficial, so there is much room for interpretation and customization.

Other specification templates include suggestions such as the VOLERE template [70] by Robertson and Robertson or mandated artifacts of heavyweight software process models such as the V-Modell XT [21].

4.6 Documenting Requirements in Agile Projects

As already hinted in section 3.5, agile design processes and methods focus on FRs and initially practiced a rather sloppy handling of NFRs. At the heart of documenting requirements in agile methods is the User Story (US) [30]. User Stories describe requirements from the perspective of various system users in one or only a few sentences often following a specific sentence template.

As a <USER/ ROLE>
I want to <OPERATION/ FUNCTIONALITY>,
so that <BUSINESS VALUE / EXPECTED RESULT>

NFRs are - if at all - documented as special user stories called constraints. Of course, they should be linked to the depending user stories. Agile methods also feature some kind of acceptance criteria, which are applied when the implementation is presented to the customer representatives at the end of an iteration. In Scrum for instance, the use of sentence templates is widespread:

Given <CONTEXT/PRECONDITION>
When <ACTIONS>,
Then <REACTION / RESULT>

US are collected in a product backlog, where they are picked from by the developers. Coming from eXtreme Programming (XP) initially, US were meant to be a simple description of a chunk of functionality agreed between customers and developers, with an emphasis on "simple". A special form of stories is called Epics: These are stories that are on a high level and abstract needing refining or stories that are too big to be handled at once and need decomposition.

Under processes like scrum, rules have been introduced defining criteria for an US to be eligible for implementation ("Definition of Ready") [72]. Beck and Fowler [20], chapter v11 provides some similar points in their approach to writing stories.

According to Leffingwell, User Stories only concern the operational or team level of a software project. In his Scaled Agile Framework (SAFe) [9] and the preceding book [53], he adds two higher levels (Program and Portfolio levels) to accommodate a big picture of agile requirements engineering especially for larger enterprises. On these levels he uses different kinds of Epics for specifying requirements. However, it seems debatable if a vast collection of recommendations and specification like SAFe is still compatible with the principle of simplicity as defined in the agile manifesto.

In general, concerning agile approaches, simplicity is partly achieved by outsourcing requirements engineering. E.g. both XP and Scrum define the role of the customer (called product owner in Scrum) as being essential for driving the project, as it is this person, who provides the developer team with stories. Thus, the question is, where does the customer/product owner get the stories from? The answer is that there always must be some form of requirements engineering, which elicits requirements from stakeholders and documents them in an appropriate form. The difference here may be revealed in the detail level, since in agile processes details are usually negotiated during an iteration on team level. But then, resorting to a traditional design process does not necessarily constitute the need of an overly detailed specification.

4.7 Requirements Documentation for Parallel Applications

Following the assumptions from section 3.6, there is no necessity of a special documentation method if the requirements do not specifically refer to a mandated or suggested parallel solution. However, when proceeding through refining, analysis and design of a specified problem, more and more characteristics of parallelization may surface, which may require appropriate documentation means.

Like with sequential applications textual descriptions as described in 4.3.1 are always possible. Of course, modeling is a possibility here too: For documenting parallelism in processes and use cases, UML provides a basic means [64].

Behavioral descriptions like activity, sequence and collaboration diagrams are suitable to document simple parallelization issues, but also static aspects like node distribution or process topology can be sketched. To depict more complex issues or to reflect special aspects of parallel applications UML can be extended by introducing special UML stereotypes or even additional model elements and relationships through an UML profile. Examples for such extensions are shown by Pllana and Fahringer [67], who intend to address the prediction of performance figures, and Labbani et.al [50], who use an own UML profile for modeling applications mixing control and data parallel processing.

Certainly, parallel problems may be described with formal methods too. An example is described by Younes and Ayed [86], who generate Event-B descriptions from UML activity diagram for the purpose of verification.

Given the fuzziness of delineation between requirements engineering (analysis) and subsequent application design, we might even include pattern-based high-level programming languages here, such as the high-level pattern expression languages suggested by Steuwer [78] or the projected *RePhrase* pattern DSL [11], chapter 7. We will expand on this when discussing design processes for parallel applications in section 6.3.

The next section deals with a field of alternative approaches to RE, which centers around the identification and analysis of goals.

5 Goal Oriented Requirements Engineering

Besides the traditional and agile approach to RE, there are other processes and methods which have gained less practical prominence, but nevertheless enjoy a fair share of reception in the scientific community. A number of these methods is subsumed in the field of Goal Oriented Requirements Engineering (GORE).

A good overview on GORE is provided by van Lamsweerde in his "guided tour" [82]. Another overview is provided by Lapouchnian [51]. In fact the definition of the term goals is quite the same, that has been given for requirement (even the functional/non-functional taxonomy) in traditional RE research. However, GORE emphasizes the hierarchical decomposition of goals plus their mutual influence and dependencies and only refers to "realizable" goals as requirements.

In the following paragraphs, we want to sketch a very brief picture of the more interesting approaches. For more details, refer to the cited publications.

5.1 Selected GORE Approaches

5.1.1 NFR Framework

An early work aiming obviously mostly at NFRs is the NFR framework [27], taking a systematic approach starting at imprecisely formulated NFRs, which are called soft-goals (depicted as clouds, Figure 5.1). In a second step, these are further refined hierarchically ("+/-" contribution links).

Subsequently these "NFR soft-goals" are complemented by operational soft-goals resembling more or less functional requirements or more abstract means to support or even impede these soft-goals (depicted as clouds with bold lines).

Chung et.al merely see the developer being in charge with the process of applying the steps propagated by the NFR framework. Only the goal refining and decomposition phase suggests the involving of other stakeholders.

In addition the NFR framework only speaks about soft-goals, clearly contradicting other RE approaches mandating NFRs to be evaluable. This is an issue, which has been addressed recently by e.g. Yrjonen and Merilinna [87].

In addition, operationalization is a somewhat creative process that has to be

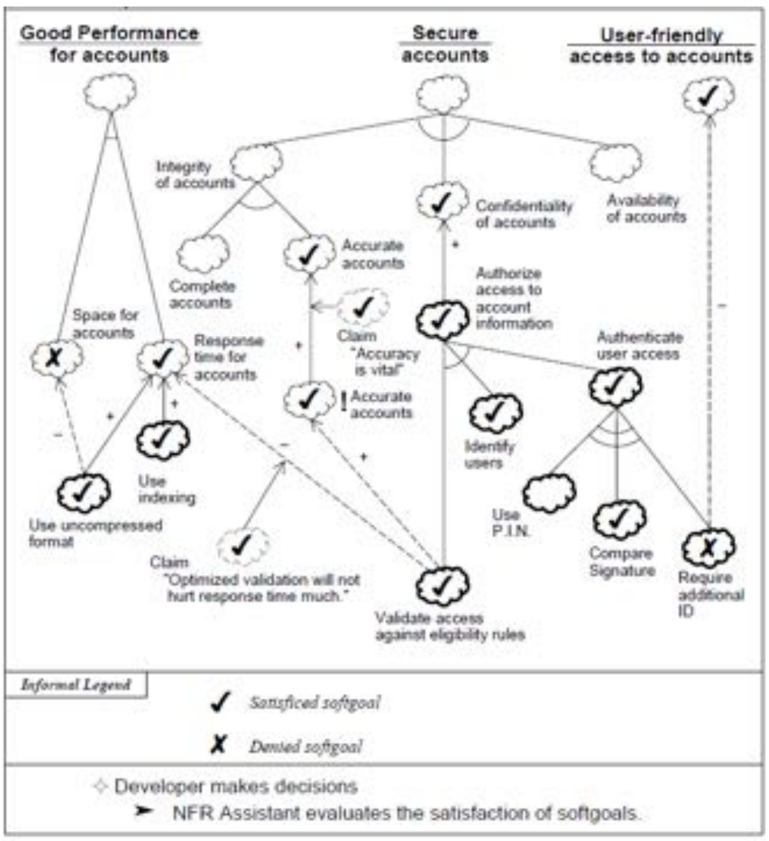

Good Performance for accounts
Secure accounts
User-friendly access to accounts
Integrity of accounts
Confidentiality of accounts
Availability of accounts
Complete accounts
Accurate accounts
Authorize access to account information
Space for accounts
Response time for accounts
Claim "Accuracy is vital"
! Accurate accounts
Authenticate user access
Identify users
Use indexing
Use uncompressed format
Use P.I.N.
Compare Signature
Require additional ID
Claim "Optimized validation will not hurt response time much."
Validate access against eligibility rules
Informal Legend
✓ Satisficed softgoal
✗ Denied softgoal
Developer makes decisions
NFR Assistant evaluates the satisfaction of softgoals.

Figure 5.1: NFR Framework smaple goalgraph (Chung)

done by developers/architects and it remains to be debated if this is in the responsibility of RE. Altogether, the NFR framework is more of an early phase design method focusing at requirements analysis. There is also a tool supporting NFRF-style graphs (amongst others) [80].

5.1.2 User Requirements Notation (URN)

While the NFR framework is more than a notation, another variant of GORE focusing more on documentation is constituted by the User Requirements Notation. This standardized notation [1] aims to cover two aspects of RE. On the one hand it provides a graphical notation for technical scenarios (Use Case Maps) and on the other hand a tree-like Goal-Oriented Requirements Language (GRL) looking quite similar to NFRF diagrams. The GRL is based on the i* Framework as proposed by Eric Yu [88] and the NFRF.

Amyot and Mussbacher [17] give a good introduction into the purpose and capabilities of these two diagram types as well as an excellent overview of scientific effort done in conjunction with URN. In addition, URN has tool support. There is an UML profile [12] and there is also a standalone tool available [16] for drawing URN diagrams.

5.1.3 KAOS

Another GORE method is KAOS [31] [84]. It combines semi-formal goal model graphs with a formal approach to goal analysis and reasoning. KAOS features four steps (see figure 5.2):

- Goal model: Identify and refine goals. A goal model in KAOS is more or less an AND/OR graph.
- Object model: Identify objects and actions from goals. Also serves as a main source for a glossary.
- Responsibility model: Derive requirements on the objects and actions to meet the constraints.
- Operational model: Assign constraints, objects and actions to agents.

The steps are non-sequential but thought to be processed iterative, i.e. progress in refining the goal model may prompt for further refining of the other models.

For modeling goals, initial input is required, which would typically be some initial problem statement and/or information coming from various elicitation efforts. Goals shall then be set up by looking up keywords and seeking rationale for statements or required functionality (why? how?). KAOS distinguishes between functional and non-functional goals, with the latter being quite similar to what we defined as NFRs and constraints in section 2.3. Goal modeling of course requires not only input from "outside" at the beginning. Usually refining involves steady

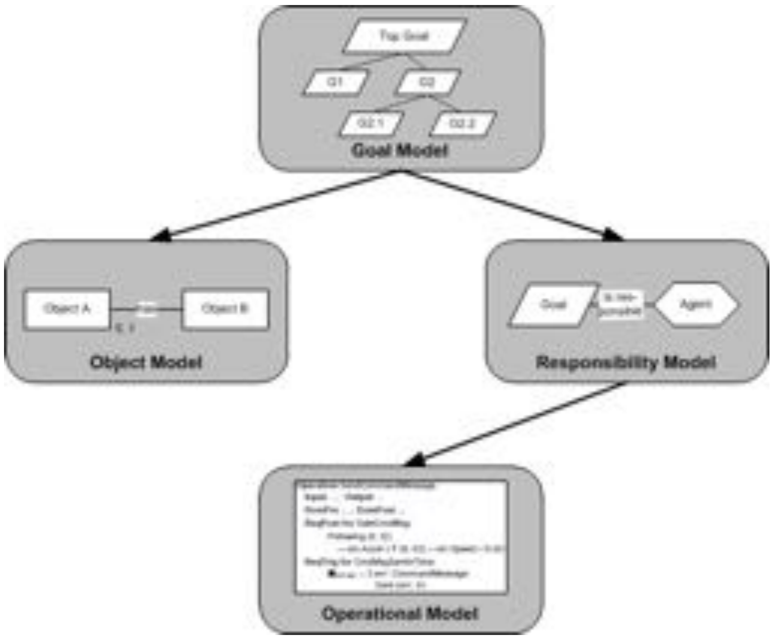


Figure 5.2: The four steps of goal oriented RE with KAOS (van Lamsweerde)

contributions from stakeholders. To assist in goal decomposition, one may use the so called generic goal patterns, more or less being goal graphs referring to some abstract problem (see figure 5.3).

Goals on Top Level may go up to strategic business goals, while at the leaf goals can be very fine grained technical goals. Once the goals being specific enough, the object model can be derived by identifying objects, their attributes and the relationships from the modeled goals. Here, the object model is quite the same as what is specified by traditional RE through the domain model.

Similarly, the responsibility model can be derived by identifying active objects (called "agents") from the goal model. Such active objects may be software components, but also human users. Agents are assigned to "terminal" goals (goal tree leaves), which are then called requirements, except if the agent is not part of the software rather than its environment then the goal is called an expectation.

In a final step, the requirements and expectations are then operationalized, i.e. the models are used to identify necessary operations to fulfill the specific requirements and expectations.

For details, refer to some case studies [81] or the KAOS tutorial provided by tool vendor Respect-IT [5].

5.1.4 Discussion of GORE Approaches

GORE approaches might seem a complete paradigm change at first, but in fact they capture the same phenomena like traditional RE does. GORE approaches also

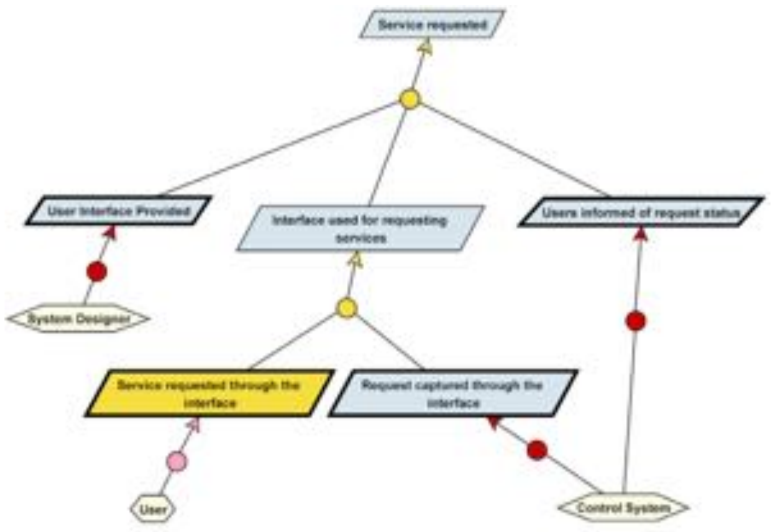

Service requested
User Interface Provided
Interface used for requesting services
Users informed of request status
System Designer
Service requested through the interface
Request captured through the interface
User
Control System

Figure 5.3: Possible Goal Pattern for a service request [5]

need to resort to elicitation and even analysis techniques used in traditional RE. In addition, goal elicitation and documentation is of course an issue with traditional RE, too. A difference here is the mandatory measurability of goals in traditional RE [71], as opposed to the presence of the so called soft-goals in GORE approaches. Moreover, some authors see GORE as a vital supportive part but not the major aspect of RE [68].

What makes GORE approaches appalling is that they strongly emphasize the focus on goals and the reasoning about goals in terms of analysis and design. In other words these methods seek to provide a comprehensible guideline to a solid analysis of elicited information and the subsequent preparation of designing the system architecture. There are numerous publications on paving the way from GORE models to system architecture [28], [25], [60], [83].

For instance, GORE methods provide criteria for determining (sufficient) completeness of a requirements model. In KAOS, this is usually achieved when every terminal goal has been assigned an agent and thus is converted into a requirement or an expectation [5]. However determining completeness of the goal decomposition still is a remaining issue. In addition, one might argue that GORE approaches are crossing the line from RE to system architecture design, but of course there is indeed a gray area between these systems/software engineering process steps.

For really large and complex systems, the extent and number of goal diagrams might become overwhelming, especially when dealing with goal dependencies and contribution modeling.

6 From Requirements to Solutions

As pointed out, there is a smooth way across from requirements engineering to system architecture design. This is also somewhat reflected in the IEEE standard for RE. It sketches several processes in the context of RE:

- Stakeholder requirements definition: deals with elicitation of requirements from stakeholders.
- Requirements analysis: seeks to transfer the stakeholder centric requirements into more technical system requirements.
- RE activities in other processes: includes RE activities in architecture/program design as well as in validation and verification.

In this chapter we will shed some light on analysis and RE in architecture design. There are numerous publications both academic and practical that deal with this topic. Noteworthy efforts include Carnegie Mellon's ADD (Attribute Driven Design) [19] or - especially for large scale enterprise architectures -TOGAF ADM [2]. Also, agile, lightweight approaches to application design have their more or less restrictive rules and guidelines.

Apart from the latter, there is a broad consensus that application design is heavily influenced by NFRs (and constraints).

6.1 The Impact of NFR on DESIGN

When considering NFRs one has to be aware that there is a difference between the NFR itself and possible means or options to realize them. Such means may be additional functional requirements, design/architectural constraints or even measures influencing the software development process. Paech [63] identifies three issues concerning realization options:

- Identification of possible options
- Evaluating NFRs compared to the options
- Trade off analysis: decide between several options

Identification of possible options usually requires creativity and experience. Applying architectural styles or patterns is a popular means to support the discovery of possible realization options. A challenge here is that pattern catalogs usually emphasize the structural and functional aspects of the patterns but do not provide information concerning the contribution of the pattern to the fulfillment of given NFRs or constraints. We will discuss this a bit more in detail in section 6.2.

Evaluating refers to the necessity of clarifying how much a considered option contributes to the fulfillment of the given NFRs. This requires a preferably exact specification of the NFRs including dependencies between the NFRs. Evaluation is followed by trade-off analysis: The options are discussed on the base of the evaluation results and subsequently the best option may be chosen. For evaluation and tradeoff, there are several methods ranging from holistic, software architecture centric approaches like ATAM [19] to common purpose decision support methods like CUA or AHP [75]. Also, GORE approaches (see chapter 5) support trade-off analysis by allowing to create reasoning structures on the base of goal models (see [46], [28] or [60]).

6.2 Patterns in Application Design

6.2.1 Introduction

Patterns have a long tradition in SW engineering. The most notable and widely known works are that of Gamma et.al [40] and Buschmann et.al [22]. Another important effort in "traditional" software engineering has been published by Fowler [37], which addresses enterprise level patterns including database backend and (Web-)UI related patterns.

In general, a pattern is a structured description of an abstract or exemplary solution to recurring design issues. Such a structured description is common to all pattern languages. Known pattern catalogs provide the following information in their descriptions:

- Pattern identification: Usually, a pattern has a significant and well-known name. Also in this section, there may be a classification of the pattern as well as a brief and short characterization of the pattern. The purpose of this information is to define a common vocabulary, with the full set of patterns constituting the pattern "language". This is also valid for looking across multiple pattern catalogs. It is not helpful to invent new names for already labeled, established patterns.

- Context, Problem and Applicability: Each pattern needs to describe the situation, in which it may be or where it is advised to be used. This is often referred to as "Forces". Apart from contextual information and forces, these sections may contain descriptions of one or more abstract or even concrete, real life problem situations. The description may also include constraints that have to be fulfilled by a pattern implementation. More or less, these information sections describe the circumstances under which the pattern can be applied.

- Solution description: Provides a full description of the proposed solution components. This includes static as well as dynamic aspects, including dependencies, relations and interactions. The solution description is usually not pointing out a specific implementation rather than a kind of template, which may be applied when encountering a challenge fitting to the problem description. In addition, a solution may of course be complemented by a more concrete code example.

- Consequences description: Describes the effects of the pattern application, e.g. resulting benefits or disadvantages. Consequence descriptions help a developer to realize the implications of his or her design decisions. This may play a role, when having to choose between two or more otherwise equivalent options.

Similar structures can be found in other pattern languages. Depending on author and pattern domain, these four sections are more or less detailed and divided into more subsections.

6.2.2 Patterns in Parallel Software Development

In parallel software development, notable efforts in the fields of pattern based design include Mattson et.al. [56], Keutzer and Mattson [49], Ortega [62], McCool [58] [59] and the ParaPhrase project, the latter being the predecessor of this project.

From the requirements engineering point of view, the problem description section is the most significant aspect. With its help, an application designer or programmer shall be able to decide upon the suitability and applicability of a pattern for his specific problem.

All of the cited pattern collections have in common that pattern aspects are described in a more or less informal, textual form. Only McCool refrains from structuring the pattern characterization. He eschews any pattern internal characterization structure and sticks to very brief and simple textual descriptions, but deals in detail with selected patterns.

These textual descriptions imply a predominantly human developer driven design and pattern selection.

Especially in parallel programming, the patterns are specified on different levels: On the one hand patterns describe (abstract) solution concepts for a class of problems, whilst others are closely related to a specific implementation - such as in the FastFlow pattern framework [15])].

An "excuse" could be the presence of *Algorithmic Skeletons* as another important concept of abstraction in parallel program design. Gonzales [44] provides an extensive survey of existing skeleton frameworks and provides a distinction to other abstraction concepts like patterns. However, his distinction remains somewhat unclear - in general, the usage of these terms seems not to be consistent. In the RePhrase project, we follow the view of Algorithmic skeletons (see [11], section 2.1) being closer to the implementation, while patterns providing a more abstract view on recurring design issues.

As an example we might have a look at the popular pipeline pattern:

- In their pattern specification, Mattson et.al [56], section 4.8 deliver a conceptual and extensive, but realization technology agnostic description. The implementation section gives a basic example of an implementation in Java.

- A more compact description of their pipeline pattern is provided by Lee [52] for the pattern language "OPL" [48], developed by the Berkeley Parallel Computing Laboratory.

- In their book, McCool et.al [59] firstly deliver only a very brief overview of the pipeline pattern, but then dedicate a full chapter to this pattern, providing detailed information on possible implementations (in frameworks like Intel TBB) and also describe some pipeline invariants. However, unlike other authors, McCool does not use a structured template for his pattern descriptions.

- In contrast, the description of the pipeline pattern (in the sense of a pattern language) in FastFlow is only rudimentary, it is merely a documentation of the implementation (in C++) - which uses the term skeleton anyway.

- In the ParaPhrase project pattern collection [23], the description of pipeline is also kept short and simple, but it is completely independent from any implementation. The implementation section just refers to some implementations in other projects, such as "Muskel" [14].

Clearly, the expressiveness and meaningfulness of the pattern description - especially the problem description - is a vital criterion when it comes to picking a selected pattern from a catalog.

Most of the pattern languages are not bound to a specific design process and are one vital means during architecture and application design. For instance, applicability of the GoF patterns is possible in heavyweight design approaches as well as in agile software processes. This is also the case with pattern collections in development of parallel programs. Still, there are a few pattern languages proposed in close relationship to a design process or guideline.

6.3 Design Processes for Parallel Applications

Designing parallel applications is not always trivial and as such does not work by picking only one pattern per problem from the pattern catalog. Complex, often non-embarrassingly parallel problems usually require a combination of patterns and

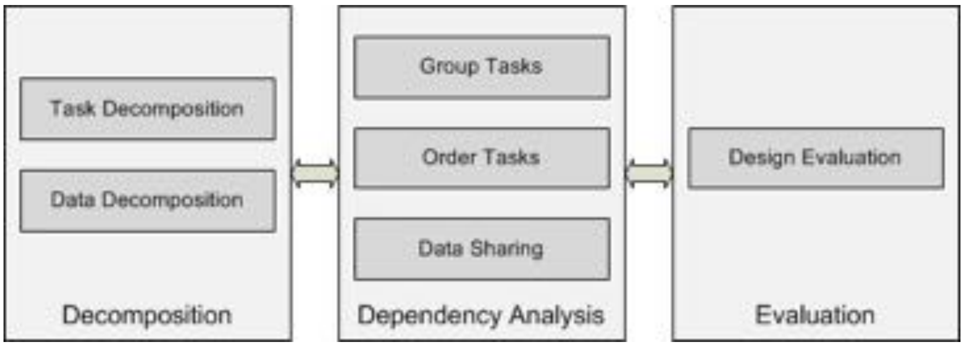


Figure 6.1: Finding Concurency Design Space (Mattson)

techniques to be used. To provide some guidance, some pattern based approaches are complemented with a more or less defined design process.

These design processes do not address solving a particular domain problem specified, e.g. finding the right optimization algorithm for some production process. They just assist in solving a subject-specific problem via parallel processing. In the RePhrase project, the definition of such a design process is also part of the envisioned methodology. In this section, we will have a look at other work with a focus on requirements related issues.

6.3.1 Informal Pattern Based Design

6.3.1.1 Approach by Mattson

In their book, Mattson et.al. [56] introduce the notion of design spaces, each addressing a certain phase in the design of a parallel solution. The first design space "Finding Concurrency" (see figure 6.1) deals with analysis of the problem to solve. The patterns provided here resemble analysis and design process step recommendations sharing a similar scheme rather than patterns in the sense of design or implementation.

During finding concurrency, Mattson et.al recommend the consideration of non- functional influences as flexibility, efficiency and simplicity in general as they deem these attributes as decisive for the quality of early design of a parallel solution. The subsequent design spaces "Algorithm structures" and "Supporting structures" build on these findings. For the algorithm structures, dominant pattern selection criteria are the findings from decomposition weighted against the target platform characteristics. As the latter may include constraints issued by stakeholders, other non-functional factors play a neglectable role, only occasional references to NFR are made in pattern descriptions. Supportive structures are selected upon the chosen algorithm structures. Mattson et.al. claim a preferred match of certain communication structures (e.g. "Fork/Join") to algorithm structures (e.g. "Pipeline"). Another selection criterion is the degree of a support structure in diverse programming models. Here, HW and SW constraints might come into consideration again.

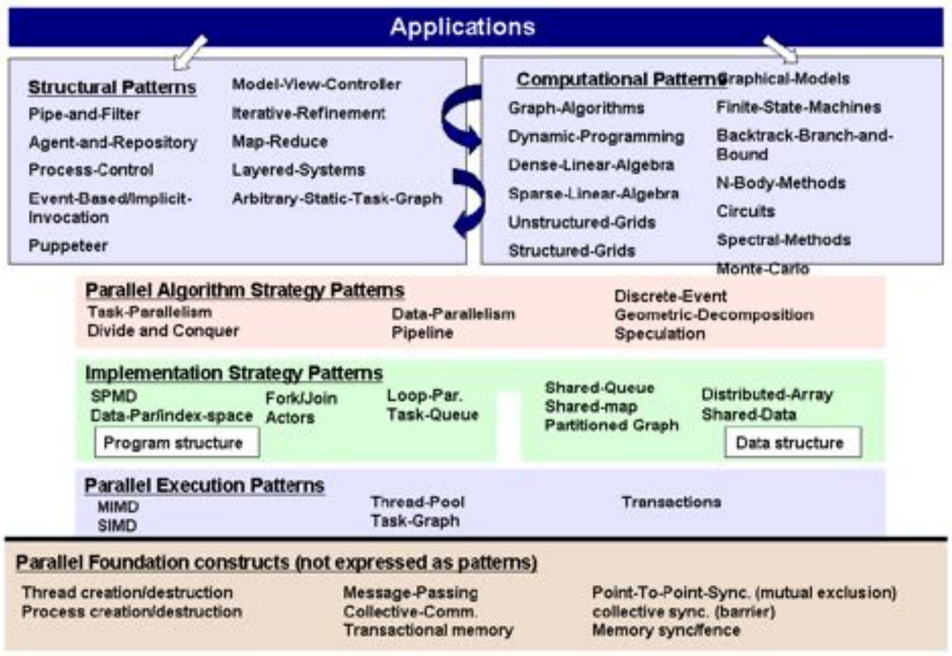


Figure 6.2: Structure of the OPL (Keutzer et al.)

However, they do not further elaborate on how pattern selection is influenced by other diverse NFRs.

6.3.1.2 The Berkley Parallel Laboratory Pattern Catalog

In subsequent work done at the Berkley Parallel Laboratory, Mattson's pattern catalogue has been extended and integrated into the already mentioned OPL pattern language [49] depicted in figure 6.2. Building on parallel foundation constructs (such as thread and process handling) not expressed as patterns, pattern layers representing pattern types or categories - with each category containing a number of specific patterns - are presented as a construction kit for parallel applications. On top there are computational patterns representing various computational problem classes and strategies of how to solve them by parallelization. These are heavily intertwined with the class of structural patterns acting as principal architectural styles for parallel programs.

Further down , there are patterns for selecting an appropriate parallelization strategy for an algorithm and the right implementation strategy. Above the foundation layer, basic parallel execution patterns are provided, which resemble the execution models of available hardware. According to Keutzer et al. these lower three layers are somewhat connected, e.g. a specific algorithm strategy advising a certain implementation strategy.

There is no explicitly defined design process or guideline. OPL is just a number of pattern descriptions classified in the described categories with a pattern from a higher layer (not always) pointing out possible "matching" patterns from lower layer. The description of the patterns use a consistent section structure, however, there is a difference in formatting, content extent and quality - even within a specific layer. In addition, the language contains some patterns, which do not belong into the field of parallelization, such as the MVC-pattern.

Altogether, the OPL pattern set published at the project website (*http://parlab.eecs.berkeley.edu/wiki/patterns/patterns*) seems like an unfinished business and, as the Berkeley Parallel Lab has been closed in 2013, it is unlikely that there will be further development. Nevertheless, the organization of the OPL provides some useful insights for discussing our purpose (see chapter 7).

6.3.1.3 Approach by Ortega

Another pattern based design process has been published by Ortega [61] [62] Ortega counts performance and cost as the most relevant NFRs for designing parallel solutions with cost being primarily related to the effort of implementing the solution. Furthermore, reliability and maintainability are quoted as additional significant factors in the design of parallel applications.

The design process consists of four steps, starting with an analysis of the given problem. It is this phase, which has the closest ties to requirements engineering, as it relies on the continuous participation of the relevant stakeholders, which are responsible to deliver the necessary input for the problem statement in form of requirements. The purpose of the problem analysis step is to provide a problem specification document which serves as a foundation for subsequent phases.

In the beginning, requirements are collected from stakeholders, discussed and aggregated into an informal problem statement document. In a subsequent step the problem is analyzed in more detail - already mainly conducted by the parallel application developer. Certainly, this specification task usually involves stakeholders as well for refining and validating collected requirements and the analysis results.

The final problem specification document shall describe the problem precisely and shall contain descriptions of the domain algorithms and data structures. In addition it shall cover information about the parallel platform being used as well as quantified requirements about performance and cost. Information concerning the platform refers to the hardware and software constraints limiting the solution space for following design steps.

Besides the suggested performance and cost requirements, other non- functional requirements may be specified here, too. Ortega hints, that these NFRs have to be quantified, but remains silent in his examples: Neither the two case studies nor the example demonstrating the design process specifies measurable or testable requirements.

In addition, Ortega is barely referencing these requirement statements as he is proceeding through the design process. Only in the final step evaluation these requirements are used for testing the resulting implementation, thus treating them as acceptance criteria rather than an influencing factor in design.

This specification serves as input to the following design phases of coordination design, which addresses the selection of fundamental architectural patterns and communication design, whose purpose is to find communication patterns, i.e. appropriate mechanisms to exchange data between the participants in the designed architecture.

A "Detail Design" phase puts it all together by deciding upon issues like synchronization mechanisms.

In a final evaluation the implemented parallel application is tested for the fulfillment of the requirements specified during the problem analysis phase.

Like Mattson, Ortega proposes selection criteria for his architectural and communication patterns (e.g. type of parallelism and type of processing for architecture, type of synchronization / type of memory organization for communication) and provides some basic steps for accomplishing the selection. These are merely rough guidelines rather than a detailed process step specification. Although Ortega recommends the quantified requirements section in the problem specification, he barely references it through the design process. Only the specified constraints ("target platform") play a role during detailed design and implementation, while performance goals are mentioned in context with evaluation only.

6.3.2 Rewrite Rule Driven High-Level Design

The approaches by Mattson and Ortega both mandate a highly skilled developer to tackle the design of a parallel program. There, a developer needs to take care of very low level issues like communication and synchronization. Various skeleton frameworks have made efforts to ease this by providing high-level semantics to construct parallel solutions, yet still these approaches are very implementation focused.

6.3.2.1 The RePhrase Approach

This is where the RePhrase vision draws on. It looks to employ a domain specific language based on a set of patterns for constructing high-level solution expressions. These expressions may be annotated with additional non-functional properties. By subsequently applying specific rewriting rules, it is possible to seek functionally equivalent expressions. Following that, code is generated for the targeted parallel platform.

Both of these steps take the annotations made on pattern combination expressions into account when picking their possible set of solutions. By applying specific refactoring techniques under consideration of efficiency measurement results the ensuing application code shall be optimized for the targeted hardware. A possible procedure for developing a new parallel application from scratch may look like this:

1. At the beginning of the design phase, a developer has no more than a requirements specification.

2. During problem analysis the developer designs a parallel solution by writing pattern expressions following the notation and available pattern terms proposed by the DSL.

3. In addition the developer may use annotations for specifying non-functional properties, which may act as hints for application of rewriting rules and the subsequent selection of suitable implementation skeletons.

4. After having designed the logical solution expression, pattern rewrite rules are be applied by the developer and/or automatically by the development tool to conform "logically significant" annotations. As a result the developer gets one or more functionally equal, but still logical (in the sense of implementation agnostic) solutions for the specified problem. The evaluation of nonfunctional property models against annotations already plays a role at this stage.

5. In this step, the implementations for the patterned solution is chosen and parametrized by the developer - ideally with tool support - by taking further "implementation-significant" annotations or other requirements into account. Choosing an implementation would again be supported by the non-functional property models, which are matched to annotation values.

Finally, the developer completes the implementation skeletons as needed.

6.3.2.2 Related Work

A similar, interesting approach has been published by Steuwer et.al. In [78], they propose a three staged design process for parallel applications, which has strong similarities to the RePhrase vision. The main target is to allow the creation of highly performing and portable code out of a comparatively simple set of basic patterns - or algorithmic primitives as they are called here - by application of expression rewrite rules. The targeted implementation layer here is OpenCL, which in turn is an abstraction of parallel hardware specific implementations. Figure 6.3 shows an overview of the proposed approach.

The programmer or application designer is poised to work with only a small set of well-known algorithmic primitives, such as "Map", "Reduce", etc. He analyzes the problem to solve and creates the appropriate algorithmic expression.

In a second step this expression is subjected to a set of rewrite rules transferring it to an expression consisting of OpenCL primitives. Depending on the initial expression, there might be not one OpenCL expression, but a tree of options representing different possible solutions.

In a third step, code is generated for each of the solutions by employing a random based search strategy. The performance of each solution is measured and

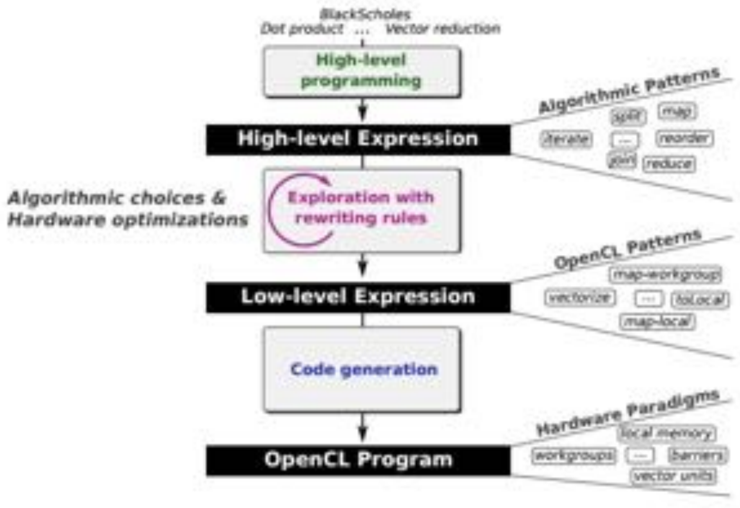


Figure 6.3: Generating Performance Portable Code (Steuwer)

the one with the best performance values is picked. Depending on the complexity of the code and the solution space, these search operations may be immensely time consuming. Steuwer et.al do not mention possible annotation of his high-level or low-level expressions, for possibly supporting his search method for a solution.

Unlike RePhrase, which looks to support different parallel frameworks like FastFlow, OpenCL or OpenMP, Steuwer's approach seems to focus on OpenCL.

7 Requirements Engineering within the RePhrase Pattern Based Design Process

In this chapter we will now put the pieces together and discuss possible answers to our research questions.

7.1 On the Structure of a Pattern Set and Pattern Descriptions

Firstly we will discuss important aspects of a pattern set's structure and the description of individual patterns. Furthermore, we discuss how these characteristics impact pattern selection and also take a look on the intended RePhrase approach using rewrite rules on algorithmic expressions in design of a parallel solution.

7.1.1 Pattern Sets and Design Process

As our brief look on design processes shows (see section 6.3), we could identify two principal approaches in developing parallel applications. This bears an obvious influence on the organization of the pattern catalog itself. Informal, developer-oriented approaches (Mattson, Ortega) tend to focus on architectural patterns and patterns of communication mechanisms, while approaches looking to support non-programmers concentrate on an algorithmic view relinquishing architecture and communication (at least on top level).

Thus, the structure of the pattern set should match the design process it intends to support. This of course mandates the presence of such a process or at least a guideline, which instructs the targeted user how to pick and apply the patterns and which patterns are intended to support which phase of application development.

7.1.2 Delimitability of Patterns

Selecting a pattern is about choices, which means having at least a reasonable number of potential alternatives. Furthermore, to be able to select amongst alternatives it is necessary to have a distinct characterization. The existing pattern sets concentrate on functional aspects in their problem descriptions. In other words, applicability is decided solely on functional characteristics.

In section 6.1 we have quoted the influence of NFRs on solution architecture. The views discussed originate out of the classic software engineering community and mostly concern design and development of large software systems. But being able to pick a pattern amongst functionally equivalent alternatives by weighing non-functional characteristics seems desirable in designing parallel programs, too.

This would mandate a distinct description based on significant non-functional characteristics on otherwise functional equivalent pattern alternatives.

Looking at the descriptions RePhrase initial pattern set from [11], this seems unlikely, though. The members of the pattern set are described in a very compact and abstract way. The pattern alternatives can easily be delineated by these functional semantics, thus there is no use for discriminating by NFRs. In addition, when taking into account the highly abstract pattern outline, it seems hardly conceivable to find non-functional and distinctive characteristics.

Tying concrete efficiency characteristics to an otherwise abstract pattern makes little sense. E.g. computational complexity might be specified for an algorithm, but not on an abstract description of a map pattern.

7.1.3 Delimiting: Reasoning Structures

Let's take a look at the farm pattern from Rephrase D2.1 [11], section 3.2.1. The implementation description mentions two alternatives, which for a moment we now assume being separate patterns "Active Worker Farm" (A) and "Passive Worker Farm" (B). Both would be equivalent given their present functional semantics. If we can find several distinct non-functional characteristics to weigh out these two options then a distinction would seem justifiable. Finding just one characteristic would not be sufficient - there must rather be a reasoning structure of more than one characteristic. For instance, labeling pattern A as being generally more performing than option B would render pattern B superfluous.

The question is whether statements regarding the potential fulfillment of non-functional requirements can be issued in a pattern description and how concrete these statements may be given the abstract level of the original pattern description.

For instance, there is no evidence of how to anticipate the *maintainability* of patterns A and B on the current abstract level of description. And of course this is difficult as maintainability usually relies on code complexity measures like *Halstead* or *McCabe* metrics and thus is dependent on the implementation's source code. Naturally this lack of appropriateness holds for all internal quality attributes that target the quality of source code.

As another example, let's pick the characteristic of *reliability*. Typical metrics for reliability (precisely, its fault tolerance aspect) include MTBF/MTTF/MTTR key figures. These figures have even more influencing factors: Apart from the implementation, reliability strongly depends on the deployment context and the underlying target system. In addition, reliability cannot always be measured on the fly - it might need longer observations to determine these figures on an individual system.

A further NFR mentioned in ISO 250xx is *functional suitability*, which includes aspects like accuracy and appropriateness. There is no evidence how to label abstract patterns like Map, Reduce or Pipeline with figures of fulfillment potential.

More likely to be properly characterized by "external" quality attributes that performance, reliability and functional suitability belong to, are concrete computational methods or algorithms that target a specific problem, e.g. a heuristic method to solve an optimization problem compared to an exact one. Another example would be to classify upon sorting algorithms.

7.1.4 Goal Graphs as Reasoning Structure

Building a consistent reasoning structure must be on the cards for an easy to handle pattern set in terms of RE. In chapter 5, we introduced goal oriented requirements engineering (GORE). GORE could be a possible means to create such reasoning structures. A publication by Gross and Yu [46] suggests the use of goal graphs for visualizing such reasoning structures in a consistent way. They showed how to build a goal based reasoning structure for object oriented patterns by example of the popular Observer pattern. There are basically two steps in this approach, first starting by pointing out the characteristics of the pattern in a goal graph. This is achieved by analyzing existing textual descriptions or discussion. Figure 7.1 shows the proposed characteristics of the Observer pattern.

Thin lined clouds represent the soft-goals, bold lined clouds represent operationalizing goals. The links represent the contributions and influence between individual goals (Make, Hurt, Break).

For performing a tradeoff analysis, in a second step the goal graph is extended by the goal graph of a possible alternative solution (Figure 7.2).

The discussion of the different positive and negative contributions to the identified soft-goals (which may be of functional or non-functional character) delivers a base for a decision concerning the selection of a pattern.

There are several other efforts of structuring design decisions this way. While Gross and Yu's approach is based on the *i** notation, Mussbacher et.al propose the use of the GRL for this purpose [60]. Figure 7.3 provides an abstract GRL based reasoning structure example. The pattern contributes to several forces (modeled as soft-goals), which have more or less correlation relationships and in turn contribute to raised requirements.

Another work elaborating on quantitative reasoning with goal-graphs (based on the NFR framework) has been published by Supakkul et.al [79]. Of course, such reasoning structures could be achieved with the discussed KAOS approach, too.

A similar reasoning approach could be followed, when choosing an implementation affine pattern (skeleton/idiom). Here even source code related NFRs

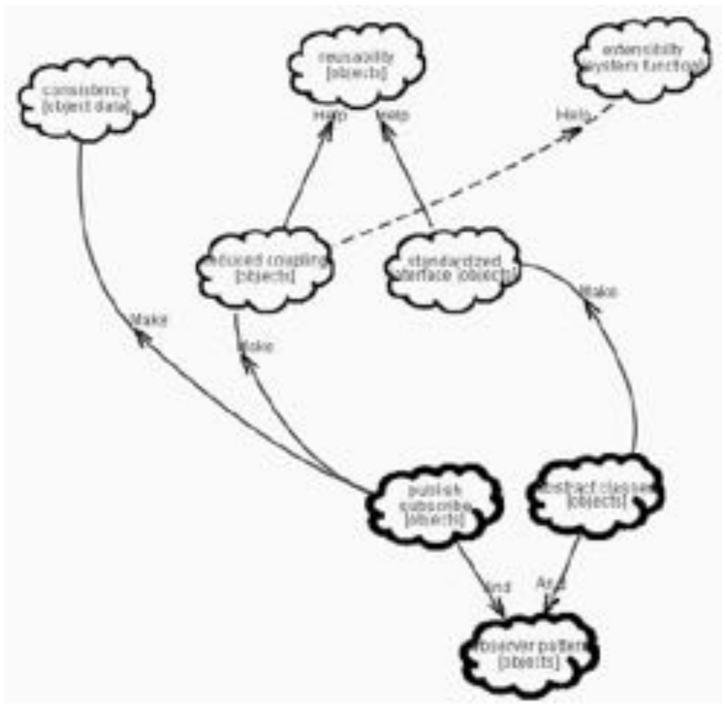

consistency [object data]
reusability [objects]
extensibility [system functions]
Help
Help
Help
reduced coupling [objects]
standardized interface [objects]
Make
Make
Make
publish subscribe [objects]
abstract classes [objects]
And
And
observer pattern [objects]

Figure 7.1: Observer patterns properties as goal graph (Gross and Yu)

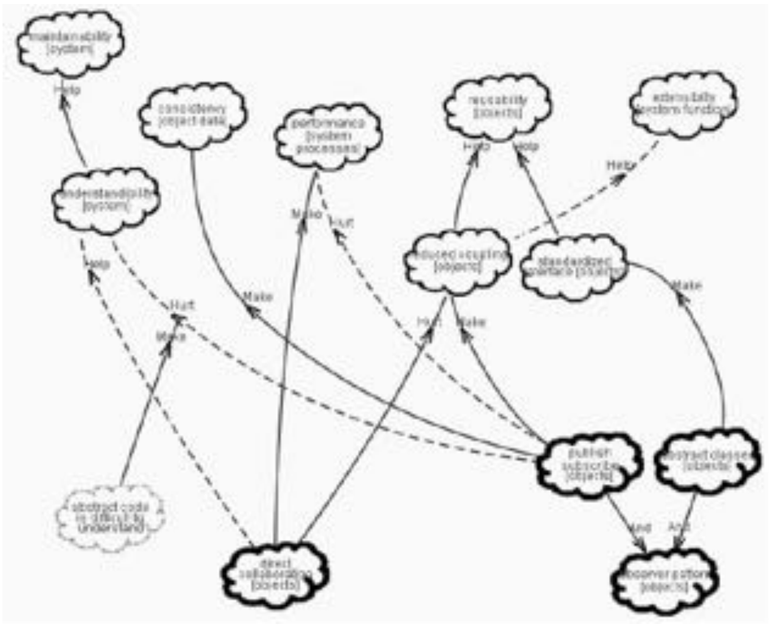

maintainability [system]
Help
consistency [object data]
performance [system processes]
reusability [objects]
extensibility [system functions]
Help
Help
Help
understandability [system]
Make
Hurt
reduced coupling [objects]
standardized interface [objects]
Help
Make
Hurt
Make
Make
Hurt
Make
Make
publish subscribe [objects]
abstract classes [objects]
abstract code is difficult to understand
direct collaboration [objects]
And
And
observer pattern [objects]

Figure 7.2: Reasoning structure of observer pattern as an NFR goal graph (Gross and Yu)

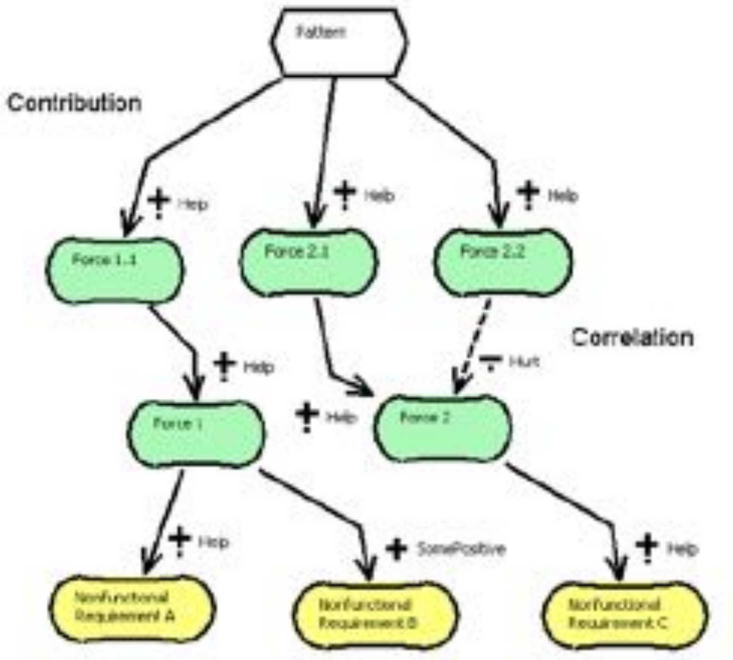


Figure 7.3: GRL based reasoning graph (Mussbacher)

may be significant, but above all constraints concerning hardware and software context become relevant at this stage.

7.1.5 Patterns in Rewrite Rule Supported Design

So far, we have been looking at the structure of the pattern set and the pattern description from the point of view of selecting single patterns in "informal" design processes. When taking on the weighing of alternatives in expression based design, things might look a bit different.

The annotation of basic algorithmic patterns or algorithmic primitives is pointless for the same reasons as it is described in subsection 7.1.2. The primitive expressions then are converted by the application of rewrite rules. As [11] chapter 7 states, the purpose of these rewrite rules is to allow the exploration of the potential solution space a given DSL expression opens up. The key here is (also stated in [11], chapter 7) rewrite rules being picked on the base of reasoning structures which may be matched with elicited requirements and/or refined requirements during early design steps.

Steuwer et.al. [78] have used fixed rewrite rules in their approach, being founded on performance and target platform specifications. In contrast, RePhrase should seek to adopt an annotation mechanism, which allows at least qualitative reasoning on significant NFRs.

7.1.5.1 Influence of Elicited Constraints

So far, we have only considered NFRs in the sense of quality attributes. A key role will belong to what we defined as *constraints* in section 2.3. Rewrite rules or their application must take target hardware and programming model restrictions into account. In their work, Steuwer et.al demonstrate this by rewriting a basic high-level expression to a target specific low-level expression. Where they seek optimal expressions by generating the code and measuring performance, RePhrase should employ a selection strategy based on the annotations applied to DSL expressions and well-known ruleset (e.g. a decision tree). Of course, constraints may be significant not only for generating target centric algorithmic expressions, but may also influence the code generator.

7.1.5.2 Effects on Pattern Set Structure

However, when taking the rewrite rule based route, structuring the full pattern set and the description of individual patterns is insignificant (just for documentation), as the designer of a parallel solution has nothing to do with low level patterns. The only important patterns at top level are the basic algorithmic operations (primitives as called by Steuwer et.al.). At least on this top level and from the designer's perspective, even the distinction between stream parallel patterns and data parallel patterns might be seen as obsolete: Pulling data from a stream rather than from a data source with fixed content is merely a technical attribute of an abstract pattern - typically elicited during the requirements process.

7.1.5.3 Consequences on Requirements Elicitation

Reasoning structures provide information about the characteristics being important for picking a solution. Characteristics modeled in such structures are the ones who shall be considered on requirements elicitation and refining - they provide the base for the forming of specific questions.

For the rewrite rule supported approach, there might be no significant characteristics at top level. But the characteristics influencing the selection and application of rewrite rules and affecting the code generation have to be made explicit to provide information for a requirements engineer or the solution designer.

7.2 On the Concreteness of Requirements

Ideally, the more concrete and detailed a specification is, the less misinterpretations might occur. We have discussed completeness and specification detail levels in principle in section 3.3. Seen from a pragmatic point of view and backed by the findings of our survey (see chapter 8), usually there is a more or less reasonably sized gap from the actual level of detail and completeness to the ideal one. As found, the detail level of initial requirements depends heavily on the project context. There are projects where finding a fundamental (but possibly sequential) solution is the primary goal: Here an algorithm is developed from scratch or by applying known methods in new fields. Such often highly innovative and explorative projects tend to have fewer requirements than projects based on a detailed conception of the problem domain. For the former, stakeholders are able to issue ("business") goals but have no or only a vague vision of the method to achieve these. For the latter, it is likely to have domain experts in the stakeholder group, which usually have a clear conception of the problem, the used methods and the formal relations, yet they do not know how to implement them efficiently. In such projects, there even may be sequential or already parallel - albeit less efficient - implementations (legacy code) in operation yet.

As stated in our discussion of parallel design processes (see section 6.3), knowledge about the domain concepts is an essential basic factor for designing parallel applications. During design, data segmentation is an essential factor. As Steuwer et.al have shown, the ideal segmentation choices go right down to the characteristics of the targeted hardware. Another issue is the parallelization of tasks operating on these data.

However, statements regarding data structuring and segmentation, as well as regarding task or process parallelization (in conjunction of concurrent) data access is not what one will get from a stakeholder during requirements engineering. These are rather issues to be solved during problem analysis and system design.

Nevertheless, RE can support the following phases in a project by asking the right and purposeful questions:

- Clarifications concerning functional aspects or the domain problem cannot be generalized, as the fields of application are too manifold. However, the requirement engineer or developer needs to grow at least a fundamental perception of the problem domain to ease problem analysis - covering the mentioned data structure development and segmentation as well as possible task / function parallelization. A helpful means might be computational patterns as introduced by Keutzer and Mattson, which have been discussed in subsection 6.2.2. These patterns abstract typical computational problems and point up design and implementation strategies. These design strategies refer to other, more specific patterns or pattern combinations.

- Questions concerning NFRs must be aligned with reasoning structures in the pattern description - either in textual form or a (semi-)formal form like pointed out in subsection 7.1.4. As already pointed out, if a functionally equivalent pattern subset cannot be distinguished by a certain NFR, then this particular NFR is obviously irrelevant at this point. Also, if pattern descriptions or subsequent design process steps do not consider a specific NFRs, it is useless for pattern selection and parallel design.

  In Rephrase, the use of annotations is planned for annotating high-level pattern expressions. It is required to make the rationale behind the available annotations and their possible values known to the person in charge of the RE process.

- Eliciting constraints must follow a similar approach. We have discussed the influence of constraints on implementation: Especially targeted HW and SW platforms have a strong influence on eligibility of programming models and technologies or on data segmentation and high-level algorithmic design.

7.3 On Factors Influencing Pattern Selection

7.3.1 From Keywords and Phrases to Solution Strategies

One of our questions is, if there are certain keywords pointing to a specific solution. As already mentioned in section 7.2, this cannot be generalized right away. Of course, specifying concrete performance requirements or asking for performance improvements is a general indicator for parallelization. Even so, parallelization is usually not the only option to speed an existing application or secure the performance goals of a newly developed one.

Besides performance demands, texts containing references to efficiency, scalability, concurrency or synonymous terms are also suggesting parallel solutions.

But providing a full list of general indicators for parallelization is not the point here. The question is to give advice on specific solution strategies for specific given keywords / phrases. Again, we have to point to the efforts made by the Berkley Parallel Labs discussed before (see section 6.3.1.2). As mentioned, their computational patterns describe typical algorithmic problems and describe important aspects concerning parallelization. These pattern descriptions have a more or less detailed section "Known Uses", where the fields of application of a computational pattern are listed and described. The challenge here is then to find a possible methodic / algorithmic approach to a given real-world problem. Solving this question however is beyond the scope of parallel programming - this is rather the task of experts in the respective domain, the problem to solve origins from.

However, as these persons might not be experts in (parallel) programming there is the need to close this gap. A possible approach to support that would require building explicit reasoning structures from implicit knowledge by mapping typical problems or their characteristics to possible methodical / algorithmic approaches, i.e. lifting the "Known Uses" and "Examples" sections of such computational patterns to another higher-level pattern or knowledge management layer (e.g. ontologies). Mapping characteristics, problem descriptions or keywords to other information is typically covered by the field of knowledge management. Finding matchings could be supported by methods of model analysis or natural language processing - depending on the chosen form of requirements documentation.

If there is more than one approach to a domain problem, a comparison section illustrating advantages and disadvantages of alternative approaches can be helpful, too. This might work on the knowledge level as well as down to implementation pattern level. For instance, already Gamma et.al featured a comparison and relationship model of individual patterns in their pattern set.

7.3.2 Pattern Selection

As discussed in section 6.2, pattern descriptions illustrating the applicability of a pattern have to be expressive in a way such that an easy match to a given problem is possible. The way the pattern set is structured and the way its usage in design and development is mandated, is of great significance, too.

In RePhrase, the initial pattern set is divided into stream parallel and data parallel patterns, i.e. the availability of data to be processed is a top selection criterion, which pattern subset shall be used for further analysis and design steps. This fact obviously suggests examining and clarifying this very specific aspect of the given challenge in the first place. The pattern subset is then the base for further requirement clarification and refining. As mentioned before, the designer or requirements engineer needs to consider all the possible influencing factors from the systems future context (discussed in section 3.1), which are referenced in the design process and its underlying artifacts (e.g. code generation rules). This is most definitely the case for HW or SW constraints. However, the Rephrase pattern set does not mention any other selection criteria (on the abstract level as pattern are described in D2.1) than the functional semantics of the pattern. Nevertheless, there must be further steps in the design process selecting implementations - these steps then heavily depend on constraints and possibly on internal NFRs.

Degree of parallelism, as it is mentioned in the "Functional Parameters" section of a RePhrase pattern description is not a requirement rather than the result of the design process, i.e. the steps dealing with data segmentation and task decomposition.

Combining the patterns is a different story and heavily depends on the underlying algorithmic problem - as said computational patterns may be of help here. Of course this has connections to the targeted HW and SW as well (as demonstrated by Steuwer et.al).

A full and complete mapping of influencing factors on pattern selection for specific real-world problems is beyond the scope of this document.

7.3.3 Measuring Influencing Factors

Functional requirements usually cannot be measured in a quantitative sense. If a function works correctly has to be verified by a test case against a specific requirements or an explicitly defined acceptance criterion.

Measuring NFRs is widely covered in various publications. In the ISO quality standard the series 2504x [35] covers the topic of measuring and evaluating quality requirements. Some examples have already been discussed in section 7.1. As discussed performance and efficiency related requirements are dominant factors in parallel programming. There is a broad range of methods to measure the performance of applications, in addition performance may be estimated (e.g. [66]), which can be helpful in pre-selecting or filtering patterns (combinations).

Alternatively, several implementation alternatives may be evaluated by deployment and subsequent profiling such as intended in the RePhrase vision or published by Steuwer [78]. The evaluation can of course take different efficiency metrics into account - depending on the requirements of the respective project (performance time-behavior, memory usage/turnover, ...).

Constraints are not supposed to be measured in a quantitative sense. As we defined in section 2.3, constraints limit the solution space and either are adhered to during design or disregarded.

7.4 On Heuristics for Pattern Selection

According to the definition of Merriam-Webster, a heuristic is *"helping to discover or learn; specif., designating a method of education or of computer programming in which the pupil or machine proceeds along empirical lines, using rules of thumb, to find solutions or answers"*.

In our context, we have on the one hand the need of a general guideline leading from eliciting requirements to designing a solution. This has been discussed above:

- Questions being asked during requirements elicitation and clarification need to be aligned along the reasoning structure of the pattern set and the underlying design process.
- The reasoning structure of a pattern based design process is made up of the functional semantics of the individual patterns and the potential fulfillment of NFRs by a pattern or its implementation. In addition, constraints (especially HW and SW) influence design decisions.
- Abstract, implementation agnostic patterns can hardly be associated with concrete information concerning the support of specific NFRs.
- But NFRs and constraints influence the transition from abstract algorithmic pattern expressions to implementations.

On the other hand the question arises, if it is possible to develop heuristics for specific scenarios, i.e. to select a specific pattern for a concrete problem. As discussed in subsections 7.3.1 and 7.3.2 this may be supported by some knowledge management structure or domain problem patterns, which provide a mapping from typical domain problems to computational patterns describing algorithmic approaches and possible parallelization issues.

- Computational patterns help to bridge the gap between fundamental algorithmic approaches and appropriate parallelization strategies.

- The knowledge layer helps to bridge the gap between real-world, domain-specific problems and possible and algorithmic approaches, which are manifest in the computational patterns.

8 A Survey: RE in Parallel Application Development

By conducting a small number of interviews, we have examined role and extent of requirements engineering in software projects involving parallel programming.

8.1 Purpose and Goals

The goal has been to clarify the impact of stakeholder requirements on finding solutions for parallel problems, especially the influence perceived by NFRs and constraints, by examining implementation projects, which have come up with a parallel solution.

Q: Which are significant requirements gathered in the initial elicitation, i.e. requirements coming from the contractor / client or other important stakeholders ?

Q: Which are significant requirements obtained upon problem analysis and refinement, be it with or without consultation of the client ?

Q: Can we map existing patterns to significant requirements ?

Q: Can we find a set of requirements common to parallel projects?

It doesn't matter, whether the design process has been characterized by a pattern based approach or not. The crucial thing is, if in the final solution of the given problem:

- A pattern or a combination of patterns can be observed,
- And if yes, which requirements and motives led to choosing this solution.

We have been using the taxonomy of requirements as discussed in section 2.3.

8.2 Survey Focus and Target Groups

We have examined software projects, which involved parallel programming problems. The focus has been primarily put on parts of a project which have been solved with a parallel implementation. For the larger projects in our sample, we have not been interested in phenomena affecting the user interface or common business logic rather than the portion which subsequently was implemented using a parallel programming approach.

The survey has been carried out as a series of informal interviews following a general guideline. The purpose of the guideline has been to set the agenda and provide the interviewer with exemplary questions.

Targeted interviewees have been primarily designers/developers, who were assigned the task of solving particular parallel programming problem in a software project. In some cases (especially for the project and RE related part) the project manager had been asked the relevant questions. The following main sections had to be covered by the interview and thus have been featured in the guide.

- Overall project context: The purpose has been to get an idea of the project's purpose and goals as well as a brief overview of the project organization and the general extent and formalization of the initial requirements engineering phase. This section might have been answered by the project manager.

- Particular parallel challenges: The questions in this section have aimed to gather one or more problems that have been due to be solved by parallel design and programming. Of particular interest in this interview phase has been the general characteristics of the requirements the developer has been provided with - for finding out if and how much clarification efforts had to be made.

- Problem solving approach and method: The developer has been asked which approach he/she took to solve the problem including how a potential requirements clarification has been carried out.

  The developer has also been asked to give a brief sketch of his solution, e.g. if it uses well-known patterns.

- Influence of requirements: This section's purpose has been to find out the influencing requirements (functional, non-functional and constraining), which have led to the chosen solution sketched before and to discuss their importance.

- Reflection: A final group of questions has had the intent to ask the developer his personal, critical view on approach, design method and the developed solutions - targeting potential improvements for future, related projects.

8.3 Results

As the survey only has been based on a small number (5) of projects, the findings may not be representative for the whole problem domain of parallel programming. On the other hand, they give an insight on the typical issues and requirements developers are confronted with in small to medium sized projects.

small: project solely focused on the one particular domain problem to be solved by parallelization

medium: other minor aspects such as an user interface or integration into an existing system

There were no large scale parallelization projects, such as massively parallel / cluster based applications.

The project covered real-world problems from different domains including image analysis, optimization of production and engineering processes and analysis of climate data.

No project featured a formal RE process, elicitation was mostly done in one or a few workshops sketching the domain problem.

Project A and to a lesser extent project C had the explicit requirement of parallelizing an existing sequential solution, for all others the focus was primarily the development of algorithms for solving the domain problem. In other words, these other projects did not feature any stakeholder requirements explicitly demanding parallelization. Hence, decisions pro parallelization and used technologies and libraries came during the design phase.

Project A was the only one explicitly using a pattern based approach by exploring the adequacy of a number of alternative solutions. Furthermore, project A together with project C were the only ones citing other NFR influences issued by stakeholders than performance related demands.

- Project A as it was seeking for a high level of maintainability of the resulting source code after applying well-known patterns/skeletons (Maintainability: Code complexity, Refactoring effort). The following in-project evaluation of these aspects had been informal, though.

- Project C was to be deployed in a production context with harsh real-time demands, naming Reliability and Accuracy been prominent NFRs besides performance ("time behavior").

All other projects only featured performance - primarily in the sense of time behavior in the initial set of NFRs. The aspect of resource utilization ("scalability") in the sense of efficiently using available CPU/GPU power played also a role, but not in the initial requirements discussions with stakeholders rather than during designing the parallel solution. In addition, most NFR figures were not specified concretely rather than in an abstract way (e.g. "faster than existing solution") or

Pr	Domain	Patterns used	NFRs by SH	Constraints by SH
A	Production Opt. Local Search Algorithms	Explicit: PoolEvolution, Pipeline	Performance, Scalability, Maintainability (Code Complexity, Refactoring Effort)	None
B	Climate Data Analysis Machine Learning	Workflow-Graph-Interpreter	Performance	None
C	Image Processing	Workflow-Graph-Interpreter	Performance, Reliabiliy (Stability) Functional Suitability (Accuracy)	Win or Linux
D	Electrical Engineering	None	Performance	Customer's Std PC HW Coexistence with Intel MKL based solution
E	Production Opt. Machine Learning Transfer Learning	Map	Performance Resource Util	None

Table 8.1: Pattern usage and stakeholder requirements in projects

a coarse range ("should be 5-10 times faster"). NFRs like portability or security played no role in the examined projects. Unlike security, portability might be of particular interest in parallel computation.

Also noteworthy is the initial absence of significant HW/SW constraints on the part of the stakeholders. Thus, technologically relevant decisions (e.g. use of a specific parallel programming model or library) often have come during design or evolved even later after experimental prototyping. However, especially project C cited a major influence of HW compatible data segmentation on performance.

The detail level of functional requirements varied from very coarse (i.e. resembling production target figures) to highly detailed and specific (domain method-/algorithm fully specified on conceptual level by expert).

There were also no solid hints suggesting specific solution strategies or methods for keywords or phrases being used in the requirements with one reason being that nearly all projects had a rather informal approach to RE.

Table 8.1 shows a brief overview of the usage of patterns in the examined projects and initial (NFR/constraint) requirements issued by stakeholders.

8.4 Consequences

As expected, NFRs (others than performance related) issued by stakeholders were of a very coarse detail level if present at all. Those which were specified wouldn't either be suitable to aid in the selection of functionally-equivalent patterns on an abstract level. Internal quality attributes. Maintainability, Code Complexity,...) cannot be applied to abstract patterns. But of course they may support picking an implementation technology or skeleton.

Looking at stability and reliability requirements bears similar issues: Employing them as distinguishing characteristics in a pattern catalog would require to classify otherwise totally abstract patterns as more or less fulfilling.

For further implications, consult chapter 7.

9 Conclusion

In this document we provided an overview of the goals and tasks of requirements engineering in a software project and delivered a taxonomy of requirements. We furthermore highlighted the two main issues of RE - requirements elicitation and requirements documentation - by elaborating on possible methods, their selection and their conduct.

Subsequently, we related these RE tasks to parallel application design and the RePhrase vision of application development in particular. After a look at traditional, proposed design processes, we also examined a related hi-level expression based approach to parallel programming. In the final section of this work, we elaborated on the effects and implications of RE on the RePhrase vision of a design process and the proposed pattern set, backed by a short survey of projects in the field of parallel programming.

Bibliography

[1] Recommendation z.150, user requirements notation (urn)- language requirements and framework.

[2] Togaf version 9.1, an open group standard, 2011. acessed: 2016.05.18.

[3] IEEE 830 recommended practice for software requirements specifications, Oct 1998.

[4] IEC 25000 software and system engineering–software product quality requirements and evaluation (square)–guide to square, 2005.

[5] Kaos tutorial. http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf, 2007.

[6] ISO/IEC 29148 systems and software engineering – life cycle processes – requirements engineering, Dec 2011.

[7] Systems Modeling Language, Version 1.3. http://www.omg.org/spec/SysML/1.3/, 2012.

[8] ISO/IEC 19510:2013 : Object Management Group Business Process Model and Notation, July 2013.

[9] Scaled agile framework:. http://scaledagileframework.com/, 2015. acessed: 2016.02.02.

[10] Unified Modeling Language (uml). http://www.omg.org/spec/UML/Current, June 2015.

[11] Wp2. rephrase deliverable 2.1: Initial generic patterns report. Technical report, UNIPI, 2015.

[12] Muhammad R Abid, Daniel Amyot, Stéphane S Somé, and Gunter Mussbacher. A uml profile for goal-oriented modeling. In *International SDL Forum*, pages 133–148. Springer, 2009.

[13] S. Adam, J. Doerr, M. Eisenbarth, and A. Gross. Using task-oriented requirements engineering in different domains 150; experiences with application in research and industry. In *2009 17th IEEE International Requirements Engineering Conference*, pages 267–272, Aug 2009.

[14] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), 2001.

[15] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core.(a fastflow short tutorial). 2011. acessed: 2016.05.18.

[16] D. Amyot, R. Rashidi-Tabrizi, G. Mussbacher, J. Kealey, E. Tremblay, and J. Horkoff. *6th International i* Workshop (iStar 2013)*, chapter Improved GRL Modeling and Analysis with jUCMNav 5. 2013.

[17] Daniel Amyot and Gunter Mussbacher. User requirements notation: the first ten years, the next ten years. *Journal of software*, 6(5):747–768, 2011.

[18] Aybüke Aurum and Claes Wohlin, editors. *Engineering and managing software requirements*. Springer, Berlin, 2005.

[19] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2012.

[20] Kent Beck and Martin Fowler. *Planning extreme programming*. Addison-Wesley Professional, 2001.

[21] Manfred Broy and Andreas Rausch. Das neue v-modell® xt. *Informatik-Spektrum*, 28(3):220–229, 2005.

[22] Frank Buschmann, Kelvin Henney, and Douglas Schimdt. *Pattern-oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John wiley & sons, 2007.

[23] Sonia Campa and Massimo Torquati. Final pattern definition report d2.5. Technical report, UNIPI, 2013.

[24] Lan Cao and Balasubramaniam Ramesh. Agile requirements engineering practices: An empirical study. *Software, IEEE*, 25(1):60–67, 2008.

[25] Christine Choppy, Denis Hatebur, and Maritta Heisel. Systematic architectural design based on problem patterns. In Paris Avgeriou, John Grundy, Jon G. Hall, Patricia Lago, and Ivan Mistrík, editors, *Relating Software Requirements and Architectures*, pages 133–159. Springer Berlin Heidelberg, 2011.

[26] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer, 2009.

[27] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*, chapter The NFR Framework in Action, pages 15–45. Springer US, Boston, MA, 2000.

[28] Lawrence Chung, Sam Supakkul, Nary Subramanian, JoséLuis Garrido, Manuel Noguera, MariaV. Hurtado, MaríaLuisa Rodríguez, and Kawtar Benghazi. Goal-oriented software architecting. In Paris Avgeriou, John Grundy, Jon G. Hall, Patricia Lago, and Ivan Mistrík, editors, *Relating Software Requirements and Architectures*, pages 91–109. Springer Berlin Heidelberg, 2011.

[29] David Cohen, Mikael Lindvall, and Patricia Costa. An introduction to agile methods. *Advances in computers*, 62:1–66, 2004.

[30] Mike Cohn. Advantages of user stories for requirements. http://www.mountaingoatsoftware.com/articles/advantages-of-user-stories-for-requirements, 2004. acessed: 2016.04.25.

[31] Robert Darimont, Emmanuelle Delor, Philippe Massonet, and Axel van Lamsweerde. Grail/kaos: an environment for goal-driven requirements engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 612–613. ACM, 1997.

[32] D. de Almeida Ferreira and A. R. da Silva. Rsl-pl: A linguistic pattern language for documenting software requirements. In *Requirements Patterns (RePa), 2013 IEEE Third International Workshop on*, pages 17–24, July 2013.

[33] A Duran, B Bernardez, M Toro, R Corchuelo, A Ruiz, and J Perez. Expressing customer requirements using natural language requirements templates and patterns. In *IMACS/IEEE CSCC 99 Proceedings*, 1999.

[34] Armin Eberlein and JCSP Leite. Agile requirements definition: A view from requirements engineering. In *Proceedings of the International Workshop on Time-Constrained Requirements Engineering (TCRE02)*, pages 4–8, 2002.

[35] Kazuhiro Esaki, Motoei Azuma, and Toshihiro Komiyama. Introduction of quality requirement and evaluation based on iso/iec square series of standard. In *Trustworthy Computing and Services*, pages 94–101. Springer, 2012.

[36] Sergio Espana, Nelly Condori-Fernandez, Arturo Gonzalez, and Óscar Pastor. Evaluating the completeness and granularity of functional requirements specifications: a controlled experiment. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 161–170. IEEE, 2009.

[37] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[38] Xavier Franch. Software requirements patterns: A state of the art and the practice. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 943–944, Piscataway, NJ, USA, 2015. IEEE Press.

[39] Xavier Franch, Carme Quer, Samuel Renault, Cindy Guerlain, and Cristina Palomares. Constructing and using software requirement patterns. In *Managing requirements knowledge*, pages 95–116. Springer, 2013.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[41] Tom Gilb. *Competitive engineering: a handbook for systems engineering, requirements engineering, and software engineering using Planguage*. Butterworth-Heinemann, 2005.

[42] Tom Gilb. Rich requirement specs: The use of planguage to clarify requirements. www.gilb.com/dl44, 2006. acessed: 2016.05.08.

[43] Martin Glinz. On non-functional requirements. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 21–26. IEEE, 2007.

[44] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[45] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[46] Daniel Gross and Eric Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, 2001.

[47] Gerhard Keller, Markus Nüttgens, and August-Wilhelm Scheer. Semantische prozeßmodellierung auf der basis ereignisgesteuerter prozeßketten (epk). *Veröffentlichungen des Instituts für Wirtschaftsinformatik (UdS Saarbrücken)*, 89:11, 1992.

[48] Kurt Keutzer, Berna L Massingill, Timothy G Mattson, and Beverly A Sanders. A design pattern language for engineering (parallel) software: merging the plpp and opl projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, page 9. ACM, 2010.

[49] Kurt Keutzer and Tim Mattson. A design pattern language for engineering (parallel) software. *Intel Technology Journal*, 13(4), 2009.

[50] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Éric Rutten. Uml2 profile for modeling controlled data parallel applications. In *Advances in Design and Specification Languages for Embedded Systems*, pages 301–317. Springer, 2007.

[51] Alexej Lapouchnian. Goal-oriented requirements engineering: An overview of the current research. Technical report, University of Toronto, 2005.

[52] Yunsup Lee. Pipeline pattern. http://parlab.eecs.berkeley.edu/wiki/_media/patterns/pipeline-v1.pdf, 03 2009. acessed: 2016.05.18.

[53] Dean Leffingwell. *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison-Wesley Professional, 2011.

[54] Ricardo J Machado, João M Fernandes, Paula Monteiro, and Helena Rodrigues. Refinement of software architectures by recursive model transformations. In *International Conference on Product Focused Software Process Improvement*, pages 422–428. Springer, 2006.

[55] Neil Maiden. Improve your requirements: Quantify them. *Software, IEEE*, 23(6):68–69, 2006.

[56] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[57] Steve McConnell. *Rapid development: taming wild software schedules*. Pearson Education, 1996.

[58] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[59] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[60] Gunter Mussbacher, Michael Weiss, and Daniel Amyot. Formalizing architectural patterns with the goal-oriented requirements language. In *Fifth Nordic Conference on Pattern Languages of Programs*, page 13, 2007.

[61] Jorge Luis Ortega-Arjona. *Architectural patterns for Parallel Programming: models for performance estimation.* PhD thesis, University of London, 2007.

[62] Jorge Luis Ortega-Arjona. *Patterns for parallel software design*, volume 21. John Wiley & Sons, 2010.

[63] Barbara Paech and Daniel Kerkow. Non-functional requirements engineering-quality is essential. In *10th International Workshop on Requirments Engineering Foundation for Software Quality*, 2004.

[64] Yasset Pérez-Riverol and Roberto Vera Alvarez. A uml-based approach to design parallel and distributed applications. *CoRR*, abs/1311.7011, 2013.

[65] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2009.

[66] Sabri Pllana, Siegfried Benkner, Fatos Xhafa, and Leonard Barolli. A novel approach for hybrid performance modelling and prediction of large-scale computing systems. *International journal of grid and utility computing*, 1(4):316–327, 2009.

[67] Sabri Pllana and Thomas Fahringer. Uml based modeling of performance oriented parallel and distributed applications. In *Simulation Conference, 2002. Proceedings of the Winter*, volume 1, pages 497–505. IEEE, 2002.

[68] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[69] Klaus Pohl. The three dimensions of requirements engineering. In *Seminal Contributions to Information Systems Engineering*, pages 63–80. Springer, 2013.

[70] James Robertson and Suzanne Robertson. Volere requirements specification template. http://www.volere.co.uk/template.htm, 2016. acessed: 2016.04.28.

[71] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012.

[72] Kenneth S Rubin. *Essential Scrum: A practical guide to the most popular Agile process*. Addison-Wesley, 2012.

[73] Chris Rupp and The Sophists. *Requirements-Engineering und-Management: Aus der Praxis von klassisch bis agil*. Carl Hanser Verlag GmbH Co KG, 2014.

[74] Christine Rupp. Requirements templates-the blueprint of your requirement. *Sophist Group, Nürnberg*, 2014.

[75] Thomas L Saaty. Decision making with the analytic hierarchy process. *International journal of services sciences*, 1(1):83–98, 2008.

[76] Ian Sommerville. Formal specifications. http://iansommerville.com/software-engineering-book/files/2014/06/Ch_27_Formal_spec.pdf, 2009. accessed: 2016.02.02.

[77] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997.

[78] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217, New York, NY, USA, 2015. ACM.

[79] S Supakkul and L Chung. Goal achievement evaluation in the nfr framework. http://utdallas.edu/~supakkul/NFR-modeling/label%20evaluation%20and%20world%20assumptions/label-propagation.htm, 2010. acessed: 2016.05.03.

[80] Sam Suppakul. Re-tools: A multi-notational requirements modeling toolkit. http://www.utdallas.edu/~supakkul/tools/RE-Tools/, 2015. accessed: 2016.03.18.

[81] Axel Van Lamsweerde. Building formal requirements models for reliable software. In *Reliable SoftwareTechnologies - Ada-Europe 2001*, pages 1–20. Springer, 2001.

[82] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[83] Axel Van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, pages 25–43. Springer, 2003.

[84] Axel Van Lamsweerde and Emmanuel Letier. From object orientation to goal orientation: A paradigm shift for requirements engineering. In *Radical Innovations of Software and Systems Engineering in the Future*, pages 325–340. Springer, 2004.

[85] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[86] A. B. Younes and L. J. B. Ayed. Using uml activity diagrams and event b for distributed and parallel applications. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 163–170, July 2007.

[87] Anton Yrjönen and Janne Merilinna. Extending the nfr framework with measurable nonfunctional requirements. In *NFPinDSML@ MoDELS*, 2009.

[88] Eric SK Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium on*, pages 226–235. IEEE, 1997.

[89] Didar Zowghi and Chad Coulin. Requirements elicitation: A survey of techniques, approaches, and tools. In *Engineering and managing software requirements*, pages 19–46. Springer, 2005.

[90] Didar Zowghi and Vincenzo Gervasi. The three cs of requirements: consistency, completeness, and correctness. In *International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany: Essener Informatik Beitiage*, pages 155–164, 2002.