Requirements Engineering

Amel Bennaceur, Thein Than Tun, Yijun Yu, and Bashar Nuseibeh

Abstract Requirements engineering (RE) aims to ensure that systems meet the needs of their stakeholders including users, sponsors, and customers. Often considered as one of the earliest activities in software engineering, it has developed into a set of activities that touch almost every step of the software development process. In this chapter, we reflect on how the need for RE was first recognised and how its foundational concepts were developed. We present the seminal papers on four main activities of the RE process, namely, (1) elicitation, (2) modelling and analysis, (3) assurance, and (4) management and evolution. We also discuss some current research challenges in the area, including security requirements engineering as well as RE for mobile and ubiquitous computing. Finally, we identify some open challenges and research gaps that require further exploration.

1 Introduction

This chapter presents the foundational concepts of requirements engineering (RE) and describes the evolution of RE research and practice. RE has been the subject of several popular books [47, 102, 56, 90, 98, 107] and surveys [20, 82]; this chapter clarifies the nature and evolution of RE research and practice, gives a guided introduction to the field, and provides relevant references for further exploration of the area.

All authors have contributed equally to this chapter.

A. Bennaceur · T. T. Tun · Y. Yu
The Open University, Milton Keynes, UK
e-mail: Amel.Bennaceur@open.ac.uk; t.t.tun@open.ac.uk; yijun.yu@open.ac.uk

B. Nuseibeh
The Open University, Milton Keynes, UK

Lero The Irish Software Research Centre, Limerick, Ireland
e-mail: bashar.nuseibeh@lero.ie; B.Nuseibeh@open.ac.uk

S. Cha et al. (eds.), *Handbook of Software Engineering*,
https://doi.org/10.1007/978-3-030-00262-6_2

The target readers are students interested in the main theoretical and practical approaches in the area, professionals looking for practical techniques to apply, and researchers seeking new challenges to investigate. But first, what is RE?

> Requirements engineering is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families.—Zave [111]

Zave's definition emphasises that a new software system is introduced to solve a real-world problem and that a good understanding of the problem and the associated context is at the heart of RE. Therefore, it is important not only to define the goals of the software system but also to specify its behaviour and to understand the constraints and the environment in which this software system will operate. The definition also highlights the need to consider change, which is inherent in any real-world situation. Finally, the definition suggests that RE aims to capture and distil the experience of software development across a wide range of applications and projects.

Although Zave's definition identifies some of the key challenges in RE, the nature of RE itself has been changing. First, although much of the focus in this chapter is given to software engineering, which is the subject of the book, RE is not specific to software alone but to socio-technical systems in general, of which software is only a part. Software today permeates every aspect of our lives, and therefore, one must not only consider the technical but also the physical, economical, and social aspects. Second, an important concept in RE is stakeholders, i.e. individuals or organisations who stand a gain or loss from the success or failure of the system to be constructed [82]. Stakeholders play an important role in eliciting requirements as well as in validating them.

The chapter covers both the foundations and the open challenges of RE. When you have read the chapter, you will:

- Appreciate the importance of RE and its role within the software engineering process;
- Recognise the techniques for eliciting, modelling, documenting, validating, and managing requirements for software systems;
- Understand the challenges and open research issues in RE.

The chapter is structured as follows. Section 2 introduces the fundamental concepts of RE including the need to make explicit the relationship between requirements, specifications, and environment properties, the quality properties of requirements, and the main activities with the RE process. Section 3 presents seminal work in requirements elicitation, modelling, assurance, and management. It also discusses the RE techniques that address cross-cutting properties, such as security, that involves a holistic approach across the RE process. Section 4 examines the challenges and research gaps that require further exploration. Section 5 concludes the chapter.

2 Concepts and Principles

In the early days of software engineering, approximately from the 1960s up to around the 1980s, many software systems used by organisations were largely, if not completely, constructed in-house, by the organisations themselves. There were serious problems with these software projects: software systems were often not delivered on time and on budget. More seriously, their users did not necessarily like to use the constructed software systems. Brooks [19] assessed the role of requirements engineering in such projects as follows:

> The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Although it was fashionable to argue that users do not really know what their requirements are (and so it was difficult for software engineers to construct systems users would want to use), it was also true that software engineers did not really know what they mean when they say requirements. Much confusion abounds around the term requirements. In fact, Brooks himself talked about "detailed technical requirements" and "product requirements" in the same paper without really explaining what they mean. Elsewhere, people were also using terms like "system requirements", "software requirements", "user requirements", and so on. Obviously people realised that when they say requirements to each other, they might be talking about very different things, hence the many modifiers. The need to give precise meanings to the terms was quite urgent. In the following, Sect. 2.1 starts by introducing Jackson and Zave's framework for requirements engineering [112], which makes explicit the relationship between requirements, specifications, and environment properties. Section 2.2 defines the desirable attributes of requirements. Finally, Sect. 2.3 introduces the main activities within the RE process.

2.1 Fundamentals: The World and the Machine

Zave and Jackson [112] propose a set of criteria that can be used to define requirements and differentiate them from other artefacts in software engineering. Their work is closely related to the "four-variable model" proposed by Parnas and Madey [85], which defines the kinds of content that documents produced in the software engineering process should contain.

Central to the proposal of Zave and Jackson is the distinction between the machine and the world. Essentially the machine is the software system. The portion of the real world where the machine is to be deployed and used is called the *environment*. Hence, scoping the problem by defining the boundary of the environment is paramount. The machine and the environment are described in terms of *phenomena*, such as events, objects, states, and variables. Some phenomena

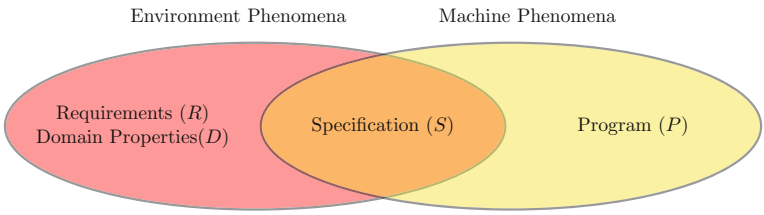


Fig. 1 World, machine, and specification phenomena

belong to the world, and some phenomena belong to the machine. Since the world and the machine are connected, their phenomena overlap (Fig. 1).

Typically the machine observes some phenomena in the environment, such as events and variables, and the machine can control parts of the environment by means of initiating some events. This set of machine observable and machine controllable phenomena sits at the intersection between the machine and the world, and they are called specification phenomena (S). There are also parts of the environment that the machine can neither control nor observe directly. Indicative statements that describe the environment in the absence of the machine or regardless of the machine are often called *assumptions* or *domain properties* (D). Optative statements expressing some desired properties of the environment that are to be brought about by constructing the machine are called *requirements* (R). Crucially, requirements statements are never about the properties of the machine itself. In fact, Zave and Jackson assert that all statements made during RE should be about the environment. That means that during RE, the engineer has to describe the environment without the machine, and the environment with the machine. From these two descriptions, it is possible to derive the specification of the machine systematically [50].

Accordingly, Zave and Jackson suggest that there are three main kinds of artefacts that engineers would produce during the RE process:

1. statements about the *domain* describing properties that are true regardless of the presence or actions of the machine,
2. statements about *requirements*, describing properties that the users want to be true of the world in the presence of the machine,
3. statements about the *specification* describing what the machine needs to do in order to achieve the requirements.

These statements can be written in natural language, formal logic, semi-formal languages, or indeed in some combination of them, and Zave and Jackson are not prescriptive about that. What is important is their relationship, which is as follows:

The specification (S), together with the properties of the domain (D),
should satisfy the requirements (R): $S, D \vdash R$.

Returning to the issue of confusion about what requirements mean, is the term any clearer in the light of such research work? It seems so. For example, a statement like "The program must be written in C#" is not a requirement because this is not a property of the environment; it is rather an implementation decision, that

unnecessary constrains potential system specifications. Statements like "A library reader is allowed borrow up to 10 different books at a time" are a requirement, but "Readers always return their loans on time" is an assumed property of the environment, i.e. a domain property.

Hence, RE is grounded in the real world; it involves understanding the environment in which the system-to-be will operate and defining detailed, consistent specification of the software system-to-be. This process is incremental and iterative as we will see in Sect. 2.3. Zave and Jackson [112] specify five criteria for this process to complete:

1. Each requirements R has been validated with the stakeholders.
2. Each domain property D has also been validated with the stakeholders.
3. The requirements specification S does not constrain the environment or refer to the future.
4. There exists a proof that an implementation of S satisfies R in environment W, i.e. $S, W \vdash R$ holds.
5. S and D are consistent, i.e. $S, D \nvdash false$.

The ideas proposed by Zave and Jackson are quite conceptual: they help clarify our thinking about requirements and how to work with the requirements productively during software development. There are many commercial and non-commercial tools that use some of the ideas: REVEAL [5] for example helps address the importance of understanding the operational context through domain analysis.

2.2 Qualities

Requirements errors and omissions are relatively more costly than those introduced in later stages of development [13]. Therefore, it is important to recognise some of the key qualities of requirements, which include:

- *Measurability.* If a software solution is proposed, one must be able to demonstrate that it meets its requirements. For example, a statement such as "response time small enough" is not measurable, and any solution cannot be proven to satisfy it or not. A measurable requirement would be "response time smaller than 2 s". Agreement on such measures will then reduce potential dispute with stakeholders. Having said that, requirements such as response time are easier to quantify than others such as requirements for security.
- *Completeness.* Requirements must define all properties and constraints of the system-to-be. In practice, completeness is achieved by complying with some guidelines for defining the requirements statements such as ensuring that there are no missing references, definitions, or functions [13].

- *Correctness* (sometimes also referred to as *adequacy* or *validity*). The stakeholders and requirements engineer have the same understanding of what is meant by the requirements. Practically, correctness often involves compliance with other business documents, policies, and laws.
- *Consistency.* As the RE process often involves multiple stakeholders holding different views of the problems to be addressed, they might be contradictory at the early stages. Through negotiation [15] and prioritisation [11], the conflicts between these requirements can be solved, and an agreement may be reached.
- *Unambiguity.* The terms used within the requirements statements must mean the same both to those who created it and those who use it. Guidelines to ensure unambiguity include using a single term consistently and a glossary to define any term with multiple meanings [44].
- *Pertinence.* Clearly defining the scope of the problem to solve is fundamental in RE [90]. Requirements must be relevant to the needs of stakeholders without unnecessarily restricting the developer.
- *Feasibility.* The requirements should be specified in a way that they can be implemented using the available resources such as budget and schedule [13].
- *Traceability.* Traceability is about relating software artefacts. When requirements change and evolve over time, traceability can help identify the impact on other software artefacts and assess how the change should be propagated [23].

These qualities are not necessarily exhaustive. Other qualities that are often highlighted include *comprehensibility*, *good structuring*, and *modifiability* [102]. Considerations of these quality factors guide the RE process. Indeed, since RE is grounded in the physical world, it must progress from acquiring some understanding of organisational and physical settings to resolving potential conflicting views about what the software system-to-be is supposed to do and to defining a specification of the software system-to-be that satisfies the above properties. This process is often iterative and incremental with many activities.

Robertson and Robertson [90] define the Volere template as a structure for the effective specification of requirements and propose a method called quality gateway to focus on atomic requirements and ensure "each requirement is as close to perfect as it can be". IEEE 830-1998: IEEE Recommended Practice for Software Requirements Specifications [44] provides some guidance and recommendations for specifying software requirements and details a template for organising the different kinds of requirements information for a software product in order to produce a software requirements specification (SRS).

2.3 Processes

Requirements often permeate throughout many parts of systems development (see Fig. 2). At the early stages of system development, requirements have a significant influence on system feasibility. During system design, requirements are used to

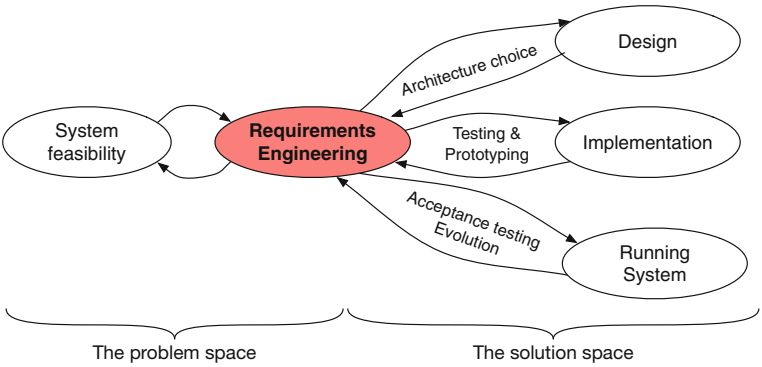


Fig. 2 RE and software development activities

inform decision-making about different design alternatives. During systems implementation, requirements are used to enable system function and subsystem testing. Once the system has been deployed, requirements are used to drive acceptance tests to check whether the final system does what the stakeholders originally wanted. In addition, requirements are reviewed and updated during the software development process as additional knowledge is acquired and stakeholders' needs are better understood. Each step of the development process may lead the definition of additional requirements through a better understanding of the domain and associated constraints. Nuseibeh [81] emphasise the need to consider requirements, design, and architecture concurrently and highlight that this is often the process adopted by developers.

While the definition of the requirements helps delimit the solution space, the requirements problem space is less constrained, making it difficult to define the environment boundary, negotiate resolution of conflicts, and set acceptance criteria [20]. Therefore, several guidelines are given to define and regulate the RE processes in order to build adequate requirements [90]. Figure 3 summarises the main activities of RE:

❶ **Elicitation** Requirements elicitation aims to discover the needs of stakeholders as well as understand the context in which the system-to-be will operate. It may also explore alternative ways in which the new system could be specified. A number of techniques can be used including: (1) traditional data gathering techniques (e.g. interviews, questionnaires, surveys, analysis of existing documentation), (2) collaborative techniques (e.g. brainstorming, RAD/JAD workshops, prototyping), (3) cognitive techniques (e.g. protocol analysis, card sorting, laddering), (4) contextual techniques (e.g. ethnographic techniques, discourse analysis), and (5) creativity techniques (e.g. creativity workshops, facilitated analogical reasoning) [113]. Section 3.1 is dedicated to elicitation.

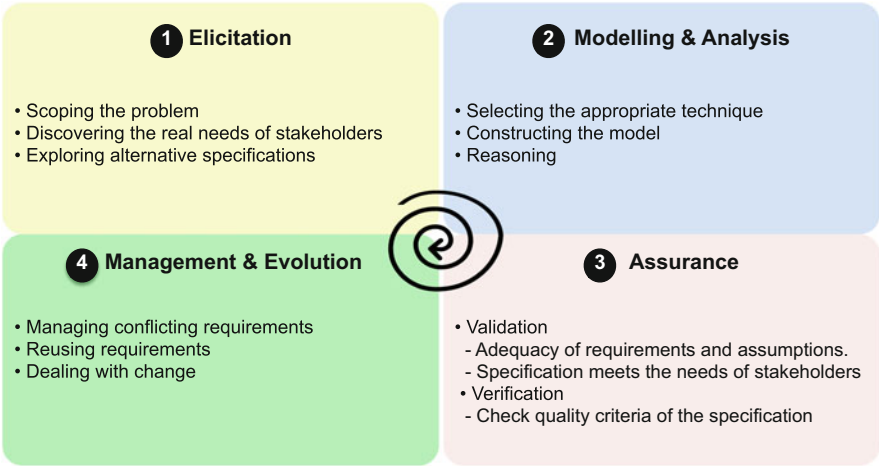


Fig. 3 Main activities of RE

❷ **Modelling and Analysis** The results of the elicitation activity often need be described precisely and in a way accessible by domain experts, developers, and other stakeholders. A wide range of techniques and notations can be used for requirements specification and documentation, ranging from informal to semi-formal to formal methods. The choice of the appropriate method often depends on the kind of analysis or reasoning that needs to be performed. Section 3.2 is dedicated to modelling and analysis.

❸ **Assurance** Requirements quality assurance seeks to identify, report, analyse, and fix defects in requirements. It involves both validation and verification. Validation aims to check the adequacy of the specified and modelled requirements and domain assumptions with the actual expectations of stakeholders. Verification covers a wide range of checks including quality criteria of the specified and modelled requirements (e.g. consistency). Section 3.3 is dedicated to assurance.

❹ **Management and Evolution** Requirements management is an umbrella term for the handling of changing requirements, reviewing and negotiating the requirements and their priorities, as well as maintaining traceability between requirements and other software artefacts. Section 3.4 discusses some of the issues of managing change and the requirement-driven techniques to address them.

These RE activities happen rarely in sequence since the requirements can rarely be fully gathered upfront and changes occur continuously. Instead, the process is iterative and incremental and can be viewed, like the software development process, as a spiral model [14]. In the following section, we present the seminal work associated with each of the activities in the RE process.

3 Organised Tour: Genealogy and Seminal Works

RE is multidisciplinary in nature. As a result, different techniques for elicitation, modelling, assurance, and management often co-exist and influence one another without a clear chronological order. This section presents the key techniques and approaches within each of the RE activities shown in Fig. 4. We begin with requirements elicitation techniques and categorise them. Next, we describe techniques for modelling and analysing requirements, followed by the techniques for validating and verifying requirements. We then explore techniques for dealing with change and uncertainty in requirements and their context. Finally, we discuss properties such as security and dependability that require a holistic approach that cuts across the RE activities.

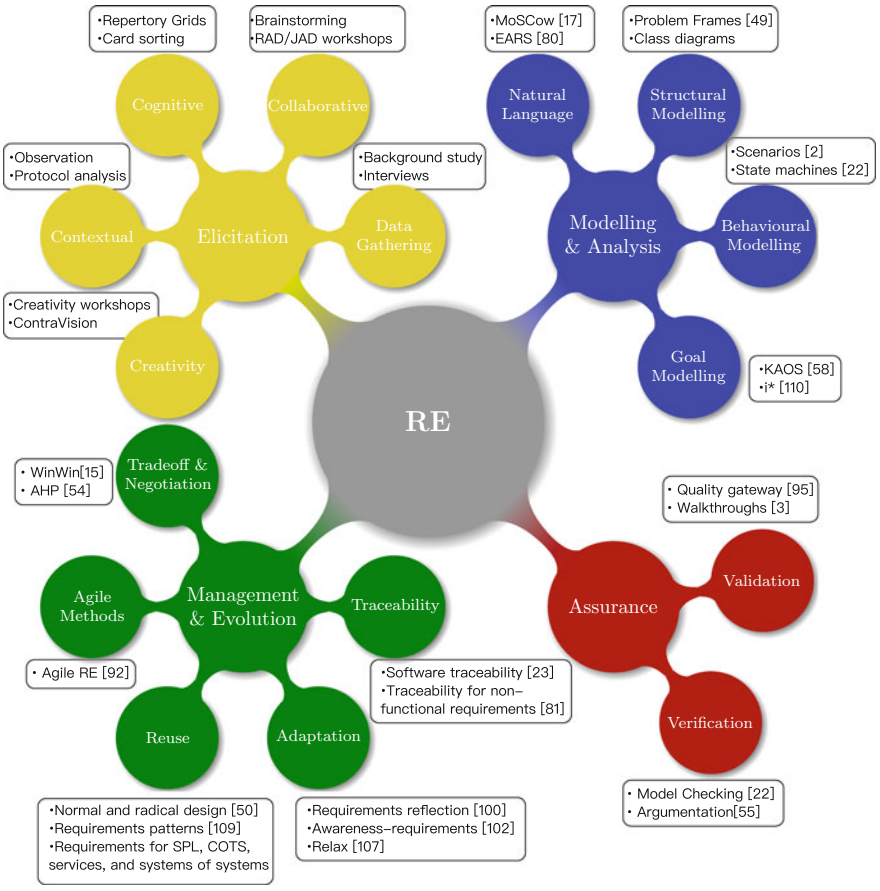


Fig. 4 Classification of key RE techniques

3.1 Elicitation

Requirements elicitation aims at acquiring knowledge and understanding about the system-as-is, the system-to-be, and the environment in which it will operate. First, requirements engineers must scope the problem by understanding the context in which the system will operate and identifying the problems and opportunities of the new system. Second, they also need to identify the stakeholders and determine their needs. These needs may be identified through interaction with the stakeholders, who know what they want from the system and are able to articulate and express those needs. There are also needs that the stakeholders do not realise are possible but that can be formulated through invention and creative thinking [69]. Finally, elicitation also aims to explore alternative specifications for the system-to-be in order to address those needs.

However, eliciting requirements is challenging for a number of reasons. First, in order to understand the environment, requirements engineers must access and collect information that is often distributed across many locations and consult a large number of people and documents. Communication with stakeholders is paramount, especially for capturing tacit knowledge and hidden needs and for uncovering biases. These stakeholders may have diverging interests and perceptions and can provide conflicting and inconsistent information. Key stakeholders may also be not easy to contact or interested in contributing. Finally, changing socio-technical context may lead to reviewing priorities, identifying new stakeholders, or revising requirements and assumptions.

A range of elicitation techniques have been proposed to address some of those challenges. An exhaustive survey of those techniques is beyond the scope of this chapter. In the following, we present the main categories and some representative techniques as summarised in Table 1. We refer the interested reader to the survey by Zowghi and Coulin [113] for further details.

3.1.1 Data Gathering

This category includes traditional techniques for collecting data by analysing existing documentation and questioning relevant stakeholders.

Background Study Collecting, examining, and synthesising existing and related information about the system-as-is and its context is a useful way to gather early requirements as well as gain an understanding of the environment in which the system will operate. This information can originate from studying documents about business plans, organisation strategy, and policy manuals. It can also come from surveys, books, and reports on similar systems. It can also derive from the defect and complaint reports or change requests. Background study enables requirements engineers to build up the terminology and define the objective and policies to be considered. It can also help them identify opportunities for reusing existing specifications. However, it may require going through a considerable amount of

Table 1 Summary of requirements elicitation techniques

Category	Main idea	Example techniques
Data gathering	Collecting data by analysing existing documentation and questioning stakeholders	• Background study • Interviews
Collaborative	Leveraging group dynamics to foster agreements	• Brainstorming • RAD/JAD workshops
Cognitive	Acquiring domain knowledge by asking stakeholders to think about, characterise, and categorise domain concepts	• Repertory grids • Card sorting
Contextual	Observing stakeholders' and users' performing tasks in context	• Observation • Protocol analysis
Creativity	Inventing requirements	• Creativity workshops • ContraVision

documents, processing irrelevant details, and identifying inaccurate or outdated data.

Interviews Interviews are often considered as one of the most traditional and commonly used elicitation techniques. It typically involves requirements engineers selecting specific stakeholders, asking them questions about particular topics, and recording their answers. Analysts then prepare a report from the transcripts and validate or refine it with the stakeholders. In a structured interview, analysts would prepare the list of questions beforehand. Through direct interaction with stakeholders, interviews allow requirements engineers to collect valuable data even though the data obtained might be hard to integrate and analyse. In addition, the quality of interviews greatly depends on the interpersonal skills of the analysts involved.

3.1.2 Collaborative

This category of elicitation techniques takes advantage of the collective ability of a group to perceive, judge, and invent requirements either in an unstructured manner such as with brainstorming or in a structured manner such as in Joint Application Development (JAD) workshops.

Brainstorming Brainstorming involves asking a group of stakeholders to generate as many ideas as possible to improve a task or address a recognised problem, then to jointly evaluate and choose some of these ideas according to agreed criteria. The free and informal style of interaction in brainstorming sessions may lead to generate many, and sometimes inventive, properties of the system-to-be as well as

tacit knowledge. The challenge however is to determine the right group composition to avoid unnecessary conflicts, bias, and miscommunication.

Joint Application Development (JAD) Workshops Similar to brainstorming, JAD involves a group of stakeholders discussing both the problems to be solved and alternative solutions. However, JAD often involves specific roles and viewpoints for participants, and discussions are supported by a facilitator. The well-structured nature of these facilitated interactions and the collaboration between involved parties can lead to rapid decision-making and conflict solving.

3.1.3 Cognitive

Techniques within this category aim to acquire domain knowledge by asking stakeholders to think about, characterise, and categorise domain concepts.

Card Sorting Stakeholders are asked to sort into groups a set of cards, each of which has the name of some domain concept written or depicted on it. Stakeholders then identify the criteria used for sorting the cards. While the knowledge obtained is at a high level of abstraction, it may reveal latent relationships between domain concepts.

Repertory Grids Stakeholders are given a set of domain concepts for which they are asked to assign attributes, which results in a concept $\times$ attribute matrix. Due to their finer-grained abstraction, repertory grids are typically used to elicit expert knowledge and to compare and detect inconsistencies or conflict in this knowledge [80].

3.1.4 Contextual

Techniques within this category aim to analyse stakeholders in context to capture knowledge about the environment and ensure that the system-to-be is fit for use in that environment.

Observation Observation is an ethnographic technique whereby the requirements engineers observe actual practices and processes in the domain without interference. While observation can reveal tacit knowledge, it requires significant skills and effort to gain access to the organisation without disrupting the normal behaviour of participants (stakeholders or actors) as well as to interpret and understand these processes.

Protocol Analysis In this technique, requirements engineers observe participants undertaking some tasks and explaining them out loud. This technique may reveal specific information and rational for the processes within the system-as-is. However, it does not necessarily reveal enough information about the system-to-be.

3.1.5 Creativity

Most of the aforementioned elicitation techniques focus on distilling information about the environment and existing needs of the stakeholders. Creativity elicitation techniques emphasise the role of requirements engineers to bring about innovative change in a system, which would give a competitive advantage. To do so, creativity workshops introduce creativity techniques within a collaborative environment. Another technique consists in using futuristic videos, or other narrative forms, in order to engage stakeholders in exploring unfamiliar or controversial systems.

Creativity Workshops Creativity workshops [68] encourage a fun atmosphere so that the participants are relaxed and prepared to generate and voice novel ideas. Several techniques are used to stimulate creative thinking. For example, facilitated analogical reasoning can be used to import ideas from one problem space to another. Domain constraints can be removed in order to release the cognitive blocks and create new opportunities for exploring innovative ideas. Building on combinatorial creativity, requirements engineers can swap requirements between groups of participants so as to generate new properties of the systems-to-be by combining proposed ideas from each group in novel ways.

ContraVision ContraVision [70] uses two identical scenarios that highlight the positive and negative aspects of the same situation. These scenarios use a variety of verbal, musical, and visual codes as a powerful tool to trigger intellectual and emotional responses from the stakeholders. The goal of ContraVision is to elicit a wide spectrum of stakeholders' reactions to potentially controversial or futuristic technologies by providing alternative representations of the same situation.

3.1.6 Choosing and Combining Elicitation Techniques

Requirements elicitation remains a difficult challenge. The problem is not a lack of elicitation techniques, since a wide range already exists, each with its strengths and weakness. But taken in isolation, none is sufficient to capture complete requirements. The challenge is then to select and plan a systematic sequence of appropriate techniques to apply. ACRE framework [67] supports analysts for selecting elicitation techniques according to their specified features and building reusable combinations of techniques. Empirical studies have also been conducted to identify most effective elicitation techniques, best practices, and systematic ways for combining and applying them [27, 30].

3.2 Modelling and Analysis

While requirements elicitation aims to identify the requirements, domain properties, and the associated specifications, modelling aims to reason about the interplay and

Table 2 Summary of requirements modelling techniques

Category	Main idea	Example techniques
Natural language	Guidelines and templates to write requirements statements	• MoSCoW • EARS
Structural	Delimiting the problem world by defining its components and their interconnections	• Problem frames • Class diagrams
Behavioural	Interactions between actors and the system-to-be	• Scenarios • State machines
Goal modelling	Desired states of actors, relation to tasks goal	• KAOS • i*

relationships between them. There is a wide range of techniques and notations for modelling and analysing requirements, each focusing on a specific aspect of the system. The choice of the appropriate modelling technique often depends on the kind of analysis and reasoning that need to be performed. Indeed, while the natural language provides a convenient way of representing requirements early in the RE process, semi-formal and formal techniques are often used to conduct more systematic and rigorous analysis later in the process. This analysis may also lead to the elicitation of additional requirements.

Table 2 shows the main categories of requirements modelling and analysing techniques. When discussing these techniques, we will use as our main example, a system for scheduling meetings [103]. In this example, a meeting initiator proposes a meeting, specifying a date range within which the meeting should take place as well as potential meeting location. Participants indicate their availability constraints, and the meeting scheduler needs to arrange a meeting that satisfies as many constraints as possible.

3.2.1 Natural Language

At the early stages of RE, it is often convenient to write requirements in a natural language, such as English. Although natural languages are very expressive, statements can be imprecise and ambiguous. Several techniques are used to improve the quality of requirements statements expressed in natural languages, and they include (1) stylistic guidelines on reducing ambiguity in requirements statements [4]; (2) requirements templates for ensuring consistency, such as Volere [89]; (3) controlled syntax for simplifying and structuring the requirements statements, such as the Easy Approach to Requirements Syntax (EARS); and (4) controlled syntax for requirements prioritisation using predefined imperatives, such as MoSCoW [17].

EARS The Easy Approach to Requirements Syntax (EARS) [75] defines a set of patterns for writing requirements using natural language as follows:

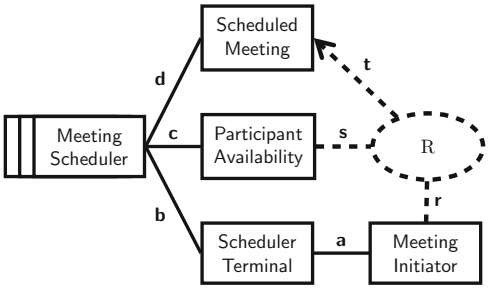
1. *Ubiquitous requirements:* define properties that the system must maintain.
   For example, the meeting scheduler `shall` display the scheduled meetings.
2. *State-driven requirements*: designate properties that must be satisfied while a pre-condition holds.
   For example, `WHILE` the meeting room is available, the meeting scheduler `shall` allow the meeting initiator to book the meeting room.
3. *Event-driven requirements*: specify properties that must be satisfied once a condition holds.
   For example, `WHEN` the meeting room is booked, the meeting scheduler `shall` display the meeting room as unavailable.
4. *Option requirements*: refer to properties satisfied in the presence of a feature.
   For example, `WHERE` a meeting is rescheduled, the meeting scheduler `shall` inform all participants.
5. *Unwanted behaviour requirements*: define the required system response to an unwanted external event.
   For example, `IF` the meeting room is unavailable, `THEN` the meeting scheduler `shall` forbid booking this meeting room.

MoSCoW Requirements are often specified in standardisation documents using a set of keywords, including `MUST`, `MUST NOT`, `SHOULD`, `SHOULD NOT`, and `MAY`. The meaning of these words was initially specified in the IEFC RFC 2119 [17]. These terms designate priority levels of requirements and are often used to determine which requirements need to be implemented first. Although controlled syntax is a step towards clarity, natural language remains inherently ambiguous and not amenable to automated, systematic, or rigorous reasoning and analysis.

3.2.2 Structural Modelling

Structural modelling techniques focus on delimiting the problem world by defining its components and their interconnections. These components might be technical or social. This category of techniques is often semi-formal; i.e. the main concepts of the technique and the relationships are defined formally, but the specification of individual components is informal, i.e. in natural language. In the following we present two structural modelling techniques: problem frames, which specify relationships between software specifications, domains, and requirements, and class diagrams, which define interconnections between classes.

Problem Frames Jackson [48] introduced the problem frames approach which emphasises the importance of structural relationships between the software (called the *machine*) and the physical world context in which the software is expected to operate. Descriptions of the structural relationships include (1) relevant components of the world (called the *problem world domains*); (2) events, states, and values these



Interface	Phenomena
a	MI!{Date, Loc}
b	ST!{Date, Loc}
c	PA!{Excl sets, Pref sets, Locs}
d	MS! {Date, Time, Loc}
r	MI!{Meeting Request}
s	PA!{Avails, Prefs}
t	SM!{Schedule}

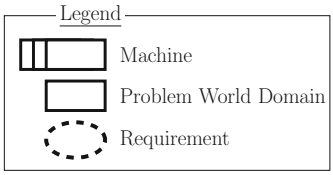


Fig. 5 Problem diagram: schedule a meeting

components share, control, and observe (called the shared phenomena); and (3) the behaviours of those components (how events are triggered, how events affect properties, and so on). Requirements are desired properties of the physical world, to be enacted by the machine (see Sect. 2.1).

Such structural relationships are partially captured using semi-formal diagrams called a *problem diagram*. Figure 5 shows the problem diagram for a requirement in the meeting scheduler problem.

The problem world domains are components in the environment that the machine (Meeting Scheduler) interacts with. They are:

- Meeting Initiator: the person who wants to schedule a meeting with other participant(s);
- Scheduler Terminal: the display terminal meeting initiator uses to schedule a meeting;
- Participant Availability: a database containing availability of all potential participants. Availability includes the exclusion set, preference set, and location requirement of every participant; and
- Scheduled Meeting: a date, time, and location of a meeting scheduled.

The solid lines a, b, c, and d are domain interfaces representing shared variables and events between the domains and the machine involved. For example, at the interface a, the variables Date and Loc are controlled by Meeting Initiator (as denoted by MI!) and can be observed by the Scheduler Terminal. In other words,

when the meeting initiator enters the data and location information of a new meeting to schedule, the terminal will receive that information. The same information is passed to the machine via the interface b. At the interface c, the machine can read the exclusion set, preference set, and location requirement of each participant. At the interface d, the machine can write the date, time, and location of a scheduled meeting.

In the diagram, the requirement is denoted by R in the dotted oval, which stands for "schedule a meeting". More precisely, the requirement means that when the meeting initiator makes a meeting request at the interface r, the meeting scheduler should find a date, time, and location for the meeting (t) such that the date and time is not in the exclusion set of any participant, the date and time is in the largest possible number of preference sets, and the location is the same as the largest possible number of location requirements of the participants (s). In other words, the requirement is a desired relationship between a meeting request, participant availability, and scheduled meeting.

Having described the requirement, the problem world domains, and the relationships among them, the requirements engineer can then focus on describing the behaviour of Meeting Scheduler (called the *specification*).

The main advantage of this structural modelling approach is that it allows the requirements engineer to separate concerns initially, to check how they are related, and to identify where the problem lies when the requirement is not satisfied.

Class Diagrams A class diagram is a graph that shows relationships between classes where a class may have one or more instances called objects. In RE, a *class* is used to represent a type of real-world objects. A *relationship* links one or more classes (a class can be linked to itself) and can also be characterised by *attributes*. The *multiplicity* on one side of a relationship specifies the minimum and maximum number of instances on this side that may be linked to an instance on the other side.

Figure 6 depicts an extract from a class diagram for the Meeting Scheduler example. The main classes are Person and Meeting, and its subclass is ScheduledMeeting. Name and Email are attributes of Person. Initiates is a relationship

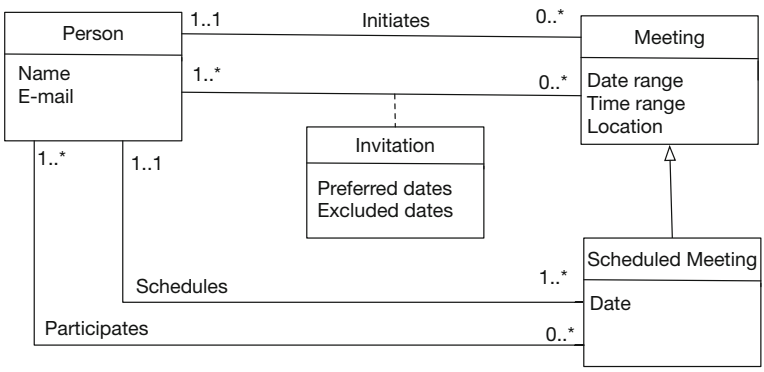


Fig. 6 A class diagram for the Meeting Scheduler example

between Person and Meeting specifying that a meeting is initiated by one and only one person (multiplicity 1..1) and that a person can initiate 0 or multiple meetings (multiplicity 0..*). For each invited participant to a meeting, the Invitation relationship specifies the preferred and excluded dates, which are attributes of this relationship.

Structural modelling techniques reveal relationships between constituents of the system-to-be, and they are helpful when scoping the problem, decomposing it, and reusing specifications. In addition to such static models, it is also useful to describe the dynamics of the system using behavioural models.

3.2.3 Behavioural Modelling

This category of modelling techniques focuses on interactions between different entities relevant to the system-to-be. They can be driven by the data exchanges between actors as in *scenarios* [2] or the events that show how an actor or actors react/respond to external or internal events as in *state machines* [22].

Scenarios A *scenario* is a description of a sequence of actions or events for a specific case of some generic task that the system needs to accomplish. A *use case* is a description of a set of actions, including variants, that a system performs that yield an observable result of value to actors. As such, a use case can be perceived as a behavioural specification of the system-to-be. It specifies all of the relevant normal course actions, variants on these actions, and potentially important alternative courses that inhibit the achievement of services and high-level system functions. In one sense, scenarios can be viewed as instances of a use case. Use cases and scenarios enable engineers to share an understanding of user needs, put them in context, elicit requirements, and explore side effects.

In the Unified Modelling Language [91], sequence diagrams are used to represent scenarios. Figure 7 shows a sequence diagram for the Meeting Scheduler example. Each actor (Meeting Initiator, Meeting Scheduler, and Participant) is associated with a timeline. Messages are shown as labelled arrows between timelines to describe information exchange between actors. For example, to start the scheduling process, the Meeting Initiator sends a MeetingRequest to the Meeting Scheduler, which then sends it to the participant. The scheduler receives the ParticipantAvail messages and sends back a MeetingResponse to the Meeting Initiator, which then makes the final decision about the date and transmits the message back to the Meeting Scheduler. Finally, the Meeting Scheduler notifies the participants about the final date for the scheduled meeting.

State Machines Sequence diagrams capture a particular interaction between actors. A classical technique for specifying the dynamics, or behaviour, of a system as a set of interactions is state machines. A state machine can be regarded as a directed graph whose nodes represent the states of a system and edges represent events leading to transit from one state to another. The start node indicates the initiation of an execution, and a final state, if there is one, indicates a successful

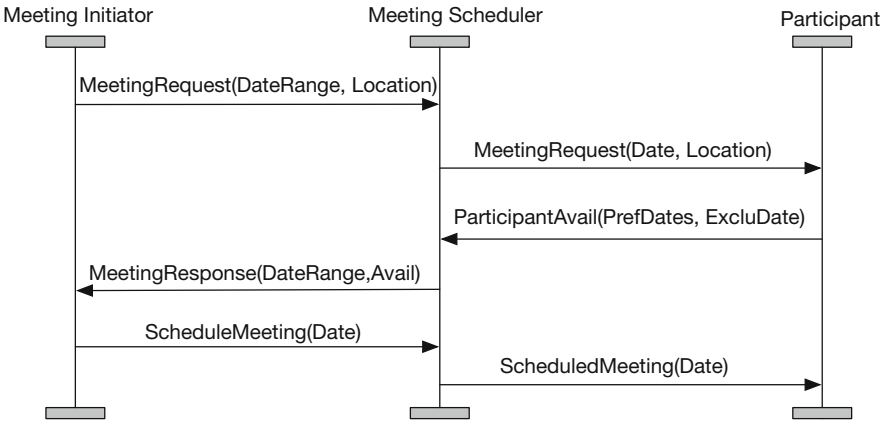


Fig. 7 A sequence diagram for the Meeting Scheduler example

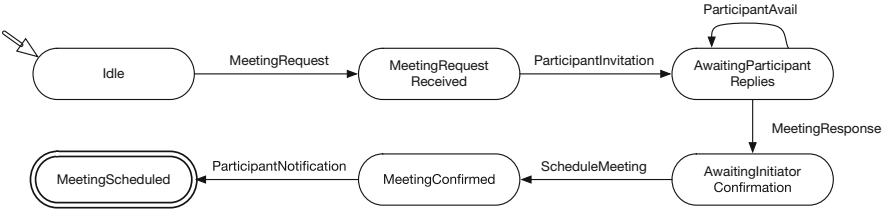


Fig. 8 A state machine for the Meeting Scheduler example

termination of an execution. Figure 8 depicts the behaviour of the Meeting Scheduler, which starts at an Idle state, in which it can receive a MeetingRequest and progresses to reach a terminating states the meeting is successfully scheduled.

In safety-critical systems, it is often important to model and analyse requirements using formal methods such as state machines [59]. Indeed, when requirements are formally specified, formal verification techniques such as model checking (which we will discuss in Sect. 3.3.2) can be used to ensure that the system, or more precisely a model thereof, satisfies those requirements [64]. However, recent studies showed that despite several successful case studies for modelling and verifying requirements formally, it remains underused in practice [73].

3.2.4 Goal Modelling

While structural modelling techniques focus on *what* constitutes the system-to-be, and behavioural modelling techniques describe *how* its different actors interact, goal modelling techniques focus on *why*, i.e. the rationale and objectives of the different system components or actors as well as *who* is responsible for realising them. In the following, we will present two goal modelling techniques: KAOS, which focuses on

refinement relationships, and i*, which focuses on dependencies in socio-technical systems.

KAOS A KAOS goal model [101] shows how goals are *refined* into sub-goals and associated domain properties. A KAOS *goal* is defined as a prescriptive statement that the system should satisfy through the cooperation of agents such as humans, devices, and software. Goals may refer to services to be provided (functional goals) or quality of service (soft goals). KAOS domain properties are descriptive statements about the environment. Besides describing the contribution of sub-goals (and associated domain properties) to the satisfaction of a goal, refinement links are also used for the operationalisation of goals. In this case, refinement links map the goals to *operations*, which are atomic tasks executed by the agents to satisfy those goals. *Conflict* links are used to represent the case of goals that cannot be satisfied together. Keywords such as Achieve, Maintain, and Avoid are used to characterise the intended behaviours of the goals and can guide their formal specification in real-time temporal logic [71]. A KAOS requirement is defined as a goal under the responsibility of a single software agent.

Figure 9 depicts an extract of the meeting scheduler where the goal of eventually getting a meeting scheduled Achieve[MeetingScheduled] is refined into a functional sub-goal consisting in booking a room and a soft sub-goal involving maximising attendance. The satisfaction of the former assumes that the domain property A room is available holds true and is assigned to the Initiator agent. The latter can be hindered by an obstacle involving a participant never available. To mitigate the risk posed by this obstacle, additional goals can be introduced.

KAOS defines a goal-oriented, model-based approach to RE which integrates many views of the system, each of which captured using an appropriate model. Traceability links are used to connect these different models. Besides the goal model representing the *intentional* view of the system, KAOS defines the following models:

- An obstacle model that enables risk analysis of the goal model by eliciting the obstacles that may obstruct the satisfaction of goals.

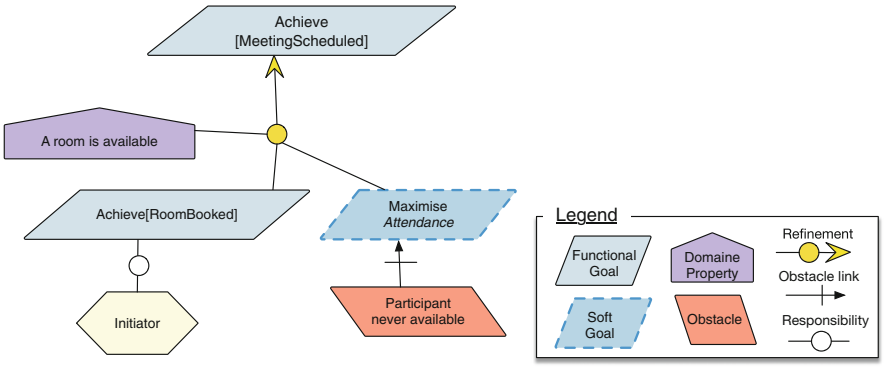


Fig. 9 A KAOS model for the Meeting Scheduler example

- An object model that captures the structural view of the system and is represented using UML class diagrams.
- An agent model that defines the agents forming the systems and for which goals they are responsible.
- An operation model represented using UML use cases.
- A behaviour model captures interaction between agents as well as the behaviour of individual agents. UML sequence diagrams are used to represent interaction between agents, while a UML state diagram specifies the admissible behaviour of a single individual agent.

i* The i* modelling approach emphasises the *who* aspect, which is paramount for modelling socio-technical systems [109]. The central notion in i* is the *actor*, who has intentional attributes such as objectives, rationale, and commitments. Besides actors, the main i* elements are *goals*, *tasks*, *soft goals*, and *resources*. A goal represents a condition or state of the world that can be achieved or not. Goals in i* mean functional requirements that are either satisfied or not. A task represents one particular way of attaining a goal. Tasks can be considered as activities that produce changes in the world. In other words, tasks enact conditions and states of the world. Resources are used in i* to model objects in the world. These objects can be physical or informational. Soft goals describe properties or constraints of the system being modelled whose achievement cannot be defined as a binary property.

Dependencies between actors are defined within a *Strategic Dependency (SD) model*. In particular, one actor (the depender) can rely on another actor (dependee) to satisfy a goal or a quality, to achieve a task, and to make a resource available. Figure 10 illustrates an SD model between three actors of the Meeting Scheduler example. The task dependency between Initiator and Meeting Scheduler indicates that these two actors interact and collaborate to OrganiseMeeting. In other words, the responsibility of performing the task is shared between these two actors. The goal dependency between Meeting Scheduler and Participant means that the former relies on the latter for satisfying the goal AvailabilityCompleted and for doing so quickly. In other words, the satisfaction of that goal is the responsibility of Participant.

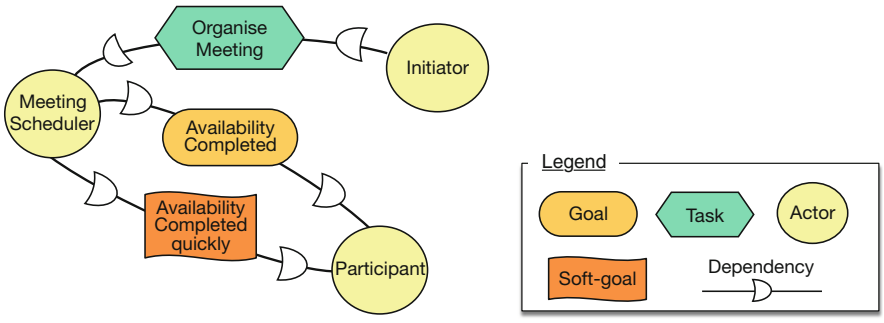


Fig. 10 An i* SD model for the Meeting Scheduler example

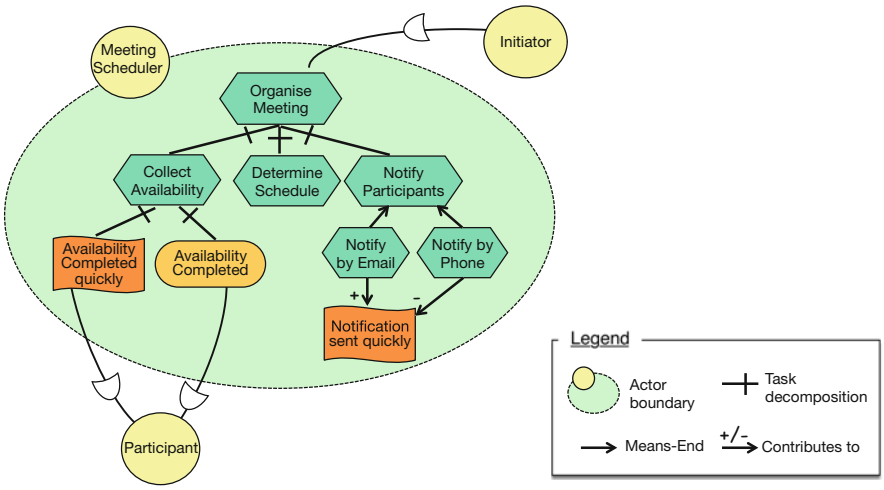


Fig. 11 An i* SR model for the Meeting Scheduler example

The *Strategic Rationale (SR) model* provides a finer-grained description by detailing what each actor can achieve by itself. It includes three additional types of links or relationships. *Task decomposition* links break down the completion of a task into several other entities. *Means-end* links indicate alternative ways for achieving a goal or a task. *Contributes-to* links indicate how satisfying a goal or performing a task can contribute positively or negatively to a soft goal. These links are included within the boundary of one actor, whereas dependency links connect different actors.

Figure 11 depicts an i* SR model where Meeting Scheduler has its SR model revealed. OrganiseMeeting is decomposed into three subtasks. To achieve CollectAvailabilty, MeetingScheduler requires availability to be completed and therefore relies on Participant to satisfy this pre-condition. MeetingScheduler is the sole responsible for performing the task DetermineSchedule. Two alternative methods can be used to notify participants about the schedule meeting: email or phone. This is represented using two means-end links to the NotifyParticipant task. As notification by email is quicker than through phone, contributes-to links are used to express this positive and negative influence. The i* modelling language has been compiled in the i* 2.0 standard [26].

3.2.5 Choosing and Combining Modelling Techniques

Requirements models can serve many purposes, facilitate discussion, document agreement, and guide implementations. By focusing on a specific aspect, each modelling technique provides an abstraction that is better suited for particular projects or at particular stages of RE. For example, structural techniques such as problem frames can help scope the problem at the early stages of the RE

process, goal-oriented techniques can be highly beneficial when the project has ill-defined objectives, and behavioural techniques with their finer-grained description are easily understood by developers. Hence, Alexander [2] advocates the need for requirements engineers to understand the benefits and assumptions of each technique and combine them to fit the specificities of the software project at hand.

3.3 Assurance

Requirements assurance seeks to determine whether requirements satisfy the qualities defined in Sect. 2.2 and to identify, report, analyse, and fix defects in requirements. It involves both validation and verification (see Fig. 12). Validation checks whether the elicited requirements reflect the real needs of the stakeholders and that they are realistic (can be afforded, do not contradict laws of nature) and consistent with domain constraints (existing interfaces and protocols). Verification of requirements analyses the coherence of requirements themselves. Verification against requirements specification involves showing and proving that the implementation of a software system conforms to its requirements specification. In the following, we discuss techniques for validating and verifying requirements.

3.3.1 Validation

Traditionally, software engineering techniques tend to focus on code quality. Yet, today more than ever software quality is defined by the users themselves. This is, e.g. reflected in the move from software quality standards such as ISO/IEC 9126 [45] to ISO/IEC 25022 [46], which put increasing emphasis on requirements analysis, design, and testing. Ultimately, what determines the success of software is its acceptance by its users [28]. Criteria as to whether a software system fulfils all basic promises, whether it fits the environment, produces no negative consequences, and that it delights the users are decisive [28]. RE places an important focus on ensuring that the requirements meet the expectations of the stakeholders from the start. To do so, some techniques focus on checking each requirement individually,

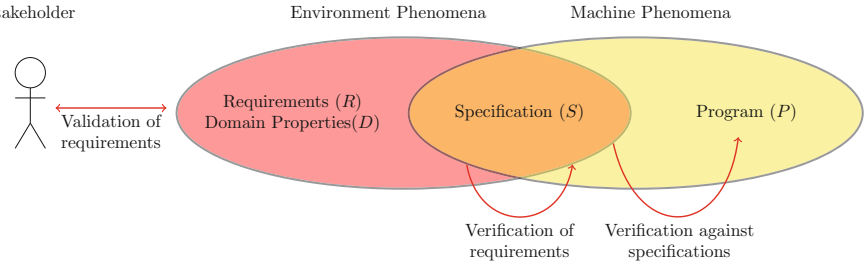


Fig. 12 Requirements assurance

e.g. quality gateway, while others regard the requirements specification as a whole, e.g. walkthroughs.

Quality Gateway The quality gateway [90] defines a set of tests that every requirement specified using the Volere template must pass. The goal of the quality gateway is to prevent incorrect requirements from passing into the design and implementation. The tests are devised to make sure that each requirement is an accurate statement of the business needs. There is no priority between the tests nor is there an order in which they are applied. In particular, each requirement must have (1) a unique identifier to be traceable, (2) a fit criterion that can be used to check whether a given solution meets the requirement, (3) a rationale to justify its existence in the first place, and (4) a customer satisfaction value to determine its value to the stakeholders. Each requirement is relevant and has no reference to specific technology which limits the number of solutions. Another test consists in verifying that all essential terms within the Volere template are defined and used consistently.

Walkthroughs Walkthroughs are inspection meetings where the requirements engineers review the specified requirements with stakeholders to discover errors, omissions, exceptions, and additional requirements [3]. Walkthroughs can take two forms: (1) free discussion between the participants or (2) structured discussions using checklists to guide the detection of defects or using scenarios and animations to facilitate understanding.

3.3.2 Verification

Verification aims to check that the requirements meet some properties (e.g. consistency) or that the software specification meets the requirements in the given domain. It can be formal as is the case with model checking or semi-formal as is the case with argumentation.

Model Checking Model checking is an automated formal verification technique for assessing whether a model of a system satisfies a desired property [22]. Model checking focuses on the behaviour of software systems, which is rigorously analysed in order to reveal potential inconsistencies, ambiguities, and incompleteness. In other words, model checking helps verifying the absence of execution errors in software systems. Once potential execution errors are detected, they can be solved either by eliminating the interactions leading to the errors or by introducing an intermediary software such that the composed behaviour satisfies the desired property. In its basic form, the model of a system is given as a state machine and the property to verify specified in temporal logic. A model checker then exhaustively explores the state space, i.e. the set of all reachable states. When a state in which the property is not valid is reachable, the path to this state is given as a counterexample, which is then used to correct the error.

Argumentation Formal verification is often insufficient in capturing various aspects of a practical assurance process. First of all, facts in our knowledge are not always complete and consistent, and yet they may still be useful to be able to make limited inferences. Secondly, new knowledge may be discovered which invalidates what was previously known, and yet it may be useful not to purge the old knowledge. Thirdly, rules of inference that are not entirely sound, and perhaps domain-dependent, may provide useful knowledge. In addressing these issues, argumentation approaches have emerged as a form of reasoning that encompasses a range of nonclassical logics [31] and is thought to be closer to human cognition [12].

Structurally an argument contains two essential parts: (1) a *claim*, a conclusion to be reached, and (2) grounds, a set of assumptions made that support the conclusion reached. A claim is usually a true/false statement, and assumptions may contain different kinds of statements, facts, expert opinions, physical measurements, and so on. There are two kinds of relationships between arguments: an argument *rebuts* another argument if they make contradicting claims, and argument *undercuts* another argument if the former contradicts some of the assumptions of the latter [12]. This structure lends itself very well to visualisation as a graph and to capturing dialogues in knowledge discovery processes.

Argumentation approaches have been used in a number of application areas including safety engineering. Kelly and Weaver [54] propose a graphical notation for presenting "safety cases" as an argument. The aim of a safety case is to present how all available evidence show that a system's operation is safe within a certain context, which can naturally be represented as an argument. In their Goal Structuring Notation (GSN), claims are represented as goals, and assumptions are categorised into different elements including solution, strategy, and context. Argumentation approaches have also been used in security engineering in order to make security engineers think clearly about the relationship between the security measures and the security objectives they aim to achieve [40]. For example, Alice might claim that her email account is secure (node A in Fig. 13) because she uses long and complex passwords for an email account (node B and black arrow to A). It is easy to undercut the argument by saying that an attacker will use a phishing attack to steal the password (node C and red arrow to the black arrow). In the

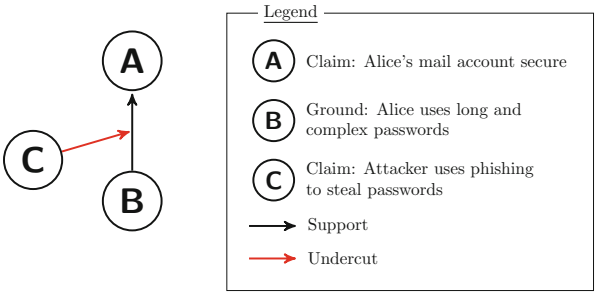


Fig. 13 A simple argument structure

security argumentation method, this counterargument could lead to the introduction of secure measures against phishing attacks.

3.4 Management and Evolution

This section explores techniques for managing conflicts and change in requirements. We start by discussing techniques for negotiating and prioritising requirements before focusing on agile methods, reuse of domain knowledge, requirements traceability, and adaptation.

3.4.1 Negotiation and Prioritisation

RE typically involves multiple stakeholders with different viewpoints and expectations, which could have conflicting requirements. Such requirements need to be identified and resolved to the extent possible before the system is implemented. Consensus building through negotiation among the stakeholders and prioritization of requirements are two main techniques for managing conflicting requirements.

Negotiation Stakeholders, including users, customers, managers, domain experts, and developers, have different expectations and interests in a software project. They may be unsure of what they can expect from the new system. Requirements negotiation aims to make informed decisions and trade-offs that would satisfy the relevant stakeholders. WinWin [15] is a requirements negotiation technique whereby the stakeholders collaboratively review, brainstorm, and agree on requirement based on defined win conditions. A win condition represents stakeholders' goals. While conflicting win conditions exist, stakeholders invent options for mutual gain and explore the option trade-offs. Options are iterated and turned into agreements when all stakeholders concur. A glossary of terms ensures that the stakeholders have a common understanding of important terms. A WinWin equilibrium is reached when stakeholders agree on all win conditions.

Prioritisation Satisfying all requirements may be infeasible given budget and schedule constraints, and as a result, requirements prioritisation may become necessary. Prioritised requirements make it easier to allocate resources and plan an incremental development process as well as to replan the project when new constraints such as unanticipated delays or budget restrictions arise. MoSCoW (see Sect. 3.2.1) is a simple prioritisation technique for categorising requirements within four groups: Must, Should, Could, and Won't. However, it is ambiguous and assumes a shared understanding among stakeholders. Karlsson and Ryan [53] propose to compare requirements pairwise and use AHP (analytic hierarchy process) to determine their relative value according to customers or users. Engineers then evaluate the cost of implementing each requirement. Finally, a cost-value diagram shows how these two criteria are related in order to guide the selection of priorities.

To determine the value of requirements, several criteria can be used, including business value, cost, customer satisfaction [90], or risk [110], as estimated by domain experts or stakeholders. The challenge is however to agree on the criteria used for prioritisation [88].

3.4.2 Agile Methods

Agile methods refer to a set of software development methods that encourage continuous collaboration with the stakeholders as well as frequent and incremental delivery of software. Rather than planning and documenting the software implementation, the requirements, the design, and the implementation emerge simultaneously and co-evolve. This section describes the characteristics of agile methods as they relate to RE activities.

Elicitation Agile methods promote face-to-face communication with customers over written specifications of requirements. As a result, agile methods assume that customers' needs can be captured when there is effective communication between customers and developers [87]. Furthermore, domain knowledge is acquired through iterative development cycles, leading to the emergence of requirements together with the design.

Modelling and Analysis Requirements are often expressed in agile development methods with user stories. User stories are designed to be simple and small. Each user story describes a simple requirement from a user's perspective in the form of "As a `[role]`, I want `[functionality]` so that `[rationale]`". Measurability, which is one of the main quality properties of requirements, is hardly specified within user stories. Nevertheless, *acceptance tests* can be used to assess the implementation of a user story. Many quality requirements are difficult to express as user stories, which are often designed to be implemented in a single iteration [87].

Assurance Agile methods focus more on requirements validation rather than verification as there is no formal modelling of requirements. Agile methods promote frequent review meetings where developers demonstrate given functionality as well as acceptance testing to evaluate, using a yes/no test, whether the user story is correctly implemented.

Management and Evolution Agile teams prioritise requirements for each development cycle rather than once for the entire project. One consequence is that requirements are prioritised more on business value rather than other criteria such as the need for a dependable architecture [87]. Agile methods are responsive to requirements change during development and are therefore well-suited to projects with uncertain and volatile requirements.

3.4.3 Reuse

Software systems are seldom designed from scratch, and many systems have a great deal of features in common. In the following we discuss how requirements can be reused across software development projects. We then discuss the role and challenges for RE when software systems are developed by reusing other software components.

Requirements Reuse Jackson [49] highlights that most software engineering projects follow *normal design* in which incremental improvements are made to understand well successful existing systems. As such, many systems are likely to have similar requirements. One technique for transferring requirements knowledge across projects is using software requirements patterns to guide the formulation of requirements [108]. A requirements pattern is applied at the level of an individual requirement and guides the specifications of a single requirement at a time. It provides a template for the specification of the requirement together with examples of requirements of the same type, suggests other requirements that usually follow on from this type of requirements, and gives hints to test or even implement this type of requirements.

In contrast to normal design, *radical design* involves unfamiliar requirements that are difficult to specify. To mitigate some of the risks associated with such radical design, agile methods introduce development cycles that continually build prototypes that help stakeholders and developers understand the problem domain.

Requirements for Reuse Modern software systems are increasingly built by assembling, and reassembling existing software components, which are possibly distributed among many devices. In this context, requirements play an essential role when evaluating, comparing, and deciding the software components to reuse. We now discuss the role of RE when reuse takes place in-house across related software systems (software product lines) or externally with increasing level of autonomy from commercial off-the-shelf (COTS) products to service-oriented systems, to systems of systems.

- *Software Product Lines.* A software product line defines a set of software products that share an important set of functionalities called *features*. The variation in features aims to satisfy the needs of different set of customers or markets. As a result, there exists a set of core requirements associated with the common features and additional requirements specific to individual products, representing their variable features. Moon et al. [77] propose an approach to collect and analyse domain properties and identify atomic requirements, described through use cases, and rules for their composition. Tun et al. [99] propose to use problem frames to relate requirements and features, which facilitates the configuration of products that systematically satisfy given requirements.
- *COTS.* Commercial off-the-shelf products are software systems purchased from a software vendor and either used as is or configured and customised to fit users' needs. Hence, the ownership of the software system-to-be is shared, and the

challenge shifted from defining requirements for developing a software system to *selecting and integrating* the appropriate COTS products [6]. The selection aims to match and find the closest fit between the requirements specifications and the specification of the COTS. The integration aims to find the wrappers or adaptors that compensate for the differences between the requirements specifications and the specification of the selected COTS.

- *Service-Oriented Systems.* Service-oriented systems rely on an abstraction that facilitates the development of distributed systems despite the heterogeneity of the underlying infrastructure, i.e. *middleware*. Indeed, software systems progressively evolved from fixed, static, and centralised to adaptable, dynamic, and distributed systems [79]. As a result, there was an increasing demand for methods, techniques, and tools to facilitate the integration of different systems. For RE, this means a shift from specifying requirements for developing a bespoke system or selecting a COTS products from one vendor to the *discovery and composition* of multiple services to satisfy the requirements of the system-to-be. A major challenge for discovery was syntactic mismatches between the specification of services and the requirements. Indeed, as the components and requirements are specified independently, the vocabulary and assumptions can be different. For composition, the challenges were related to interdependencies between service and behavioural mismatches with requirements. Semantic Web services, which provide a richer and more precise way to describe the services through the use of knowledge representation languages and ontologies, were then used to enable the selection and composition of the services even in the case of syntactic and behavioural differences [84].
- *Systems of Systems.* These are systems where constituent components are autonomous systems that are designed and implemented independently and do not obey any central control or administration. Examples include the military systems of different countries [57] or several transportation companies within a city [96]. There are often real incentives for these autonomous systems to work together, e.g. to allow international cooperation during conflicts or ensure users can commute. The RE challenges stem from the fact that each system may have its own requirements and assumptions but their collaboration often needs to satisfy other (global) requirements [60], e.g. by managing the inconsistent and conflicting requirements of these autonomous systems [104].

3.4.4 Adaptation

Increasingly software systems are used in environments with highly dynamic structures and properties. For example, a smart home may have several devices: these devices may move around the house, in and out of the house, and they could be in any of a number of states, some of which cannot be predicted until the system becomes operational. A number of phrases are used to describe such systems include "self-adaptive systems" [21] and "context-aware" systems [29]. Understanding and controlling how these systems should behave in such an environment give an

additional dimension to the challenges of RE, especially when their performance, security, and usability requirements are considered.

Research in this area is still maturing, but a few fundamentals are becoming clear [95]. First of all, there is a need to describe requirements for adaptation quite precisely. A number of proposals have been made to this end including (1) the concept of "requirements reflection" which treats the requirements model as an executable model whose states reflect the state of the running software [95], (2) the notion of "awareness-requirements" which considers the extent to which other requirements should be satisfied [97], (3) numerical quantification of requirements so that their parameters can be estimated and updated at runtime [34], and (4) rewriting of requirements to account for uncertainty in the environment so that requirements are not violated when the system encounters unexpected conditions [106].

Secondly, self-adaptive systems not only have to implement some requirements, but they also have to monitor whether their actions satisfy or violate requirements [35], often by means of a feedback loop in the environment [36]. This leads to the questions about which parts of the environment need to be monitored, how to monitor them, and how to infer requirements satisfaction from the recorded data.

Thirdly, self-adaptive systems have to relate their requirements to the system architecture at runtime. There are a number of mechanisms for doing this, including (1) by means of switching the behaviour in response to monitored changes in the environment [94], (2) by exploiting a layered architecture in order to identify alternative ways of achieving goals when obstacles are encountered, and (3) by reconfiguring components within the system architecture [55]. Jureta et al. [51] revisit Jackson and Zave to deal with self-adaptive systems by proposing configurable specifications that can be associated with a different set of requirements. At runtime, according to the environment context, different specification can be configured which satisfies predefined set of requirements. Yet, this solution requires some knowledge of all potential sets of requirements and associated specification rather than automatically reacting to changes in the environment. Controller synthesis can also be used to generate the software that satisfies requirements at runtime through decomposition and assignment to multiple agents [58] or by composing existing software components [10].

3.4.5 Traceability

Software evolution is inevitable, and we must prepare for change from the very beginning of the project and throughout software lifetime. Traceability management is a necessary ingredient for this process. Traceability is concerned with the relationships between requirements, their sources, and the system design as illustrated in Fig. 14. Hence, different artefacts can be traced at varying levels of granularity. The overall objective of traceability is to support consistency maintenance in the presence of change by ensuring that the impact of changes can be located quickly for assessment and propagation. In addition, in safety-critical systems, traceability

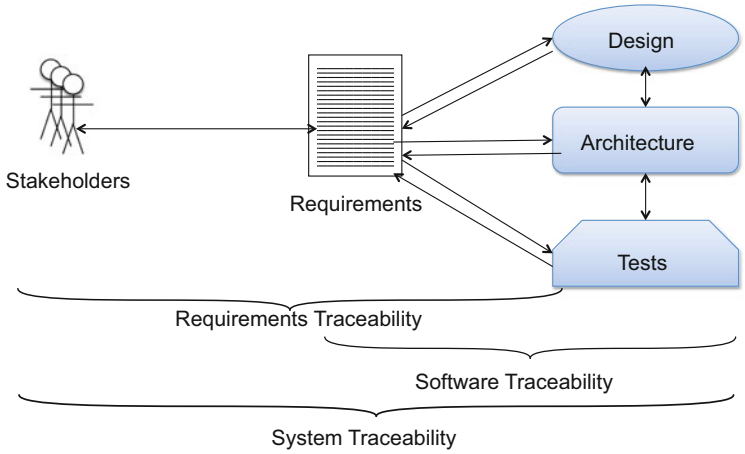


Fig. 14 Requirements, software, and system traceability

supports approval and certification by establishing links between the necessary requirements and the software complying with them. The main goal of traceability is that it is requirements-driven [24], meaning that traceability must support stakeholders' needs. Challenges for RE include tools for creating, using, and managing traceability links between relevant requirements, stakeholders, and other software artefacts across projects regardless of the software development process [37]. These challenges are made even more difficult when non-functional requirements that often cross-cut multiple components and features are considered [76].

3.5 RE for Cross-Cutting Properties

Non-functional properties (such as dependability and security) properties affect the behaviour of multiple components in the system. These properties are both important (as they are critical to success [38]) and challenging as they require holistic approaches to elicit, model, assure, and maintain non-functional requirements. These properties are typically related to system dependability, security, and user privacy. In this section, RE efforts for addressing those relating to system dependability, security, and user privacy are presented.

Dependability The notion of dependability encompasses a range of critical system properties. Among the researchers in this area, a consensus has emerged as to the main concepts and dependability of properties [7]. From RE point of view, the taxonomy and characterisation of dependability provide a way of conceptualising relationships between requirements and how they impact the system architecture. Avizienis et al. [7] propose a framework in which dependability and security are

defined in terms of attributes, threats, and means. They identify the following six attributes as being the core properties of dependable systems:

- *Availability*: correct system behaviour is provided to the user whenever demanded.
- *Reliability*: correct system behaviour continues to be available to the user.
- *Safety*: system behaviour has no bad consequences on the user and the environment.
- *Confidentiality*: no unauthorised disclosure of information.
- *Integrity*: no modification of data without authorisation or in an undetectable manner.
- *Maintainability*: system behaviour can be changed and repaired.

Confidentiality, integrity, and availability are collectively known as the security attributes.

Avizienis et al. identify three kinds of threats to dependability and security:

- *Fault*: A suspected or confirmed cause of an error.
- *Error*: A deviation from the designed system behaviour.
- *Failure*: A system behaviour that does not meet the user expectation.

There are different ways of managing threats, and they can be grouped as follows:

- *Fault prevention*: methods for preventing the introduction or occurrence of faults
- *Fault tolerance*: methods for avoiding failures in the presence of faults
- *Fault removal*: methods for reducing or eliminating the occurrence of faults
- *Fault forecasting*: methods for estimating the occurrence and consequence of faults

From the RE point of view, the dependability framework is valuable because a systematic consideration of the dependability requirements could have a significant impact on the system architecture and its implementation. For example, system dependability can be improved by ensuring that critical requirements are satisfied by a small subset of components in the system [52].

Security In recent years, the security of software systems has come under the spotlight. There are two main ways in which the term "security" is used in the context of RE. In a formal sense, security tends to be a triple of properties, known as CIA properties: confidentiality, integrity, and availability of information (sometimes called information security). In a broader sense, the term security is used to capture the need for protecting valuable assets from harm [40] (sometimes called "system security"). In RE approaches, analysis often begins with requirements for system security, and many of them are often refined into CIA properties. Several RE approaches to system security have been surveyed by Nhlabatsi et al. [78].

Rushby [92] suggests that it is difficult to write security requirements because some security properties do not match with behavioural properties that can be

expressed using formal methods and also because security requirements are counterfactual (i.e. you do not know what the security requirements are until the system is compromised by an attacker).

One way to think about security requirements is by looking at the system from the point of view of an attacker: What would an attacker want to be able to do with the system? Perhaps they want to steal the passwords stored on the server. Requirements of an attacker are called negative requirements or anti-requirements [25]. Once identified, the software engineer has to design the system that prevents the anti-requirements from being satisfied. The idea has been extended by considering various patterns of anti-requirements, known as "abuse frames" [62]. In goal-oriented modelling, anti-requirements are called anti-goals, and the anti-goals can be refined in order to identify obstacles to security goals, and generate countermeasures [100]. In a similar vein, a systematic process to analyse security requirements in a social and organisational setting has been proposed [63]. Complementing these attacker-focused approaches to engineering security requirements is the notion of defence in depth, which calls for ever more detailed analysis of threats and defence mechanisms. This questioning attitude to security is well supported by argumentation frameworks. In one line of work, formal and semi-formal argumentation approaches have been used to reason about system security [40].

Privacy For software systems dealing with personal and private information, there are growing concerns for the protection of privacy. In some application areas such as health and medicine, there are legal and regulatory frameworks for respecting privacy. There are systematic approaches for obtaining requirements from existing legal texts [18] and for designing adaptive privacy [83].

4 Future Challenges

> Why should I bother writing requirements? Engineers focus on building things. Requirements are dead!
> No, RE isn't only about documents.
> Well, anyway, there isn't any real business using goal models!
> Would you use RE for driveless car?

These statements exemplify the current debate around requirements engineering. In its early years, requirements engineering was about the importance of specifying requirements, focusing on the "*What*" instead of the "*How*". It then moved to systematic processes and methods, focusing on the "*Why*". It has then grown steadily over the years. The achievements were reflected in requirements engineering being part of several software engineering standards and processes. Yet, the essence of RE remains the same: it involves good understanding of problems [66], which includes analysing the domain, communicating with stakeholders, and preparing for system evolution. So what have changed in those years?

On the one hand, techniques such as machine learning, automated compositions, and creativity disrupt the traditional models of software development and call for quicker, if not immediate, response from requirements engineering. On the other hand, the social underpinning and the increasing reliance on software systems for every aspect of our life call for better methods to understand the impact and implications of software solutions on the well-being of individuals and society as a whole. For example, online social networks with their privacy implications and their societal, legal, and ethical impact require some understanding of the domain in which the software operate.

In addition, a number of pressing global problems such as climate change and sustainability engineering as well as increasingly important domains such as user-centred computing and other inter- and cross-disciplinary problems challenge existing processes and techniques. Yet, the fundamentals of RE are likely to be the same. The intrinsic ability of RE to deal with conflicts, negotiation, and its traditional focus on tackling those *wicked* problems is highly beneficial. We now summarise some trends influencing the evolution of requirements engineering as a discipline.

4.1 Sustainability and Global Societal Challenges

Software is now evolving to encompass a world where the boundary between the machine and human disappears, merging wearable with the Internet of Things into "a digital universe of people, places and things" [1]. This ubiquitous connectivity and digitisation open up endless opportunities for addressing pressing societal problems defined by the United Nations as Sustainable Development Goals,[1] e.g. eradicating hunger, ensuring health and well-being, and building resilient infrastructure and fostering innovation. These problems are wicked problems that RE has long aspired to address [32], but they still challenge existing RE techniques and processes [39]. First, the *multi- and cross-disciplinary* nature of these problems makes it hard to understand them, let alone to specify them. In addition, while *collaboration* between systems and people is important, stakeholders may have radically different views when addressing them. As each of those problems is novel and unique, they involve radical design and thereby require dealing with *failures* to adapt and adjust solutions iteratively [49].

Multidisciplinarity The need for multidisciplinary training for requirements engineers has been advocated since 2000s [82]. Agile methods also promote multi-functional teams [56]. Furthermore, Robertson and Maiden [69] highlight the need to be creative during the RE process. But nowadays, requirements engineers need to become global problem solvers with the ability to communicate, reflect, and mediate

[1] http://www.un.org/sustainabledevelopment/.

between domain experts and software engineers as well as to invent solutions. Early empirical evidence is given in the domain of sustainability as to the role of the RE mindset and its inherent focus on considering multiple perspectives to build shared understanding in an adaptive, responsive, and iterative manner [8].

Collaboration Collaboration between software engineering teams to find software solutions has attracted a lot of interest [42], so has the collaboration between software components for adaptation and interoperability [10]. Software ecosystems that compose software platforms as well as communities of developers, domain experts, and users are becoming increasingly common [16]. The intentional aspects of those ecosystems need to be well understood in order for the impact of collaboration and interconnection to be specified rather than just incurred. The requirements for emergent collaborations between people and technology and the theory and processes for understanding them are still to be defined.

Failure In extremely complex systems, failure is inevitable [61]. In order to make systems more resilient, it is important to be able to anticipate, inject, and control errors so that the side effects that are not necessary foreseen at the design time are better understood [93]. But embracing failure necessitates learning from it and distilling appropriate knowledge into the design, which is often at the heart of RE [49].

4.2 Artificial Intelligence

The research discipline of RE has focused on capturing lessons, developing strategies and techniques, and building tools to assist with the creation of software systems. Many of the related tasks, from scoping to operationalisation, were human-driven, but increasingly artificial intelligence (AI) techniques are able to assist with those tasks [86, 9]. For example, machine learning can be viewed as a tool for building a system inductively from a set of input-output examples, where specifications of such a system are given as training data sets [74]. In this context, requirements are used to guide the selection of training data. Without having this selection in line with stakeholders' needs, the learnt system may diverge from their initial purpose, as it happened with Microsoft Tay chatbot [105]. Tay was a machine learning project designed for user engagement but which has learnt inappropriate language and commentary due to the data used in the learning process. In addition, transparency requirements [43] can also play an important role in increasing users' confidence in the system by explaining the decision made with the software system.

4.3 Exemplars and Artefacts

In their 2007 survey paper, Cheng and Atlee [20] highlighted the need for the RE community to be proactive in identifying the new computing challenges.

Ten years later, RE still focuses on conceptual frameworks and reflects rather than leads and invents new application domains. Evidence is somehow given by the lack of extensive RE benchmarks and model problems (besides the meeting scheduler [103], lift management system [72], or railroad crossing control [41]) despite some successful case studies in the domain of critical systems such as aviation [59] and space exploration [65]. Yet as discussed in this section, RE can play an immense role in leading the way for understanding and eliciting alternative solutions for solving global societal challenges. The discipline of RE may need to move from reflection to disruption [33].

5 Conclusion

Requirements are inherent to any software system whether or not they are made explicit during the development. During its early days, RE research focused on understanding the nature of requirements, relating RE to other software engineering activities, and setting out the requirements processes. RE was then concerned with defining the essential qualities of requirements, which would make them easy to analyse and to change by developers. Later on a specific focus was given to particular activities within requirements engineering, such as modelling, assurance, and managing change. In the context of evolution, reuse and adaptation have become active areas in research. Agile and distributed software development environments have challenged the traditional techniques for specifying and documenting requirements. Although the tools for writing, documenting, and specifying requirements may differ, the principles of RE relating to domain understanding, creativity, and retrospection are still important, and will probably remain so. With the complexity and ubiquity of software in society, the interplay between different technical, economical, and political issues calls for the kinds of tools and techniques developed by RE research.

References

1. Abowd, G.D.: Beyond weiser: from ubiquitous to collective computing. IEEE Comput. **49**(1), 17–23 (2016). http://dx.doi.org/10.1109/MC.2016.22
2. Alexander, I.: Gore, sore, or what? IEEE Softw. **28**(1), 8–10 (2011). http://dx.doi.org/10.1109/MS.2011.7
3. Alexander, I.F., Maiden, N.: Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle. Wiley, New York (2005)
4. Alexander, I.F., Stevens, R.: Writing Better Requirements. Pearson Education, Harlow (2002)
5. Altran: Reveal tm. http://intelligent-systems.altran.com/fr/technologies/systems-engineering/revealtm.html
6. Alves, C.F., Finkelstein, A.: Challenges in COTS decision-making: a goal-driven requirements engineering perspective. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE 2002, Ischia, 15–19 July 2002, pp. 789–794 (2002). http://doi.acm.org/10.1145/568760.568894

7. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. **1**(1), 11–33 (2004). https://doi.org/10.1109/TDSC.2004.2
8. Becker, C., Betz, S., Chitchyan, R., Duboc, L., Easterbrook, S.M., Penzenstadler, B., Seyff, N., Venters, C.C.: Requirements: the key to sustainability. IEEE Softw. **33**(1), 56–65 (2016). http://dx.doi.org/10.1109/MS.2015.158
9. Bencomo, N., Cleland-Huang, J., Guo, J., Harrison, R. (eds.): IEEE 1st International Workshop on Artificial Intelligence for Requirements Engineering, AIRE 2014, 26 Aug 2014, Karlskrona. IEEE Computer Society, Washington (2014). http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6887463
10. Bennaceur, A., Nuseibeh, B.: The many facets of mediation: a requirements-driven approach for trading-off mediation solutions. In: Mistrík, I., Ali, N., Grundy, J., Kazman, R., Schmerl, B. (eds.) Managing Trade-offs in Adaptable Software Architectures. Elsevier, New York (2016). http://oro.open.ac.uk/45253/
11. Berander, P., Andrews, A.: Requirements prioritization. In: Aurum, A., Wohlin, C. (eds.) Engineering and Managing Software Requirements, pp. 69–94. Springer, Berlin (2005)
12. Besnard, P., Hunter, A.: Elements of Argumentation. The MIT Press, Cambridge (2008)
13. Boehm, B.W.: Verifying and validating software requirements and design specifications. IEEE Softw. **1**(1), 75–88 (1984). http://dx.doi.org/10.1109/MS.1984.233702
14. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Comput. **21**(5), 61–72 (1988). http://dx.doi.org/10.1109/2.59
15. Boehm, B.W., Grünbacher, P., Briggs, R.O.: Developing groupware for requirements negotiation: lessons learned. IEEE Softw. **18**(3), 46–55 (2001). http://dx.doi.org/10.1109/52.922725
16. Bosch, J.: Speed, data, and ecosystems: the future of software engineering. IEEE Softw. **33**(1), 82–88 (2016). http://dx.doi.org/10.1109/MS.2016.14
17. Bradner, S.: Key words for use in RFCs to indicate requirement levels (1997). http://www.ietf.org/rfc/rfc2119.txt
18. Breaux, T., Antón, A.: Analyzing regulatory rules for privacy and security requirements. IEEE Trans. Softw. Eng. **34**(1), 5–20 (2008). https://doi.org/10.1109/TSE.2007.70746
19. Brooks, F.P. Jr.: No silver bullet essence and accidents of software engineering. Computer **20**(4), 10–19 (1987). http://dx.doi.org/10.1109/MC.1987.1663532
20. Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: Proceedings of the Workshop on the Future of Software Engineering, FOSE, pp. 285–303 (2007)
21. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.D.M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: a research roadmap. In: Software Engineering for Self-Adaptive Systems [Outcome of a Dagstuhl Seminar], pp. 1–26 (2009)
22. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. **28**(4), 626–643 (1996)
23. Cleland-Huang, J., Gotel, O., Zisman, A. (eds.): Software and Systems Traceability. Springer, London (2012). http://dx.doi.org/10.1007/978-1-4471-2239-5
24. Cleland-Huang, J., Gotel, O., Hayes, J.H., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: Proceedings of the Future of Software Engineering, FOSE@ICSE, pp. 55–69 (2014). http://doi.acm.org/10.1145/2593882.2593891
25. Crook, R., Ince, D.C., Lin, L., Nuseibeh, B.: Security requirements engineering: when anti-requirements hit the fan. In: 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE 2002), Essen, 9–13 Sept 2002, pp. 203–205. IEEE Computer Society, Washington (2002)
26. Dalpiaz, F., Franch, X., Horkoff, J.: istar 2.0 language guide. CoRR abs/1605.07767 (2016). http://arxiv.org/abs/1605.07767

27. Davis, A.M., Tubío, Ó.D., Hickey, A.M., Juzgado, N.J., Moreno, A.M.: Effectiveness of requirements elicitation techniques: empirical results derived from a systematic review. In: Proceedings of the 14th IEEE International Conference on Requirements Engineering, RE, pp. 176–185 (2006). http://dx.doi.org/10.1109/RE.2006.17
28. Denning, P.J.: Software quality. Commun. ACM **59**(9), 23–25 (2016). http://doi.acm.org/10.1145/2971327
29. Dey, A.K., Abowd, G.D., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum.-Comput. Interact. **16**(2), 97–166 (2001)
30. Dieste, O., Juzgado, N.J.: Systematic review and aggregation of empirical studies on elicitation techniques. IEEE Trans. Softw. Eng. **37**(2), 283–304 (2011). http://dx.doi.org/10.1109/TSE.2010.33
31. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. **77**(2), 321–357 (1995). http://dx.doi.org/10.1016/0004-3702(94)00041-X; http://www.sciencedirect.com/science/article/pii/000437029400041X
32. Easterbrook, S.: What is requirements engineering? (2004). http://www.cs.toronto.edu/~sme/papers/2004/FoRE-chapter01-v7.pdf
33. Ebert, C., Duarte, C.H.C.: Requirements engineering for the digital transformation: industry panel. In: 2016 IEEE 24th International Requirements Engineering Conference (RE), pp. 4–5 (2016). https://doi.org/10.1109/RE.2016.21
34. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pp. 111–121. IEEE Computer Society, Washington (2009). http://dx.doi.org/10.1109/ICSE.2009.5070513
35. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, 1995, pp. 140–147 (1995). https://doi.org/10.1109/ISRE.1995.512555
36. Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, A.B., Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., Krikava, F., Misailovic, S., Papadopoulos, A.V., Ray, S., Sharifloo, A.M., Shevtsov, S., Ujma, M., Vogel, T.: Software engineering meets control theory. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-managing Systems, pp. 71–82 (2015). https://doi.org/10.1109/SEAMS.2015.12
37. Furtado, F., Zisman, A.: Trace++: a traceability approach for agile software engineering. In: Proceedings of the 24th International Requirements Engineering Conference, RE (2016)
38. Glinz, M.: On non-functional requirements. In: Proceedings of the 15th IEEE International Requirements Engineering Conference, RE, pp. 21–26 (2007). https://doi.org/10.1109/RE.2007.45
39. Guenther Ruhe, M.N., Ebert, C.: The vision: requirements engineering in society. In: Proceedings of the 25th International Requirements Engineering Conference - Silver Jubilee Track, RE (2017). https://www.ucalgary.ca/mnayebi/files/mnayebi/the-vision-requirements-engineering-in-society.pdf
40. Haley, C.B., Laney, R.C., Moffett, J.D., Nuseibeh, B.: Security requirements engineering: a framework for representation and analysis. IEEE Trans. Softw. Eng. **34**(1), 133–153 (2008). http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70754
41. Heitmeyer, C.L., Labaw, B., Jeffords, R.: A benchmark for comparing different approaches for specifying and verifying real-time systems. In: Proceedings of the 10th International Workshop on Real-Time Operating Systems and Software (1993)
42. Herbsleb, J.D.: Building a socio-technical theory of coordination: why and how (outstanding research award). In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, 13–18 Nov 2016, pp. 2–10 (2016). http://doi.acm.org/10.1145/2950290.2994160

43. Hosseini, M., Shahri, A., Phalp, K., Ali, R.: Four reference models for transparency requirements in information systems. Requir. Eng. **23**, 1–25 (2017)
44. IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board: IEEE recommended practice for software requirements specifications. Technical report, IEEE (1998)
45. ISO/IEC 9126: Software engineering – product quality – part 1: quality model. Technical report, ISO (2001)
46. ISO/IEC 25022: Systems and software engineering – systems and software quality requirements and evaluation (square) – measurement of quality in use. Technical report, ISO (2016)
47. Jackson, M.: Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices. ACM Press/Addison-Wesley, New York (1995)
48. Jackson, M.: Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley Longman, Boston (2001)
49. Jackson, M.: The name and nature of software engineering. In: Advances in Software Engineering: Lipari Summer School 2007. Revised Tutorial Lectures in Advances in Software Engineering, pp. 1–38. Springer, Berlin (2007). http://dx.doi.org/10.1007/978-3-540-89762-0_1
50. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: Proceedings of the 17th International Conference on Software Engineering, ICSE '95, pp. 15–24. ACM, New York (1995). http://doi.acm.org/10.1145/225014.225016
51. Jureta, I., Borgida, A., Ernst, N.A., Mylopoulos, J.: The requirements problem for adaptive systems. ACM Trans. Manag. Inf. Syst. **5**(3), 17 (2014). http://doi.acm.org/10.1145/2629376
52. Kang, E., Jackson, D.: Dependability arguments with trusted bases. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, RE '10, pp. 262–271. IEEE Computer Society, Washington (2010). http://dx.doi.org/10.1109/RE.2010.38
53. Karlsson, J., Ryan, K.: A cost-value approach for prioritizing requirements. IEEE Softw. **14**(5), 67–74 (1997). http://dx.doi.org/10.1109/52.605933
54. Kelly, T., Weaver, R.: The goal structuring notation–a safety argument notation. In: Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases (2004)
55. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Proceedings of the Future of Software Engineering track, FOSE@ICSE, pp. 259–268 (2007). http://dx.doi.org/10.1109/FOSE.2007.19
56. Laplante, P.A.: Requirements Engineering for Software and Systems. CRC Press, Boca Raton (2013)
57. Larson, E.: Interoperability of us and nato allied air forces: supporting data and case studies. Technical report 1603, RAND Corporation (2003)
58. Letier, E., Heaven, W.: Requirements modelling by synthesis of deontic input-output automata. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, 18–26 May 2013, pp. 592–601 (2013). http://dl.acm.org/citation.cfm?id=2486866
59. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Trans. Softw. Eng. **20**(9), 684–707 (1994). https://doi.org/10.1109/32.317428
60. Lewis, G.A., Morris, E., Place, P., Simanta, S., Smith, D.B.: Requirements engineering for systems of systems. In: 2009 3rd Annual IEEE Systems Conference, pp. 247–252 (2009). https://doi.org/10.1109/SYSTEMS.2009.4815806
61. Limoncelli, T.A.: Automation should be like iron man, not ultron. ACM Queue **13**(8), 50 (2015). http://doi.acm.org/10.1145/2838344.2841313
62. Lin, L., Nuseibeh, B., Ince, D.C., Jackson, M.: Using abuse frames to bound the scope of security problems. In: 12th IEEE International Conference on Requirements Engineering (RE 2004), Kyoto, 6–10 Sept 2004, pp. 354–355 (2004)
63. Liu, L., Yu, E.S.K., Mylopoulos, J.: Security and privacy requirements analysis within a social setting. In: 11th IEEE International Conference on Requirements Engineering (RE 2003), 8–12 Sept 2003, Monterey Bay, pp. 151–161 (2003)

64. Lutz, R.R.: Analyzing software requirements errors in safety-critical, embedded systems. In: Proceedings of IEEE International Symposium on Requirements Engineering, RE, pp. 126–133 (1993). https://doi.org/10.1109/ISRE.1993.324825
65. Lutz, R.R.: Software engineering for space exploration. IEEE Comput. **44**(10), 41–46 (2011). https://doi.org/10.1109/MC.2011.264
66. Maiden, N.A.M.: So, what is requirements work? IEEE Softw. **30**(2), 14–15 (2013). http://dx.doi.org/10.1109/MS.2013.35
67. Maiden, N.A.M., Rugg, G.: ACRE: selecting methods for requirements acquisition. Softw. Eng. J. **11**(3), 183–192 (1996). http://dx.doi.org/10.1049/sej.1996.0024
68. Maiden, N.A.M., Gizikis, A., Robertson, S.: Provoking creativity: imagine what your requirements could be like. IEEE Softw. **21**(5), 68–75 (2004). http://dx.doi.org/10.1109/MS.2004.1331305
69. Maiden, N.A.M., Robertson, S., Robertson, J.: Creative requirements: invention and its role in requirements engineering. In: Proceedings of the 28th International Conference on Software Engineering, ICSE, pp. 1073–1074 (2006). http://doi.acm.org/10.1145/1134512
70. Mancini, C., Rogers, Y., Bandara, A.K., Coe, T., Jedrzejczyk, L., Joinson, A.N., Price, B.A., Thomas, K., Nuseibeh, B.: Contravision: exploring users' reactions to futuristic technology. In: Proceedings of the 28th International Conference on Human Factors in Computing Systems, CHI 2010, Atlanta, 10–15 April 2010, pp. 153–162 (2010)
71. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems - Specification. Springer, Berlin (1992)
72. Marca, D., Harandi: Problem set for the fourth international workshop on software specification and design. In: Proceedings of the 4th International Workshop on Software Specification and Design (1987)
73. Martins, L.E.G., Gorschek, T.: Requirements engineering for safety-critical systems: overview and challenges. IEEE Softw. **34**(4), 49–57 (2017). https://doi.org/10.1109/MS.2017.94
74. Maruyama, H.: Machine learning as a programming paradigm and its implications to requirements engineering. In: Asia-Pacific Requirements Engineering Symposium, APRES (2016)
75. Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (EARS). In: Proceedings of the 17th IEEE International Requirements Engineering Conference, RE, pp. 317–322 (2009). http://dx.doi.org/10.1109/RE.2009.9
76. Mirakhorli, M., Cleland-Huang, J.: Tracing non-functional requirements. In: Software and Systems Traceability, pp. 299–320. Springer, Berlin (2012). http://dx.doi.org/10.1007/978-1-4471-2239-5_14
77. Moon, M., Yeom, K., Chae, H.S.: An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. IEEE Trans. Softw. Eng. **31**(7), 551–569 (2005). http://dx.doi.org/10.1109/TSE.2005.76
78. Nhlabatsi, A., Nuseibeh, B., Yu, Y.: Security requirements engineering for evolving software systems: a survey. Int. J. Secur. Softw. Eng. **1**(1), 54–73 (2010)
79. Nitto, E.D., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Autom. Softw. Eng. **15**(3–4), 313–341 (2008). http://dx.doi.org/10.1007/s10515-008-0032-x
80. Niu, N., Easterbrook, S.M.: So, you think you know others' goals? A repertory grid study. IEEE Softw. **24**(2), 53–61 (2007). http://dx.doi.org/10.1109/MS.2007.52
81. Nuseibeh, B.: Weaving together requirements and architectures. IEEE Comput. **34**(3), 115–117 (2001). http://dx.doi.org/10.1109/2.910904
82. Nuseibeh, B., Easterbrook, S.M.: Requirements engineering: a roadmap. In: Proceedings of the Future of Software Engineering Track at the 22nd International Conference on Software Engineering, Future of Software Engineering Track, FOSE, pp. 35–46 (2000). http://doi.acm.org/10.1145/336512.336523

83. Omoronyia, I., Cavallaro, L., Salehie, M., Pasquale, L., Nuseibeh, B.: Engineering adaptive privacy: on the role of privacy awareness requirements. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, 18–26 May 2013, pp. 632–641 (2013). http://dx.doi.org/10.1109/ICSE.2013.6606609
84. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: Proceedings of the International Semantic Web Conference, ISWC, pp. 333–347 (2002)
85. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995). http://dx.doi.org/10.1016/0167-6423(95)96871-J
86. Pohl, K., Assenova, P., Dömges, R., Johannesson, P., Maiden, N., Plihon, V., Schmitt, J.R., Spanoudakis, G.: Applying ai techniques to requirements engineering: the nature prototype. In: Proceedings of the ICSE-Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence (1994)
87. Ramesh, B., Cao, L., Baskerville, R.: Agile requirements engineering practices and challenges: an empirical study. Inf. Syst. J. **20**(5), 449–480 (2010). http://dx.doi.org/10.1111/j.1365-2575.2007.00259.x
88. Riegel, N., Dörr, J.: A systematic literature review of requirements prioritization criteria. In: Proceedings of the 21st International Working Conference on Requirements Engineering: Foundation for Software Quality, REFSQ, pp. 300–317 (2015). http://dx.doi.org/10.1007/978-3-319-16101-3_22
89. Robertson, S., Robertson, J.: Mastering the Requirements Process. ACM Press/Addison-Wesley, New York (1999)
90. Robertson, S., Robertson, J.: Mastering the Requirements Process: Getting Requirements Right. Addison-Wesley, Boston (2012)
91. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Pearson Higher Education, London (2004)
92. Rushby, J.: Security requirements specifications: how and what? In: Requirements Engineering for Information Security (SREIS), Indianapolis (2001)
93. Russo, D., Ciancarini, P.: A proposal for an antifragile software manifesto. Proc. Comput. Sci. **83**, 982–987 (2016)
94. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying monitoring and switching problems in context. In: 15th IEEE International Requirements Engineering Conference (RE 2007) (2007). http://oro.open.ac.uk/10264/
95. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-aware systems: a research agenda for re for self-adaptive systems. In: 2010 18th IEEE International Requirements Engineering Conference, pp. 95–103 (2010). https://doi.org/10.1109/RE.2010.21
96. SECUR-ED, C.: Deliverable d22.1: Interoperability concept. fp7 SECUR-ED EU project (2012). http://www.secur-ed.eu/?page_id=33
97. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, pp. 60–69. ACM, New York (2011). http://doi.acm.org/10.1145/1988008.1988018
98. Sommerville, I., Sawyer, P.: Requirements Engineering: A Good Practice Guide, 1st edn. Wiley, New York (1997)
99. Tun, T.T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P.: Relating requirements and feature configurations: a systematic approach. In: Proceedings of the 13th International Conference on Software Product Lines, SPLC, pp. 201–210 (2009). http://doi.acm.org/10.1145/1753235.1753263
100. van Lamsweerde, A.: Elaborating security requirements by construction of intentional anti-models. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) 26th International Conference on Software Engineering (ICSE 2004), Edinburgh, 23–28 May 2004, pp. 148–157. IEEE Computer Society, Washington (2004)

101. van Lamsweerde, A.: Requirements engineering: from craft to discipline. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, 9–14 Nov 2008, pp. 238–249 (2008). http://doi.acm.org/10.1145/1453101.1453133
102. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley, Hoboken (2009)
103. van Lamsweerde, A., Darimont, R., Massonet, P.: Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In: Proceedings of the 2nd IEEE International Symposium on Requirements Engineering, RE, pp. 194–203 (1995). http://dx.doi.org/10.1109/ISRE.1995.512561
104. Viana, T., Bandara, A., Zisman, A.: Towards a framework for managing inconsistencies in systems of systems. In: Proceedings of the Colloquium on Software-intensive Systems-of-Systems at 10th European Conference on Software Architecture (2016). http://oro.open.ac.uk/48014/
105. Wakefield, J.: Microsoft chatbot is taught to swear on twitter. Visited on 30 Mar 2017
106. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M.: Relax: a language to address uncertainty in self-adaptive systems requirement. Requir. Eng. **15**(2), 177–196 (2010). http://dx.doi.org/10.1007/s00766-010-0101-0
107. Wiegers, K., Beatty, J.: Software Requirements. Pearson Education, Harlow (2013)
108. Withall, S.: Software Requirement Patterns, 1st edn. Microsoft Press, Redmond (2007)
109. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: 3rd IEEE International Symposium on Requirements Engineering (RE'97), Annapolis, 5–8 Jan 1997, pp. 226–235. IEEE Computer Society, Washington (1997). http://dx.doi.org/10.1109/ISRE.1997.566873
110. Yu, Y., Franqueira, V.N.L., Tun, T.T., Wieringa, R., Nuseibeh, B.: Automated analysis of security requirements through risk-based argumentation. J. Syst. Softw. **106**, 102–116 (2015). http://dx.doi.org/10.1016/j.jss.2015.04.065
111. Zave, P.: Classification of research efforts in requirements engineering. ACM Comput. Surv. **29**(4), 315–321 (1997). http://doi.acm.org/10.1145/267580.267581
112. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. **6**(1), 1–30 (1997). http://doi.acm.org/10.1145/237432.237434
113. Zowghi, D., Coulin, C.: Requirements Elicitation: A Survey of Techniques, Approaches, and Tools, pp. 19–46. Springer, Berlin (2005)