Requirements Problem and Solution Concepts for Adaptive Systems Engineering, and their Relationship to Mathematical Optimisation, Decision Analysis, and Expected Utility Theory

Ivan J. Jureta

Abstract Requirements Engineering (RE) focuses on eliciting, modelling, and analyzing the requirements and environment of a system-to-be in order to design its specification. The design of the specification, usually called the Requirements Problem (RP), is a complex problem solving task, as it involves, for each new system-to-be, the discovery and exploration of, and decision making in, new and ill-defined problem and solution spaces. The default RP in RE is to design a specification of the system-to-be which (i) is consistent with given requirements and conditions of its environment, and (ii) together with environment conditions satisfies requirements. This paper (i) shows that the Requirements Problem for Adaptive System (RPAS) is different from, and is not a subclass of the default RP, (ii) gives a formal definition of the RPAS, and (iii) discusses implications for future research.

1 Introduction

1.1 Domain: Requirements Engineering

Requirements Engineering (RE) focuses on eliciting, modelling, and analysing the requirements and environment of a system-to-be in order to design its specification.

It is on the basis of its specification that the system is built, updated, changed, its new releases planned, made, announced, rolled out. Specifications can take different forms, ranging from minimalistic to-do lists that hint at expectations and subsume implicit engineering solutions, to elaborately structured documentation on contracts with employees and suppliers, responsibilities of positions in the value chain, guide-

Ivan J. Jureta
Fonds de la Recherche Scientifique – FNRS and Department of Business Administration, University of Namur, e-mail: ivan.jureta@unamur.be

lines for employee coordination and collaboration, as well as formal software specifications made for use with a model checker.

The design of the specification, usually called the **Requirements Problem (RP)**, is a complex problem solving task, as it involves, for each new system-to-be, the discovery and exploration of, and decision making in, new and ill-defined problem and solution spaces.

Difficulties involved in solving an RP instance are illustrated by the variety of topics studied in RE research, such as requirements elicitation [23, 28, 16], categorization [14, 61, 35], vagueness and ambiguity [45, 43, 34], prioritization [36, 2, 27], negotiation [42, 3, 32], responsibility allocation [14, 11, 21], cost estimation [4, 7, 53], conflicts and inconsistency [46, 26, 57], comparison [45, 43, 44], satisfaction evaluation [6, 45, 41], operationalization [22, 21, 17], traceability [24, 50, 13], and change [12, 59, 10].

RE issues are present when designing new and changing existing systems; they are there whatever the system class and domain, and regardless of the extent to which people are involved in the system: from autonomic Internet-scale clouds, to traditional desktop applications, industrial expert systems, and embedded software, all enabling anything from massive mobile apps ecosystems, global supply chains, medical processes, business processes, mobile gaming, and so on. Moreover, RE issues are present regardless of how the software in the system is designed and made, from a military waterfall to a startup's own agile dialect, and from organisations where developers talk directly to customers, to those where product designers, salespeople, or others mediate between requirements and code.

1.2 Context: Default Requirements Problem

The *de facto* default view in RE, is that the specification is produced *incrementally*, starting from a limited set of incomplete, inconsistent, and imprecise information about the requirements and the system's operating environment, and that each design step reduces incompleteness, removes inconsistencies, and improves precision, towards the specification of the system [5, 14, 25, 46, 20, 61, 56, 11, 52, 33, 17].

This important and general conceptualisation of the aim in RE is most clearly formulated in Zave & Jackson's seminal paper, "Four dark corners of requirements engineering" [61]. Their view, denoted ZJ hereafter, is echoed in some of the most influential research in the field, which both preceded and followed the said paper, including, for example, contributions from Boehm et al. [5, 3], van Lamsweerde et al. [14, 15, 57, 58, 56, 43], Mylopoulos et al. [45, 25, 11], Robinson et al. [52], Nuseibeh et al. [46, 30], to name some.

According to the ZJ view, in any concrete systems engineering project, RE is successfully completed when the following conditions are satisfied [61]:

1. "There is a set R of requirements. Each member of R has been validated (checked informally) as acceptable to the customer, and R as a whole has been validated as expressing all the customer's desires with respect to the software development project.

2. There is a set K of statements of domain knowledge. Each member of K has been validated (checked informally) as true of the environment.
3. There is a set SS of specifications. The members of S do not constrain the environment; they are not stated in terms of any unshared actions or state components; and they do not refer to the future.
4. A proof shows that $K, S \vdash R$. This proof ensures that an implementation of S will satisfy the requirements.
5. There is a proof that S and K are consistent. This ensures that the specification is internally consistent and consistent with the environment. Note that the two proofs together imply that S, K, and R are consistent with each other."

If the satisfaction of these conditions marks the end of RE in any systems engineering project, then we can give a compact formulation of the default problem that RE should solve, which we call the Default Requirements Problem hereafter:

Definition 1. Default Requirements Problem (DRP): Given a set R of requirements, and a set K of domain knowledge, **find** a specification S, such that S satisfies the following conditions:

1. There is a proof of R from K and S, written $K, S \vdash R$,
2. K and S are consistent, written $K, S \nvdash \bot$.

1.3 Issue: What if the System is an Adaptive System?

A system is an **Adaptive System (AS)** if it can detect differences between its design-time and run-time requirements and environment conditions, uses feedback mechanisms to analyse these changes and decide, with or without human input, how to adjust its behaviour as a result.

There is nothing in the DRP which makes it specific to a system class. This would suggest that the RP for AS is a subclass of DRP, in the sense that it is the DRP, with some additional properties that make it specific to the Adaptive Systems class.

It is important to know whether RPAS is a subclass of the DRP. According to Zave & Jackson, DRP "establish[es] minimum standards for what information should be represented in a requirements language" [61].

But this is not only important because of the interest in the design of languages for the representation of requirements, domain knowledge, and specifications of ASs.

More generally, if it is the valid conclusion, then there are existing RE tools (representation languages, methods, algorithms, etc.) that should be used in RE for AS, and the open question is how to specialise them to AS.

If it is not the valid conclusion, then the issue is to know which of the existing research is relevant for solving RPAS, both in RE and elsewhere, and what new research is needed. In both cases, discussing the validity of the conclusion above should provide relevant input for future research on RE for AS, and relate it to the default view of RE.

1.4 Contributions: Requirements Problem for Adaptive Systems and its Relationship to the Default Requirements Problem

This paper has three parts:

1. Part one runs from Section 2 to Section 5. It argues that the **Requirements Problem for Adaptive System (RPAS)** is *different* from the DRP, and that it is *not a subclass of* the DRP. This is argued in the following steps:
   a. The starting point, developed in Section 2, is the observation that DRP is the *minimal* RP, in the sense that if something is removed from it, there is no meaningful problem left to solve.
   b. Minimality suggests that there may be similarity between the DRP and every other RP, including the RPAS. The second step of the argument, developed in Section 3, is the observation that the DRP is not the superclass of all RPs, despite its minimality. This is argued by showing that, if we want to compare specifications when solving an RP, and we do it in order to choose the one that is somehow *the optimal one*, then that RP is not a subclass of the DRP.
   c. The third step of the argument, in Section 4, shows that to want the optimal specification as a solution to an RP, is not a new idea in RE. It has been studied in research on non-functional requirements. We argue that, once there are non-functional requirements in an RP, then this RP is not a subclass of the DRP.
   d. The fourth step of the argument, in Section 5, shows that optimality and non-functional requirements are central to Adaptive Systems engineering, and therefore, that RPAS is not a subclass of the DRP.
2. Part two proposes a general definition of the RPAS. This is done in four steps:
   a. Section 6 introduces new concepts, of problem and solution spaces, of criteria and parameters, and so on, for defining RPs in general. The new concepts are motivated by the discussion in part one of the paper.
   b. Section 7 connects the discussion of optimality to the new concepts introduced in Section 6.
   c. Section 8 introduces a new class of RPs, called Requirements Optimisation Problems, used to define the RPAS.
   d. Section 9 defines the RPAS.
3. Part three, in Sections 10–12, relates the RPAS to mathematical optimisation in general, to decision analysis in management science, and to expected utility theory in economics.

2 The Default Requirements Problem is a Minimal RP

DRP is a minimal RP, in the sense that if any of its parts is removed, the rest is not an interesting problem for RE, or no problem at all.

To see this, consider the following rewriting of the DRP. The only difference from Definition 1 is that there are now labels on parts of the problem statement. Labels are used as follows: to label the statement "it is raining" with the label **X**, we write **[**it is raining**: X]**.

Definition 2. Default Requirements Problem (DRP): Given [a set R of requirements**: R]**, and **[**a set K of domain knowledge**: K]**, **find [**a specification S**: S]**, such that **[**S satisfies the following conditions**: DR]**:

1. **[**There is a proof of R from K and S, written $K, S \vdash R$**: Satisfaction Condition]**,
2. **[**K and S are consistent, written $K, S \nvdash \bot$**: Consistency Condition]**.

There are labels on six parts of the problem statement. **R** refers to the set of requirements, **K** to the set of domain knowledge, and **S** to the specification. **DR** refers to the decision rule, that is, a rule stating what the thing to find, namely **S**, needs to satisfy, in order for it to be a solution to the problem. The Satisfaction Condition refers to the condition that there should be a proof of R from K and S, and the Consistency Condition to the consistency of K and S.

It should be fairly straightforward to notice that removing any one of the labelled parts leaves no problem at all, or no problem of interest to RE:

- If **R** is removed, then Satisfaction Condition has to go too, and this remains:

  > **Given [**a set K of domain knowledge**: K]**, **find [**a specification S**: S]**, such that **[**S satisfies the following condition**: DR]**: **[**K and S are consistent, written $K, S \nvdash \bot$.**: Consistency Condition]**

  Any S which is consistent with K is a solution to this problem, making this an uninteresting problem for RE, given that the any solution to this problem is designed independently from requirements.
- If **K** is removed, then this remains:

  > **Given [**a set R of requirements**: R]**, **find [**a specification S**: S]**, such that **[**S satisfies the following conditions**: DR]**:
  >
  > 1. **[**There is a proof of R from S, written $S \vdash R$**: Satisfaction Condition]**
  > 2. **[**R and S are consistent, written $R, S \nvdash \bot$.**: Consistency Condition]**

  Note that the Consistency Condition is changed above; another option is to remove the Consistency Condition altogether, rather than rewrite it so that R and S have to be consistent. In both cases, what remains is not a relevant problem, since it says that any specification, including those defined independently from the environment conditions, will be a solution, as long as it satisfies the two conditions above.
- If the Consistency Condition is removed, then every inconsistent specification becomes a solution. This happens if $\vdash$ is understood as the syntactic consequence relation of classical logic. This relation, then, satisfies satisfies the *ex falso quod libet* proof pattern, which is that anything follows from an inconsistent set of formulas. Here, it means that if $S \vdash \bot$, then $K, S \vdash R$, whatever the content of R and K.
- If the Satisfaction Condition is removed, the result is the same as removing **R**.

3 The Default Requirements Problem is not the Unique RP

The conclusion of this section will be that *the DRP is not the unique RP*, and therefore, that DRP is not the superclass of all RPs.

Section 3.1 explains what it means for an RP to be unique, and gives the main reason why it matters to know whether the DRP is unique. Section 3.2 lists properties that an RP can inherit from the DRP. Section 3.3 gives three RPs different from the DRP, and discusses what properties they inherit from the DRP, and in particular if they inherit all its properties. Section 3.4 defines the optimality property, which is implicit in the DRP. Section 3.5 argues that there are RPs which have a different optimality property than the DRP, and therefore, are not subclasses of the DRP.

3.1 Uniqueness Matters because of Inheritance

Uniqueness matters, because *if DRP is the unique RP, then all RPs have at least the same properties as DRP*. And if this is the case, then if we know how to solve DRP, this should help design ways to solve RPs in any other RP class.

Now, it may seem obvious that RE involves so many different problems, such as, for example, those related elicitation and negotiation, which look nothing like the DRP. And so, the conclusion from that already is that there is no unique RP.

However, any elicitation problem, negotiation problem, and so on, which tends to arise *when doing RE*, is really a problem that arises *only because a system design needs to be made or changed*. If elicitation is done without the aim of making or changing a system design, documented as a specification, then that problem is not an RE problem at all.

The unique RP, if there is one, would be the unique root of the taxonomy of RPs. This would be *the taxonomy of problems of designing systems*. Designing the system is the central problem for RE, one that gives the motivation for solving many other problems that arise when doing RE. These other problems, however difficult they may be, are the side effects that we have to deal with, because we are interested in designing systems.

3.2 What Can an RP Inherit from the DRP?

To see if an RP is a subclass of another, it is necessary to define what one can inherit from the other. If an RP X is a subclass of an RP Y, then X will inherit all the properties of Y.

Definition 3. Default Requirements Problem properties: The properties that an RP can inherit from DRP are motivated by the parts identified in Definition 2. These properties are as follows:

1. R Property: RP recognises that there is a category of information which describe conditions that are desired.
2. K Property: RP recognises that there is a category of information which describe conditions that hold independently from the system-to-be, and that the system-to-be has to live with.
3. S Property: RP recognises that there is a category of information which describe the system-to-be.
4. KSR Property: RP recognises that there are no categories of information other than those referred to by R, K, and S properties, which are relevant when solving DRP. In other words, all other kinds of information that may be useful are specialisations of those identified by the said properties.
5. Satisfaction Property: RP requires any solution to it to be such that there is proof that, if conditions described in K hold, and the system is implemented according to S, then conditions in R will be satisfied.
6. Consistency Property: RP requires that any solution to it be such that the conditions described in the K and S properties are not logically inconsistent.
7. Decision Rule Property: A description of a system-to-be is a solution to the RP if it satisfies the Consistency Property and the Satisfaction Property.

3.3 A Case of Complicated Inheritance

To illustrate inheritance between RPs, this section discusses three RPs.

3.3.1 RP1

The following is the first RP, named RP1.

> **RP1: Given [**a set G of goals**: G]**, and **[**a set K of domain knowledge**: K]**, **find [**a specification S**: S]**, such that **[**S satisfies the following conditions**: DR]**:
>
> 1. **[**There is a proof of goals in G from K and S, written $K, S \vdash G$**: Satisfaction Condition]**,
> 2. **[**K and S are consistent, written $K, S \nvdash \bot$**: Consistency Condition]**.

RP1 differs from DRP in that it has no mention of requirements R, but says that goals in G should be satisfied. However, the set of goals G has the exact same role in RP1 as requirements R has in DRP: both are used to capture information about desired conditions that the system-to-be should satisfy. RP1 is therefore a subclass of DRP, because it inherits all properties, including the R and KSR properties.

3.3.2 RP2

The second RP is as follows.

RP2: Given [a set R of requirements, partitioned onto mandatory requirements R^M and non-mandatory requirements R^{NM}**: R]**, and **[**a set K of domain knowledge**: K]**, **find [**a specification S**: S]**, such that **[**S satisfies the following conditions**: DR]**:

1. **[**There is a proof of mandatory requirements in $R^M \subseteq R$ from K and S, written $K,S \vdash R^M$**: Satisfaction Condition]**,
2. **[**K and S are consistent, written $K,S \nvdash \bot$**: Consistency Condition]**.

The difference is between RP2 and DRP is that the set of requirements R is partitioned onto mandatory and non-mandatory requirements in RP2. A solution therefore does not need to satisfy all requirements in R, but a subset thereof. RP2 is equivalent to DRP when all requirements in R are mandatory.

There are two ways of looking at inheritance between RP2 and DRP. One is to say that RP2 specialises the concept of requirement onto mandatory and non-mandatory requirement, and rewrites the Satisfaction Condition accordingly. The other is that RP2 is obtained by taking that R in DRP includes only mandatory requirements, and saying that there are other, non-mandatory requirements which remain outside DRP; in this second case, RP2 is the same problem as DRP, with the added set of non-mandatory requirements, which remain unrelated to the specification, and thereby not a factor that influences the design of the specification.

In both cases, RP2 looks like a subclass of DRP, because non-mandatory requirements play no role in the problem or the solution. RP2 thereby inherits all properties of DRP, and adds two properties: one is that any requirement is either mandatory or non-mandatory, and the other that a solution should satisfy all mandatory requirements.

Returning to the more general discussion, recall that in Section 2, it was argued that the DRP is minimal by looking at what remains after some part of it is removed.

There is a clear correspondence between parts of the DRP in Definition 2 and the properties in Definition 3.

Due to that correspondence, it follows that *if a part is removed, then what remains will fail to satisfy all the DRP properties*. Therefore, the set of properties in Definition 3 is minimal.

3.3.3 RP3

It was straightforward to determine what RP1 and RP2 inherited from DRP. RP3 is a more complicated case.

RP3: Given [a set R of requirements, partitioned onto mandatory requirements R^M and non-mandatory requirements R^{NM}**: R]** and **[**a set K of domain knowledge**: K]** , **find [**a specification S**: S]**, such that **[**S satisfies the following conditions**: DR]**:

1. **[**There is a proof of mandatory requirements in $R^M \subseteq R$ from K and S, written $K,S \vdash R^M$**: Satisfaction Condition]**,
2. **[**K and S are consistent, written $K,S \nvdash \bot$**: Consistency Condition]**,

3. **[**There is no other specification S' which satisfies both the Satisfaction Condition and the Consistency Condition, and in addition satisfies more of the non-mandatory requirements in R^{NM} than does S**: Optimality Condition.]**

RP3 partitions requirements onto mandatory and non-mandatory. In RP2, the non-mandatory requirements had no influence on which specification will be the solution. In RP3, the non-mandatory requirements appear in the Optimality Condition, that a specification has to satisfy in order to be the solution.

This suggests that the solution concept in RP3 is a subclass of the solution concept in RP2. In RP2, a solution is any specification which satisfies the Satisfaction Condition and the Consistency Condition, while in RP3, the specification also has to satisfy the Optimality Condition. In other words, the extension of the RP3 solution concept is a subset of the RP2 solution concept.

3.4 Optimality in the Default Requirements Problem

The Optimality Condition in RP3 is significantly different from the Satisfaction Condition and the Consistency Condition, because *the Satisfaction Condition and Consistency Condition are verified on a single specification, and it does not matter what other specifications there may be, while to verify if the Optimality Condition is satisfied, it is necessary to compare two or more specifications*.

It is therefore not possible to check if the Optimality Condition in RP3 is satisfied, when looking at one specification independently from others.

If we found a single specification S, which satisfies the Satisfaction Condition and the Consistency Condition, then in absence of at least one other specification S' with which to compare S in terms of how many non-mandatory requirements they satisfy, there will be no justification to the claim that S satisfies the Optimality Condition.

The DRP has its own notion of optimality, which is implicit in the Decision Rule Property.

To see it, suppose that there are three specifications S_1, S_2, and S_3, and that they all satisfy the Satisfaction Condition and the Consistency Condition for the same set of requirements R and the same domain knowledge K. Which of the three specifications is the optimal one?

The Decision Rule Property says that a specification is the solution if it satisfies the Satisfaction Condition and the Consistency Condition. As there is no other property that a specification needs to satisfy to be a solution, the only remaining conclusion is that *any specification that satisfies the Satisfaction Condition and the Consistency Condition is optimal*.

To make explicit the notion of optimality in DRP, we add the following property to the DRP.

Definition 4. Optimality Property for the DRP: RP recognizes that if there are more than one description of the system-to-be, all of which satisfy the Satisfaction and Consistency Properties, then they are all equally desirable.

This leads to the following revision of the properties that an RP can inherit from the DRP.

Definition 5. Default Requirements Problem properties (revised): The properties that an RP can inherit from the DRP are R, K, S, KSR, Satisfaction, and Consistency Properties from Definition 3, the Optimality Property from Definition 4, and the following property:

- Decision Rule Property: A description of a system-to-be is a solution to the RP if it satisfies the Consistency, Satisfaction, and Optimality Properties.

3.5 How are Optimality and Uniqueness related?

Optimality is important, because it is related to uniqueness in the following way:

In order to establish if a specification is the optimal specification and therefore the solution to the RP, it is necessary to compare specifications; any comparison of specifications requires having in the RP the information about comparison criteria; such information is not part of requirements, domain knowledge, or of the specification in the DRP, which violates the KSR Property, and therefore, the DRP is not unique.

To understand the above, suppose that in a concrete systems engineering project, we have the set R of requirements and K of domain knowledge. In RP1, we simply called every requirement a goal, so that *both the RP1 and the DRP have the same set of specifications, each of which can be a solution*.

It makes sense therefore to conclude that the RP1 is a subclass of DRP, or that they are equivalent RPs, because for any given R and K, *solving the RP1 or the DRP will involve choosing one solution from the same set of potential solutions*. Moreover, in absence of comparison criteria in either the DRP and RP1, we can choose any of the specifications as the solution.

For simplicity, the set of all specifications that we can choose one solution from, when solving an RP, will be called the **Solution Space** of that RP.

So the Solution Space is the same for RP1 and DRP, for the same given sets of requirements and domain knowledge.

Despite the similarities between RP2 and DRP, their Solution Spaces are different. In DRP, all requirements in a given set R of requirements must be satisfied. In RP2, that same set is partitioned onto mandatory R^M and non-mandatory R^{NM} requirements. It follows that the Solution Space of RP2 for given requirements R and domain knowledge K is the same as the Solution Space of DRP defined over those same requirements and domain knowledge *only if* $R^M = R$ *and* $R^{NM} = \emptyset$. Instead,

if any member of R is in R^{NM}, then it is by definition of RP2 not in R^{M}, and consequently, the Solution Space of RP2 will not include the same specifications that the Solution Space of the corresponding DRP would.

If an RP X is a subclass of another RP Y, then for the same given requirements and domain knowledge, every solution to X should also be a solution to Y. This is not the case in RP2.

The RP3 is not a subclass of the DRP for the same reason: as soon as some members of R are in R^{NM}, the Solution Spaces of RP3 and DRP will differ, as some specifications that can be solutions to the RP3 will fail to satisfy the non-mandatory requirements in R^{NM} and therefore cannot be in the Solution Space of the DRP defined over the same set of requirements R and the same domain knowledge K.

Now, it is fair to observe that RP2 is an odd RP, because there seems to be no role for non-mandatory requirements in it. But this same observation cannot be made for RP3, where non-mandatory requirements serve to define the criterion for the comparison of specifications in the Solution Space; that criterion is given in the Optimality Condition of RP3.

It is interesting to note that the conclusion we got here is counter-intuitive. To make RP3, we did three operations: (i) we specialised requirements onto mandatory and non-mandatory ones, (ii) revised the Satisfaction Condition, so that it reflects the idea that only mandatory requirements must be satisfied, and (iii) added the Optimality Condition. The three operations are non-controversial; we could not do (i) without also doing (ii), and doing (i) also made no sense without doing (iii).

The result is counter-intuitive, because the operation (i) looks simply like the specialisation of a concept that is already there in the DRP, and (iii) as just us asking that every solution satisfies an additional property, and so a specialisation of the DRP solution concept. If we look at each of these operations in isolation, they look like we are merely adding detail to what the DRP already had.

But together, the three operations resulted in a different problem to solve, because by solving RP3, we can get solutions to RP3 which are not solutions to DRP. Hence, RP3 cannot be a specialisation of the DRP.

4 Non-functional Requirements as Comparison Criteria

Non-functional requirements are an important source of comparison criteria. This section will argue that, *if* there are non-functional requirements in an RP, *and* we extract from them the criteria for the comparison of specifications, *and* we want to find the optimal specification according to those criteria, *then* the RP is not a subclass of the DRP.

Suppose that stakeholders give us non-functional requirements, also called quality requirements [6, 45]. For an ambulance system, an example can be the requirement that “ambulances should quickly arrive at incident locations”; denote this requirement `r1`. There is no universal criterion or industrial standard for just how much time amounts to “fast” in this requirement.

If we wanted to keep solving the DRP in the presence of non-functional requirements, we could do this by replacing each non-functional requirement by a variant, the satisfaction of which is binary. For example, "on average, ambulances should arrive to incident locations within 10 minutes"; denote this `r2`. And we could then have such K and S, that we can prove `r1` from them. All looks as if we are still solving the DRP.

But this transformation of `r1` misses the point. While `r1` looks like a requirement, its role in the RP is completely different than that of `r2`.

The non-functional requirement `r1` *serves as a criterion for the comparison of alternative specifications, because it states a preference relation*: by saying that ambulances should quickly arrive, it states that, when given two systems, one in which ambulances arrive slower, that one will – over this criterion only – be strictly less desirable than the system in which ambulances arrive faster. This is not at all the same as saying that ambulances should arrive within 10 min, since in that case, *any* system which satisfies this is good enough, while in the former case, only that system which achieves – among all those cosnidered – the shortest time for the ambulance to arrive, is good enough (again, if this criterion alone is considered, because when there are many criteria, perhaps some other criterion will have more importance).

Different specifications, each associated to a different design of the system-to-be, may result in different average time for an ambulance to arrive at the incident location. Some specifications will result in systems that will be faster, others slower.

There is, therefore no sense in placing `r2` in R, because how fast one specification is (or, to be precise, how fast we expect the resulting system to be), is relative between specifications. We cannot prove it from K, and S for *one* specification, because we have to take into account other, alternative, specifications.

Suppose that we do not accept the arguments above, and we want to do something to non-functional requirements in order to still keep solving the DRP. Here are some possible attempts to repair the situation:

1. We could rewrite `r1` as `r3`, with `r3` denoting the statement "specification S is the specification with the lowest time for an ambulance, on average, to arrive at an incident location", and put `r3` in R. But notice that `r3` is a rather odd requirement, since it *is about the relationship between different specifications*. This is also a practical problem, as it is not clear how we could prove `r3` from K and S, unless K and/or S also are about, or somehow mention other specifications.
2. We could say that `r1` is not a member of R, but stays outside K, S, and R, and that, to enable the comparison of specifications, we should add a variable, denote it q, for "average time to arrive at incident location", and that we should simulate, or estimate otherwise the value of q for each specification. If the various non-functional requirements result in a set Q of such variables, we could revise DRP by requesting that, for each specification S_i, this holds:

$$K, S_i \vdash R, Q_i \tag{1}$$

where Q_i is the set of value assignments to all variables in Q, produced by the specification S_i. We could then compare different specifications by the values they assign to variables, whereby we chose the variables to quantify level of satisfaction of non-functional requirements; for example, we could assume that there is a scale for time for `r1`, where values indicate the average time to arrive at an incident location.

3. We can replace syntactic consequence $\vdash$ with another relation; if we denote the new relation with $\mathrel{|\!\sim}$, we would need to define it in such a way that we have:

$$K, S_i \mathrel{|\!\sim} R \tag{2}$$

if and only if (i) R is provable from K and S_i, *and* (ii) there is no other specification S_j that better satisfies the non-functional requirements. Defining "better" requires us to define a way to aggregate, for each specification, the levels at which it satisfies all non-functional requirements, and then compare that aggregate score with those of all other specifications. The best specification would be the one with the highest score.

In each of the three approaches above, we made significant changes to DRP, in order to accommodate non-functional requirements. That is, we made new problems, different from the DRP:

- In the first approach, we revised the non-functional requirement `r1` as `r3` and placed `r3` in R. However, `r3` is about other specifications, yet DRP is about conditions that a single specification should satisfy, independently from others.
- In the second approach, we had to add Q_i, the assignment of values to measures of non-functional requirements. It is not a problem to have Q_i as a subset of R. However, if we want to choose the specification S_i which gives the most desirable Q_i, then we are not solving the DRP, because the part **DR** in DRP does not say that we should choose the most desirable specification which satisfies the Satisfaction Condition and the Consistency Condition. It actually says nothing about how desirable the specification ought to be, relative to other specifications which also satisfy these conditions.
- Finally, in the third approach, we replaced the provability condition with a relation $\mathrel{|\!\sim}$, which has to take into account the level to which non-functional requirements are satisfied in all considered specifications.

To summarise, while DRP is minimal, it is not unique. As soon as there is information about criteria for the comparison of specifications in the Solution Space *and we are interested in choosing the specification which best satisfies these non-functional requirements (whatever "best" may mean precisely in a specific project)*, then we are solving a different problem than DRP.

5 Optimality and Comparison are Central to Adaptive Systems

At this point, the important conclusion is that the DRP is not the superclass of RPs, in which we are interested in finding the optimal specification, and we have information that lets us compare specifications.

The claim in this section is that *RPAS is not a subclass of the DRP, because optimality and comparison play a central role in it*: namely, adaptation amounts to the switching between alternative ways of satisfying requirements, and therefore, each time the system needs to adapt, it needs to compare alternative ways of adapting, and choose the optimal one, among those that are available.

To further clarify this discussion, the rest of this section uses a trivial and hypothetical example, but sufficient to support these claims.

5.1 Adaptation as Switching

In this example, by *qualitative requirement*, we mean a requirement for which we say that it is either satisfied or not, not satisfied to some extent. By *quantitative variable requirement*, we mean a requirement that assigns a desirable value or range of values to a variable, which is not binary.

We have only two qualitative requirements `rA` and `rB` to satisfy. We know that we can satisfy `rA` by implementing one of five different functionalities, denoted `rAF1` to `rAF5`, and `rB` by implementing another five different functionalities, denoted `rBF1` to `rBF5`. For simplicity, let all ten functionalities be different and not related in terms of refinement or parthood, that is, they are neither more detailed variants of one another, nor parts of one another.

With two qualitative requirements and 5 functionalities satisfying each, there are 25 combinations of the 10 functionalities. But, some functionalities are not compatible. This means that we cannot make a system which includes them both. Some combinations of functionalities therefore do not give a specification which satisfies both `rA` and `rB`.

The part of the example introduced so far can be drawn as in Figure 1. There, filled circles are specifications, that is, combinations of functionalities that satisfy both `rA` and `rB`. Empty circles are incompatible combinations of functionalities, and since we cannot make a system that has those functionalities, these empty circles are not specifications.

If our problem was to find one combination where functionalities are compatible, and which satisfies both `rA` and `rB`, then this can be any one of the 20 specifications in Figure 1.

Consider adaptation to the failure of a functionality. If we were to design a system according to one specification among those in Figure 1, suppose that we chose, for example, S4, so that the system has functionalities `rAF4` and `rBF1`. If `rAF4` fails, the system would stop working as expected. If it were an Adaptive System, then it would be designed so as to switch to another functionality at run-time, for example,

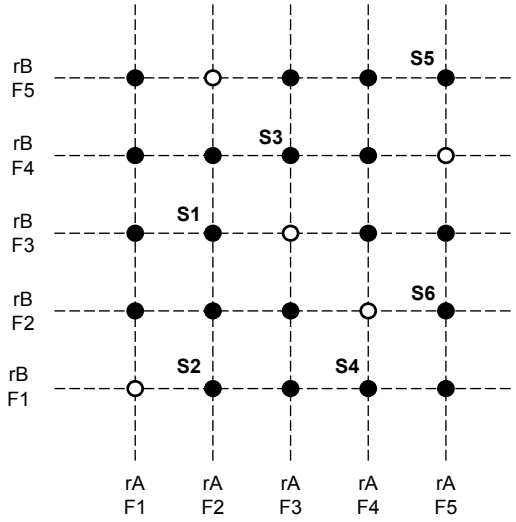

Fig. 1 `rA` and `rB` are two qualitative requirements that both need to be satisfied by a system. `rAF1` to `rAF5` are alternative functionalities that satisfy `rA`, while `rBF1` to `rBF5` are alternative functionalities that satisfy `rB`. Filled circles are combinations of functionalities that satisfy both `rA` and `rB`, and are thereby alternative specifications of a system; empty circles denote incompatible combinations of functionalities, and are not specifications.

from `rAF4` to `rAF2`. From the perspective of design-time, this amounts to a switch from the design given in specification S4, to that in S2. This is illustrated in Figure 2.

Now, assume that we have two quantitative variable requirements to satisfy, in addition to `rA` and `rB`. The first relates to scalability and the second to how the system compares to its competitors. Let `Var1` be the variable in the first, and Var2 in the second quantitative variable requirement. `Var1` can be "number of users that can simultaneously use the system" and Var2 "number of products available for purchase".

We prefer large to small values for both `Var1` and `Var2`, and we cannot accept values that are below some threshold. This is drawn in Figure 3, where T1 is the threshold for `Var1` and T2 for `Var2`, so that the shaded area shows all acceptable combinations of `Var1` and `Var2` values.

If the system were running according to S4, then it satisfies all four requirements, as its values over `Var1` and `Var2` are above their respective thresholds. If `rAF4` fails, the system would need to switch from S4 to either S3 or S5 in order to still satisfy all four requirements; if it switched to S2, it would satisfy `rA`, `rB`, and the requirement on `Var1`, but not the requirement on `Var2`.

As long as the system can switch from one specification to *any* other specification, provided that the latter satisfies all requirements and domain knowledge, then we can capture with the DRP the problem of designing that system's specification.

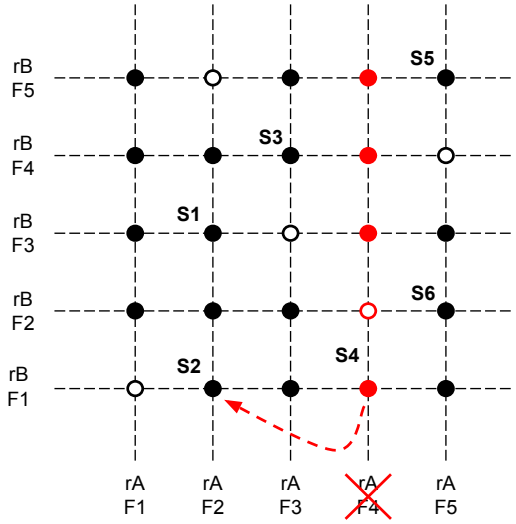

Fig. 2 System runs according to specification S4. Then, functionality `rAF4` fails, and the system activates functionality `rAF2` instead, switching thereby from specification S4 to specification S2.

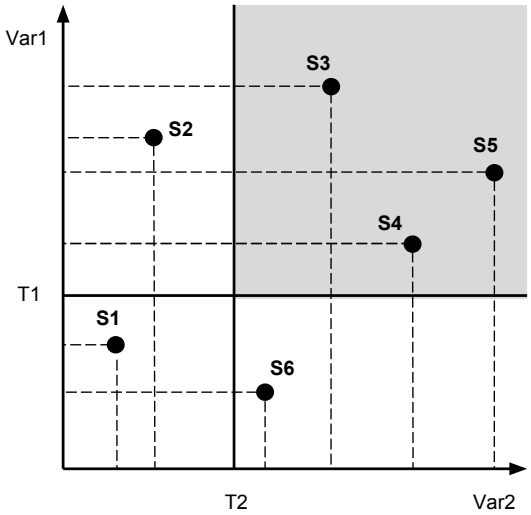

Fig. 3 Variables `Var1` and `Var2` quantify the level of satisfaction of two quantitative variable requirements. Hypothetical simulations of specifications S1 to S6 yield values show in the figure. T1 is the threshold value for `Var1`, and T2 for `Var2`.

5.2 Switching and Optimality

Switching to *any* specification fails to capture that *not all alternative specifications, that the system can switch to, are equally desirable*. The goal is to switch to the one that is optimal with regards to the requirements and domain knowledge, that the system is sensing.

This concern with whether the specification to switch to, is the optimal specification among those that we can switch to, clearly distinguishes RPAS from DRP.

We said that we prefer higher values of `Var1` and `Var2`, or in other words, we said that we have non-functional requirements which can be interpreted as suggesting that we prefer higher values of these variables. To make this more precise, we need to say which combinations of values we prefer over others. Since the region above the thresholds T1 and T2 is large, it is interesting to indicate (i) the shape of indifference curves in that region, and (ii) the direction where these indifference curves are over more desirable combinations of `Var1` and `Var2` values.

Figure 4 shows hypothetical indifference curves in the region above thresholds T1 and T2. Each indifference curve *is the set of Var1 and `Var2` value combinations that are equally preferred*. So every specification that has `Var1` and `Var2` values on the same indifference curve as S3 is equally preferred to S3. The arrow indicates the direction where `Var1` and `Var2` value combinations are preferred, so that any specification on the indifference curve with S5 is strictly preferred to any specification on the indifference curve with S3.

Having clarified with indifference curves what we mean by preference for higher `Var1` and `Var2` values, we now go back to Figure 1. There, we had 20 alternative specifications. If we want to see them as subsets of S in DRP, they are 20 alternative **configurations** of the same system. Moreover, we said that adaptation amounts to moving from one to another of these configurations, based on sensory input of the system, and feedback mechanisms that indicate what configuration to switch to. We also said that all this can be captured in DRP.

By adding the two quantitative variable requirements, with their `Var1` and `Var2`, we had restricted the set of acceptable configurations to some subset of the 20 shown in Figure 1.

Given several configurations, all above T1 and T2 thresholds, and all satisfying `rA` and `rB`, which one should we choose?

The answer is simple: given the indifference curves, we should choose any configuration which is on the the most desirable indifference curve. More generally, we want the system to switch, every time it needs to adapt, to the configuration that is the most desirable, among those that are feasible.

This is not to say that we cannot capture also this notion of optimality in DRP. For example, we can have as the only member of R in DRP, the proposition that there should be no feasible configuration which is on a more desirable indifference curve than the chosen configuration. We can, therefore see RPAS as a subclass of the DRP, although doing so seems rather odd, for there were no considerations of configurations, adaptation, preference, uncertainty, or optimality in defining the DRP.

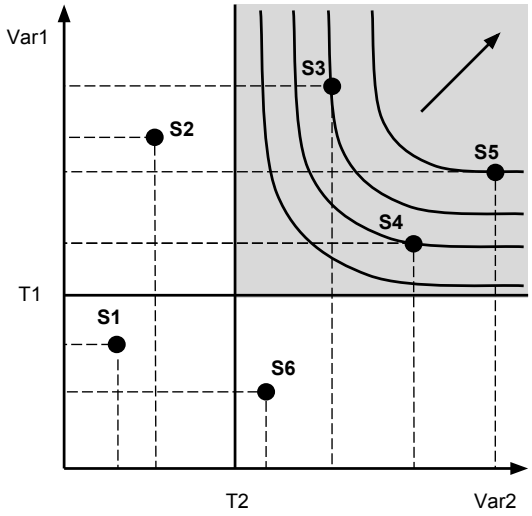

Fig. 4 Hypothetical indifference curves over value combinations of `Var1` and `Var2`. One indifference curve includes all `Var1` and `Var2` value combinations that are equally desirable. For example, any specification on the same indifference curve as S3 is equally desirable as S3. The arrow indicates the direction in which value combinations are more desirable. Therefore, S5 is preferred to S3, and S3 is preferred to S4.

6 Requirements Problem and Solution Spaces

We start with two simple definitions; they clarify what we will mean by the terms Specification and Solution in the rest of the paper.

Definition 6. Specification: A Specification is a description of a design of a system.

Definition 7. Solution: In a given RP, a Solution is a Specification such that, if we chose to commit to making the system according to that Specification, then we consider that we have solved that RP.

Another way to think about the Solution and Specification, is that the Specification is a candidate Solution in an RP that we are solving.

Hereafter, and in general in the paper, if we define a term, then we will capitalise it throughout. The capitalised term should be read as it was defined.

The conclusion that not all RPs are subclasses of the DRP, and that RPAS is also not a subclass of the DRP, means that there are many RPs that we may want to define and solve, and that each may have many Specifications, some or one of which is the optimal one, and thereby the Solution.

The goal now is to narrow down these ideas, and we will do this by defining the notions of Problem Space and Solution Space. If we agree on what these spaces are, then it will be easier to define the RPAS.

6.1 Problem Space

We call it the *Problem Space*, because it has some number of dimensions, and each dimension corresponds to something that we can evaluate a Specification for.

For example, if we have a DRP instance with one requirement $|R| = 1$ and no domain knowledge $|K| = 0$, then the Problem Space would be one-dimensional, a line, where each point refers to a level of satisfaction of the single requirement in R. If that requirement had a scale of satisfaction with 10 levels, then there would be exactly 10 points in the Problem Space. If it had two levels of satisfaction - satisfied or failed - then the Problem Space would have two points only. If the level of satisfaction was a real number between some maximum and minimum, then there would be an infinite number of points in the Problem Space.

In an RP which satisfies the R and the K properties, we can evaluate if a Specification satisfies a requirement, so that members of R would induce dimensions in the problem space. But we can also evaluate if the Specification satisfies or fails constraints from K, which is why members of K would also induce dimensions in the problem space.

To remain general, we need to avoid being constrained by the DRP. In particular, we want to be independent from the properties K and R, that is, from the categorization that the DRP imposes on criteria that Specifications are evaluated against. We do this by introducing the concept of Criterion in the definition of the Problem Space concept.

Definition 8. Problem Space: Set of points, where each coordinate of each point corresponds to the value of a Criterion.

Definition 9. Criterion: A variable, such that (i) we can establish its value for each Specification, and (ii) some of its values are more desirable than others, regardless of which Specification is being evaluated.

Various kinds of information used when solving an RP can produce Criteria for a Problem Space. The obvious example are requirements that are either satisfied or not. For each of those, we have one Criterion. non-functional requirements are more complicated, since it can be hard to find suitable variables to measure their level of satisfaction. Such variables would correspond to Criteria.

Criteria do not come from requirements only. Domain knowledge may indicate laws that the system-to-be should comply with, and each norm from the law may give a Criterion. Each of those Criteria would have two values, does comply and does not comply.

There is a nuance to keep in mind: some requirements, for example, may be refinements of other requirements, so that there can be relations between the value of different Criteria for the same Specification. For example, the value of a Specification on one Criterion may be fully determined by values of that Specification on other Criteria. The following example illustrates this.

In ambulance services, suppose that we have this statement: “Ambulances arrive at their incident locations”.

We will take this to be a requirement, and abbreviate it with `R(AmbArrive)`, where `AmbArrive` refers to the statement above, and `R` that this statement is a requirement.

We can make this requirement more detailed, by saying that it will be satisfied if these more specific requirements are satisfied: `R(IdentAmb)` for "Identify available ambulances", `R(ChooseAmb)` for "Choose ambulance to dispatch", `R(AssignAmb)` for "Assign ambulance to incident", `R(MobilizeAmb)` for "Dispatch the ambulance to the incident location", `R(ConfirmMob)` for "Confirm that the ambulance was dispatched".

In other words, we refined `R(AmbArrive)` onto five other requirements. As this means that if all five are satisfied, then `R(AmbArrive)` is satisfied, it also means that in the Problem Space:

- there is a Criterion for each of the six requirements above,
- each of these Criteria allows two values, satisfied or not satisfied (this is the case if we do not want to allow degrees of satisfaction for these requirements, but we see them as either satisfied or not),
- there is a function that ties the satisfaction of the five more specific requirements with the refined requirement, in that the latter is satisfied only if all five are satisfied. This means that the value of the Criterion corresponding to R(AmbArrive) is function of values for Criteria for the five other requirements.

In the example in Section 5.1 we had two requirements, `rA` and `rB`. We evaluated each as either satisfied or not. It follows that our Problem Space there has two Criteria, `r1` and `r2`. If we further assume that the satisfaction of one requirement is independent from the satisfaction of the other, then any Specification can correspond one of four points in the Problem Space. This is illustrated in Figure 5.

A more complicated Problem Space may involve non-functional requirements, which give Criteria that can take a real value from some range. Each Specification evaluates to one value of that Criterion, so that the number of positions that Specifications can take is infinite.

Figure 6 illustrates the Problem Space defined by one binary requirement and two variables quantifying the degree of satisfaction of non-functional requirements. There, the Problem Space amounts to two planes, one where a Specification has the coordinates $(rA\ not\ satisfied, x1, y1)$ and the other where a Specification's coordinates are $(rA\ satisfied, x2, y2)$; $x1$ and $x2$ are values of `Var1`, $y1$ and $y2$ of `Var2`.

6.2 Problem Instances

It is important to observe that Criteria define a Problem Space, not a single RP to solve.

If we have a Problem Space made of n Criteria, we can choose exactly one RP to solve by choosing one value of each Criterion. By choosing a point in the Problem Space, we have identified the exact requirements, domain knowledge, etc., that

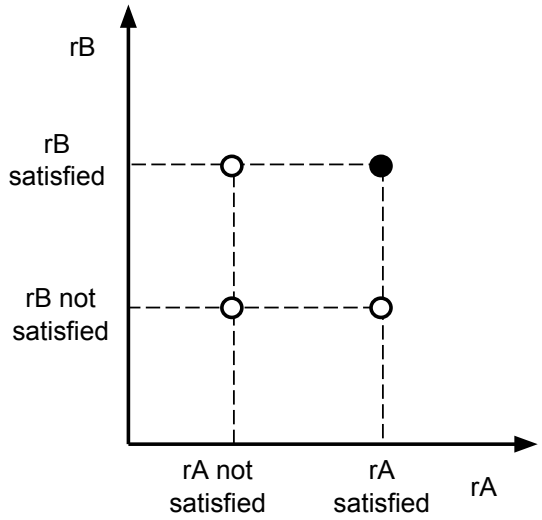

Fig. 5 Problem space defined by two requirements `rA` and `rB`, both with two levels of satisfaction. The satisfaction of one is independent from the satisfaction of the other. The black circle is a position in the Problem Space where both requirements are satisfied.

any Specification should satisfy. We capture this in our terminology by adding the following.

Definition 10. Problem Instance: A Problem Instance in a Problem Space is an assignment of a value to each Criterion in that Problem Space.

The above is important, because it suggests that, when doing RE, we can be solving one RP instance, or we may have the freedom to choose the Problem Instance instance to solve. This choice may be due to necessity, such as when it is not feasible to design the system in such a way, that it maps exactly to the Criteria values chosen in the Problem Instance.

For example, in Figure 6, the empty circle is a Problem Instance, and if choose to solve it, then we have decided to look for Specifications which do not satisfy the requirement `rA`. If we choose to solve the Problem Instance marked with the black circle, then we will be looking for Specifications that satisfy the requirement `rA`.

6.3 Solution Space

The Solution Space is made of dimensions that correspond to properties which we can *design into* the system.

For example, if we have in the Problem Space a Criterion that measures the response time of a server (or more abstractly, the responsiveness of the system), then

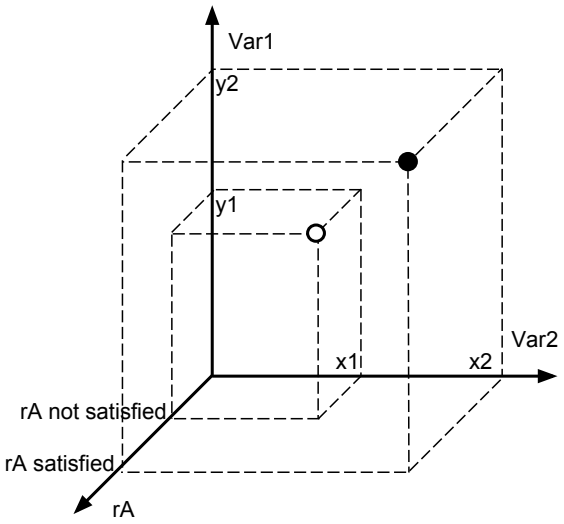

Fig. 6 Problem Space defined by the binary requirement `rA` and two non-functional requirements from Figures 1 and 3. Black circle is a point where `rA` is satisfied, while the empty circle is a point where `rA` fails.

in the Solution Space, we are interested in what we should build into the system, to make sure that it will achieve some value over that Criterion.

We introduce the following terms.

Definition 11. Solution Space: Set of points, where each coordinate of each point corresponds to the value of a Parameter.

Definition 12. Parameter: A variable, such that we can choose its value for each Specification, and this value is expected to influence the behaviour of the system-to-be in some predictable manner.

We chose the term Parameter, because its dictionary definition indicates it is a variable whose value *we choose*. This is very different from Criteria, since the idea for them is that we obtain their values through measurement or otherwise, rather than set or choose their values.

Parameter is a general notion, intended to be independent from what one chooses to call the fragments of a Specification. A two-valued Parameter can capture the common notion of functionality, as something that is either present or absent in the system-to-be. When it can take more values, a Parameter can capture the idea of parameterisable functionalities of the system-to-be; for example, that we *can or need to decide* the resolution of a screen in an operating system, the number of ambulances in a system delivering emergency services, and so on.

As for the Problem Space, there can be relations between values on various Parameters in the Solution Space; for example, a functionality `F1` may be decomposed

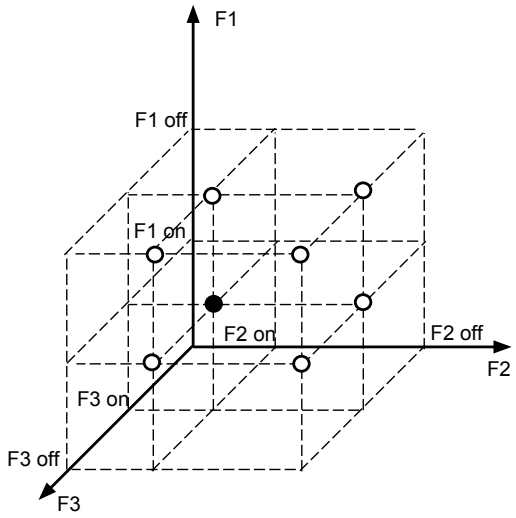

Fig. 7 Solution Space that has 8 points, defined by three Parameters, `F1`, `F2`, and `F3`, each of which can either be on (in a Specification of the system-to-be) or off (not in a Specification). On or off value of one Parameter is assumed independent from the on or off values of other Parameters. The black circle is a position where all three are included in the Specification.

into two different functionalities `F1a` and `F1b`, so that the presence or absence of `F1` depends on the presence or absence of both `F1a` and `F1b`.

Figure 7 illustrates a Solution Space defined by three Parameters, each of which can be either included in a Specification, or excluded from it. Figure 8 illustrates the Solution Space generated by two two-valued Parameters, and a Parameterwhose value can be any real number between 1 and 10.

6.4 Double Decision-Making

Problem Space and Solution Space concepts make it clear that RE may involve choosing both the Problem Instance to solve, and the Specification which solves it. For example, it may be that we change from one Problem Instance to another because of feasibility, while implementation cost could lead us to change from one Specification to another.

Problem-solving in RE involves this double and interdependent decision-making, of choosing the Problem Instance and choosing the Specification. If we were to design a process for problem-solving, then we would need to decide, for example, if we are going to first choose a Problem Instance, and then design the Specification to solve it, or if we would first choose a feasible Specification, and then see how to make the least changes to it to make sure it satisfies the Problem Instance closest

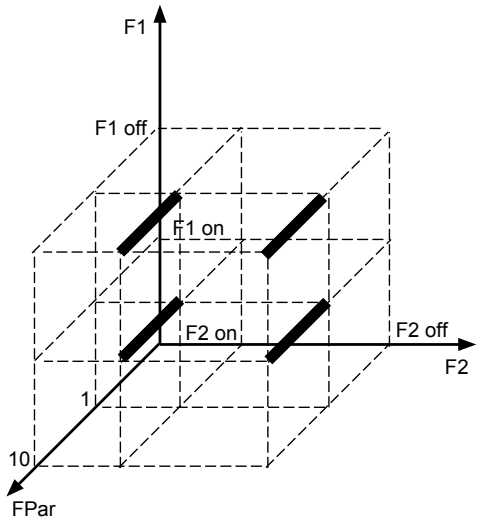

Fig. 8 Solution Space made of all points on the four thick black lines, defined by two two-valued Parameters `F1` and `F2`, and one Parameter `FPar`, which can take a real value between 1 and 10.

to the one we should solve. Or if we should do something else, such as distinguish mandatory from nice-to-have values of Criteria, and choose only among those Specifications, which satisfy all of the mandatory Criteria values.

For example, the DRP gives requirements and domain knowledge, so that we are given the Problem Instance, and we need to incrementally design a Specification, which should be the Solution to exactly that Problem Instance. This is because all members of R and all members of K must be satisfied, so that all values of all Criteria in this Problem Space are already decided.

This double decision making is an additional argument supporting the idea that DRP is not the root of a taxonomy of RPs. There is no particular reason why we must first choose one Problem Instance and then look for its Solution in the Solution Space. It can happen that we start from unrealistic requirements, and/or from conflicting requirements, and that, as we proceed in problem-solving, we have to revise the Problem Instance we are solving – that is, we need to move in the Problem Space, not only in the Solution Space. And this is the implicit assumption in RE research concerned with requirements inconsistency [30], conflicts [57], and obstacles to requirements satisfaction [58].

6.5 Relating the Problem Space and the Solution Space

There are two kinds of relations between the Specifications in the Solution Space and the Problem Instances in the Problem Space:

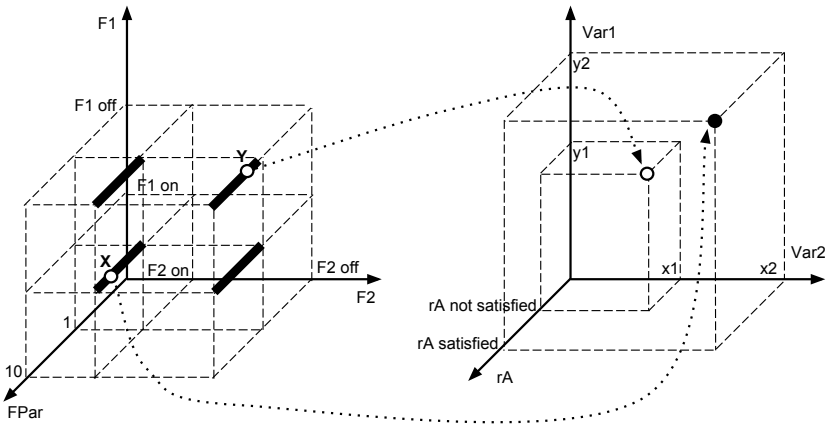

Fig. 9 Mapping Specifications from the Solution Space (left) to Problem Instances in the Problem Space (right). The dotted lines are instances of the Solve relation. Two Specifications X and Y are marked in the Solution Space. If we are looking to solve the Problem Instance where `rA` is satisfied, then Specification Y will not be appropriate, as it maps to the Problem Instance in which the requirement `rA` fails. The Problem Space shown involves non-functional requirements, whose degree of satisfaction is quantified with variables `Var1` and `Var2`.

1. The Solve relation, between one Specification and one Problem Instance, used to indicate that the former can be a Solution to the latter,
2. The Depend relation, when it makes the values of some Criteria (not necessarily all) depend on values of some Parameters.

6.5.1 The Solve Relation

By being a point in the Solution Space, the Specification can be seen as the synthesis of all our decisions, on what values to give to all Parameters.

Measurement or simulation of a Specification maps it to a Problem Instance in the Problem Space. (Measurement and simulation are not the only ways; there are others, such as relying on expert opinion, but this does matter much in this discussion.) We say that if the Specification X maps to the Problem Instance Y, then X solves Y. But since there can be many Specifications that can map to the same Problem Instance, we will say that X is the Solution of Y only if it is the one Specification chosen among all others that are considered during problem-solving.

We need a name for the relation between a Specification in the Solution Space and a Problem Instance in the Problem Space. This relation should exist between a Specification and a Problem Instance, if we believe that, when that Specification is implemented, measuring it over the Criteria in the Problem Space will result in exactly those Criteria values that the Problem Instance has.

Definition 13. Solve is a relation from one Specification in the Problem Space to one Problem Instance in the Problem Space. We say that Specification A Solves

Problem Instance B if and only if we believe that, if the system is made according to Specification A, and we measure the system according to the Criteria that define the Problem Space, then we will obtain values of these Criteriawhich correspond to the values that they have in Problem Instance B.

For a given RP, instances of the Solve relation are the result of problem-solving, as they can be found only after at least one Specification and one Problem Instance have been identified.

6.5.2 The Depend Relation

The Depend relation is between Parameters and/or Criteria, when their values are interdependent. It is not restricted to being between individual points in the Problem Space and the Solution Space. It can be used to represent that, for example, the value of a Parameter depends on the values of other Parameters and/or Criteria, that the value of a Criterion depends on values of other Criteria and/or Parameters, that the value of a Parameter has to be in some range, and so on.

A simple way to think about the Depend relation, is that, any function that relates the values of Parameters and/or Criteria is an instance of the Depend relation.

Definition 14. Depend: Given some variables, which may be Criteria and/or Parameters, if their values are interdependent, then there is an instance of the Depend relation between these variables.

Figure 10 gives an illustration of a function where the value of Criteria depend on the value of Parameters.

7 Optimal Specifications in Problem Spaces

Our definition of the Solution concept in Section 6 only says that the Solution is that Specification which we commit to, as the Specification which the system-to-be should implement.

In contrast, the discussion of optimality in Section 3 used the obvious premise that it is desirable, in general, to commit to the Specification which is somehow the "best" relative to those Specifications that are considered during problem-solving.

The goal now is to relate these two ideas, of optimality of, and commitment to a Specification, and do so using the concepts and relations introduced so far.

Relating commitment and optimality means using the following revised Solution definition.

Definition 15. Solution (replaces Definition 7): In a given RP, and thereby for its Problem Space and Solution Space, the Solution is the Optimal Specification.

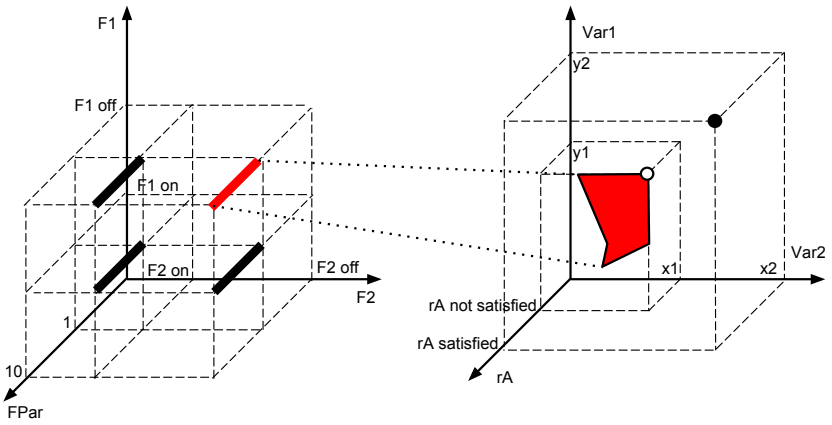

Fig. 10 If a point is chosen along the highlighted line in the Solution Space (left), then this results in a point on the highlighted area in the Problem Space (right). The figure illustrates the Depend relation between the values of the Parameters `Fpar`, `F1`, and `F2`, and the values of Criteria `rA`, `Var1`, and `Var2`.

The revision reflects the idea that relating commitment to optimality amounts to asking that we commit to the Specification which satisfies those properties that optimality imposes. This is different from the original Solution in Definition 7, where the only property of the chosen Specification is that we commit to it, regardless of how exactly it relates to other Specifications.

7.1 Preference and Utility in Problem Spaces

The revised Solution concept leads to the question of when a Specification is also an Optimal Specification, in the terminology of Problem Spaces and Solution Spaces.

To answer this, recall that Specifications in themselves are interesting only because they describe systems, which, *if* they are implemented accordingly, would achieve specific values over all of the Criteria in the Problem Space.[1] It is because it manages to map to some Criteria values that a Specification, or a set of Specifications is of any relevance in problem-solving.

The fact that we are designing Specifications because we are in fact interested in some values of Criteria, means that *whether a Specification is better than another is an issue that is solved not by looking only at the Solution Space, but at where a Specification maps to in the Problem Space*.

[1] As we cannot make all systems that implement all the Specifications, and then measure values of Criteria on systems themselves, we make the simplifying assumption that it is a Specification that maps to points in the Problem Space, not the system; this changes nothing in this discussion, other than pointing out that the mappings between the Solution Space and Problem Space will often simply be based on our experience, predictions, speculation, and such, not on actual measurement.

Therefore, whether a Specification is the Optimal Specification depends on *where it maps in the Problem Space*, since different Criteria values are not equally desirable. This was illustrated with indifference curves in Figure 4 earlier: we may be indifferent between some value combinations, which we can represent with an indifference curve, while different indifference curves reflect that we can evaluate some value combinations as strictly more desirable than others.

In the ideal and almost certainly infeasible case, we would have information about preference between every combination of values of independent Criteria. This would give us the indifference curves for these combinations, and also the corresponding utility function. In more realistic cases, we may be able to find a partial order preference relation, which compares some combinations, perhaps over a subset of Criteria in the Problem Space.

In any case, however, the important observation to make is that *in order to say which regions of the Problem Space are more desirable than others, it is necessary to have information about the relative desirability, that is, of preference of Criteria value combinations*.

As a brief digression, we recall here how the notions of *preference*, *utility*, and *indifference curves* are related in general, in economics. The term preference is used here to mean preference relation, the binary relation that indicates the relative desirability between two things; in this paper, it is the binary relation that compares the relative desirability of Criteria values, be they values of the same Criterion, or of different Criteria, or of value combinations of Criteria. Utility is a quantitative representation of information in a preference relation. If a preference relation, for example, over different values of a single Criterion, is transitive, complete, and continuous, then we can define a corresponding utility function which is continuous. An indifference curve is a set of different combinations of things compared in terms of preference, which are all equally preferred. A utility function reflecting that preference relation will return the same utility value for all members of that set. The same utility function gives many indifference curves, as there can be several sets of equally preferred combinations, such that combinations from different such sets are not equally preferred. As a final remark, utility is a generic notion, which can have different interpretations, and the specific meaning it will have depends on what one chooses to quantify desirability with.

7.2 Utility Representation with Criteria and Depend Relations

This preference information can be represented using the notions which were already introduced, namely, Criteria and the Depend relation. To add a utility function, add a Criterion whose values are read as utility values, and a Depend relation that specifies which other Criteria values determine the utility value. This is the illustrated in Figure 11, by adding a utility Criterion `U(Var1,Var2)` which gives the utility value for combinations of `Var1` and `Var2` Criteria.

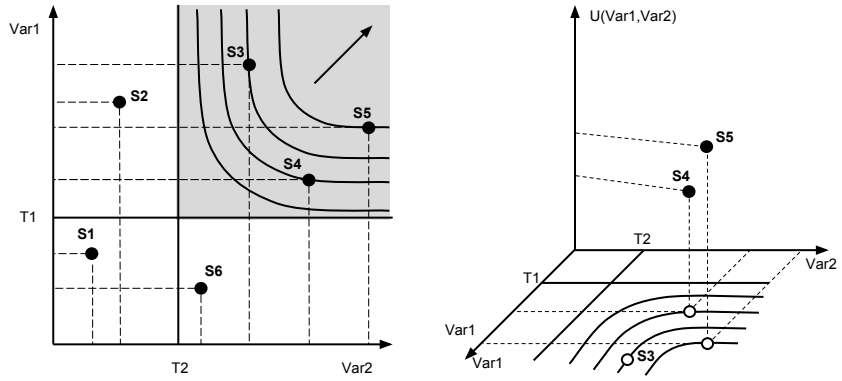

Fig. 11 The figure on the left shows a part of the Solution Space, borrowed from Figure 4. S3, S4, and S5 indicate mappings of three different Specifications to this part of the Problem Space. The three are above the threshold values `T1` and `T2`. Each is on a different indifference curve, indicating that they are not equally preferred. The part on the right adds a Criterion `U(Var1,Var2)` which gives the utility value of value combinations of `Var1` and `Var2`. Note that the figure on the right is a simplification, as the two points S4 and S5 are points on a three-dimensional surface; the shape of the indifference curve hints at the shape of that surface.

It is important to note that there can be different utility Criteria in the same Problem Space. This can be due to the fact that we may know preferences between value combinations of some Criteria, but not of others; or it may reflect differences in preferences between stakeholders, in which case we could have different utility Criteria for different stakeholders. Another important remark is that any definition of the dependence between the value of some Criteria and the value of other Criteria, be they utility Criteria or otherwise, are defined by a function that consequently is an instance of the Depend relation.

7.3 Decision Rules

Even in the trivial example introduced in Section 5.1, we can have many different preference orders, and thereby different utility functions. For example, there can be a preference order over values of `Var1`, another over those of `Var2`; if there are three stakeholders, A, B, and C, and they all have their own preferences over the values of these Criteria, then we might have nine preference orders in total; worse, we may not have a rule that tells us how to obtain preferences over combinations of `Var1` and `Var2`, from those that we have, over only values of `Var1` and only values of `Var2`.

As soon as there are two preference relations, it is necessary to explain how they are used *together* to compare Problem Instances to which Specifications map in the Problem Space.

This explanation is called the *Decision Rule*, and can take various forms.

In the example illustrated in Figure 11, we may want to do one of the following:

- Maximize the value of `U(Var1, Var2)`,
- Maximize the value of `Var1`,
- Maximize the value of `Var2`,
- And so on.

Each item gives the condition that a Specification should satisfy, in order to be the Optimal Specification. In the first item above, the Decision Rule means that the Optimal Specification is the Specification which maximizes the value of `U(Var1, Var2)`. In the second item, notice that the utility function is not given, but is implicit: namely, utility is independent from the value of `Val2`, and increases with the increase in `Val1`. The third item is the opposite case.

Definition 16. Decision Rule: A Decision Rule is the Criterion, such that in a given set of Specifications, the Specificationwhich has the highest value on that Criterion, is the Optimal Specification.

The Decision Rule is a special Criterion, as it will single out the Optimal Specification among those Specifications that we are considering. Its value may, and often will be the result of combining values of other Criteria, such as when `U(Var1,Var2)` is the Decision Rule.

8 Requirements Optimisation Problems

The Decision Rule definition, together with the notions we had introduced so far, introduces a new class of RPs, all of which include the Decision Rule defined above. They are called Requirements Optimisation Problems (ROPs).

Because they all include the Decision Rule, they are not a subclass of the DRP.

ROPs are important for the RPAS, as we will argue in Section 9 that the RPAS amounts to a set of ROPs, together with some additional constraints on that set.

This section first defines the ROP, and then illustrates how the DRP, if we only apply one change to it, becomes a subclass of the ROP.

8.1 Problem Statements

Solving an ROP involves doing design and doing decision-making.

Here, *doing design* means doing four tasks, and not necessarily in the sequence given below:

1. Constructing the Problem Space, by finding and defining Criteria, and defining Depend relations between these Criteria, when we know that there are correlations between their values, or have other reasons to believe that their values determine one another.

2. Constructing the Solution Space, by identifying and defining Parameters, and the Depend relations between the Parameters.
3. Defining Depend relations between Parameters and Criteria, so as to clarify how the choices of values of the former relate to the satisfaction of the Criteria.
4. Choosing the Decision Rule, and defining the Depend relation which makes its value depend on values of other Criteria in the Problem Space.

Doing decision-making here means identifying and committing to values of Parameters which, relative to any other combination of Parameter values, result in the highest value of the Decision Rule.

This separation between design and decision-making is introduced to highlight that there are two kinds of tasks in ROP problem-solving. The aim is *not* to suggest that they are necessarily done in sequence, that we first have to do all related to design, and then do the decision-making. The separation still fits incremental design, which, in this perspective, amounts to a sequence of activities, some of which are focused on design (such as, adding new Criteria and Parameters, choosing values thereof, etc.) and some of which are concerned with decision-making (trying to find values of Parameters) and can initiate the next design iteration (such as if we fail to find the Parameter values, which may lead to changing Parameters, Criteria, Depend relations, and so on).

The two problems are defined as follows.

Definition 17. Requirements Design Problem (RDP): Given the information about the stakeholders' expectations, and the information about the environment of the system-to-be, **define** (i) the Problem Space, (ii) the Solution Space, (iii) the Depend relations over Criteria in the Problem Space and the Parameters in the Solution Space, and (iv) the Decision Rule.

Definition 18. Requirements Optimisation Problem (ROP): Given (i) a Problem Space, (ii) a Solution Space, (iii) a set of Depend relations between Parameters and/or Criteria, (iv) a Decision Rule, and (v) a set of Parameters in the Solution Space whose values have not been set, called the Decision Set, **find** the values of Parameters in the Decision Set, such that there is no other combination of values of these same Parameters, which returns a higher value of the Decision Rule.

As Criteria and Parameters are variables, and Depend relations are functions over these variables, the ROP can be rewritten as follows:

$$\begin{aligned} &\textit{Maximise} \;\; f_0(x) \\ &\textit{subject to} \;\; f_i(x) = b_i, \textit{ for } i = 1,\ldots,n \end{aligned}$$

where:

- x is the Decision Set, that is, a set of Parameters in the Solution Space, whose values should be set,
- $f_0(x)$ is the Depend relation that returns the value of the Decision Rule in the Problem Space,
- Each $f_i(x) = b_i$, for $i = 1,\ldots,n$ is a Depend relation instance.

8.2 An Illustration: Converting the DRP into an ROP Subclass

The aim in this section is to illustrate how ROPs are related to the DRP. We do this by rewriting the DRP as a ROP subclass. That subclass will be called Revised Default Requirements Problem (RDRP).

Defining RDRP involves answering the following questions:

1. What are K, R, S of the DRP in the ROP?
2. What are the Depend relations in DRP?
3. What is the Decision Set in DRP?
4. What is the Decision Rule in DRP?

The DRP uses the syntactic consequence relation $\vdash$, leading us to assume that it is of classical, two-valued logic, so that members of K, S, and R can either take the value 0 or 1. Note that these values have a different reading depending on them being for members of K, S, or R: in R, "1" tends to read "satisfied", "achieved", etc.; in K, "1" can read "maintained", "satisfied", etc.; and in S, "1" can read "implemented", "configured", or otherwise, along these lines.

It follows that the RDRP has only integer binary variables.

In the terminology of Problem Spaces and Solution Spaces, variables that correspond to members of K and R are Criteria in RDRP, and define the Problem Space; variables that correspond to members of S are Parameters, and define the Solution Space.

The Depend relations in RDRP correspond to the relations between the members of K, S, and R. For example, if the value of a variable w_i depends on the value of other variables $y_1, \ldots, u_k$, then we define a Depend relation $w_i - f(y_1, \ldots, u_k) = 0$.

We will have such Depend relations, because if a requirement `rA` is, for example, refined by two other requirements `rA1` and `rA2`, then `rA` is satisfied if and only if both `rA1` and `rA2` are satisfied. In RDRP, this is captured by a Depend relation.

In the DRP, R is given and must be satisfied, which in the RDRP means that the values of all variables from R are set to 1, and they therefore cannot be in the Decision Set. K is also given, and since R and S should be consistent with it, we can also set all K variables to 1.

Variables from S in the DRP remain as the members of the Decision Set of the RDRP. If an S variable depends on the values of other S, K, or R variables, then we will exclude it from the Decision Set. It follows that the cardinality of the Decision Set does not need to equal cardinality of S.

All members of K and R are equally preferred, as it is equally important to satisfy every member of R, and to maintain every member of K. We concluded earlier that there are no means to compare specifications in the DRP. There is no information about preferences in the DRP.

But once we allow alternative refinement of same requirements, we have alternative Specifications to choose from, and we want the RDRP to reflect this. Suppose that we allow any member of K, R, S to be refined. This means that refining R gives us another set of requirements, which includes requirements that refine those in R. Let that set be $ref(R)$, and let $ref(K)$ and $ref(S)$ be for K and S, what $ref(R)$ is for R.

The resulting RDRP is to find a subset of $ref(S)$, such that, if all Parameters in that subset are set to 1, then all Criteria from K and R obtain the value 1. As there are is no preference information in the DRP, it is unclear what the Decision Rule should be.

The simplest choice we saw for the Decision Rule in the RDRP, is that it is equal to the sum of the members of $ref(S)$, meaning that we want to find the smallest subset of $ref(S)$ which manages to result in the assignment of the value 1 to all Criteria corresponding to members of K and R. So we want to maximise the following:

$$-\sum_{x_i \in ref(S)} x_i$$

In addition to the Depend relations for refinements, we need the following Depend relations to guarantee that the Solution must assign the value 1 to every Criterion from K and R:

$$\sum_{y_j \in R} y_j = |R|$$
$$\sum_{z_l \in K} z_l = |K|$$

Note that, since every x_i in $ref(S)$ is a binary variable, the solution to RDRP will be the smallest subset of $ref(S)$, which satisfies all Depend relations. This modifies the Decision Rule Property from the DRP; that property is neutral about the specifics of S, as long as it satisfies the Consistency and Satisfaction properties. This makes the RDRP a different problem than DRP, and this is not necessarily a relevant problem: the number of Parameters set to 1 in $ref(S)$ has, in itself, nothing to do with the quality expected from the system-to-be.

9 The Requirements Problem for Adaptive Systems

This section introduces the definition of the RPAS in the following steps. In the first step, in Section 9.1, we recall the key ideas in RE for ASs. In Section 9.2, we relate these ideas to the terminology of Problem Spaces and Solution Spaces, the RDP, and the ROP. This leads us in Section 9.3 to the definition of the design and decision-making problems that form the RPAS.

9.1 Key Ideas in Requirements Engineering for Adaptive Systems

In order to adapt to changes, the Adaptive System has to be capable of detecting changes; it can only respond and adapt to those changes that it can detect.

To detect changes, the Adaptive System has to gather data about events in its operating environment and about the functioning of its own components. At all times, and on the basis of these observations, the Adaptive System has to estimate the level to which it satisfies the stakeholders' requirements. If the levels of satisfaction are inadequate, the Adaptive System has to make changes to what it does in order to satisfy the requirements.

The AS changes its behaviour via *feedback loops*. A feedback loop defines the variables whose values need to be monitored; the values would be collected by sensors, or would be functions of variables whose values the sensors collect. When the values fall out of the predefined and allowed range, this triggers functionality in the AS, dedicated to make changes to the operation of the AS.

Capability to adapt requires a hierarchy of functionality in an AS. The lowest-level functionality interacts with the environment; the next level is functionality that enables feedback loops, which monitor signals from sensors that monitor the environment and the functionality at the lowest level; the second level is functionality that enables feedback loops that monitor and manipulate the feedback loops at the first level; and so on.

The above leads to key observations about the *run-time* of Adaptive System. (i) The level at which requirements are satisfied will vary, due to failure in the system and change in its environment. (ii) It is necessary to monitor the level of satisfaction of requirements, in order to know when the system needs to adapt. (iii) When the system adapts, it may have different ways of adapting, and each of these ways may have a different impact on requirements satisfaction levels. (iv) Whenever it needs to adapt, the system should adapt in the way that optimizes the levels of requirements satisfaction, relative to the newly observed failure of a component, or of a change in the environment.

The observations about run-time have important implications for the *design-time* of Adaptive Systems.

Due to observation (i), it may be too idealistic and impractical to think of requirements as being either satisfied or not, since this may lead to too many failed requirements, too often. It can be more practical, therefore, to define multivalued scales of requirements satisfaction, where failure equates to only some of many values. This is done through the **relaxation of requirements** [43, 59, 1], where the idea is to replace binary levels of satisfaction with, for example, continuous scales of satisfaction, or by letting the requirement be binary, but tracking the frequency of them being satisfied or failing, then using that frequency as the measure of the degree to which these requirements are satisfied.

Observation (ii) suggests that it is necessary, at design-time, to define the levels of requirements satisfaction that trigger adaptation. If the requirement has a binary satisfaction scale, being either satisfied or not, it may be relevant to define the minimal probability of observing its satisfaction; for example, in an ambulance dispatch

system, asking for the probability of at least 0.95, that an ambulance arrives to an incident location within 5 minutes of being dispatched to it. This would translate, at run-time, into looking at the frequency of incidents where the ambulance arrived 5 minutes or more, and triggering adaptation if that frequency is 5% or more of all incidents to which an ambulance was dispatched. If the requirement has a scale with many levels of satisfaction, then a threshold value has to be defined on that scale, such that, when the satisfaction is below threshold, the system needs to adapt. This has led to research on **awareness requirements** [55], which are used to define these thresholds for triggering adaptation.

At run-time, when awareness requirements are satisfied, feedback loops become active, and the system adapts. Because of observations (iii) and (iv), it is necessary to define at design-time the requirements that the system should satisfy when adapting. These are the so-called **evolution requirements** [54], and place constraints on how the system adapts. In the terminology of research on the RE for Adaptive Systems, evolution requirements place constraints on the range of **reconciliation tactics** [19, 18, 51] that the system may choose to apply, when adapting. The ambulance dispatch system could adapt to the failure in its component that records ambulance location, by requiring that control assistants who dispatch ambulances, record these locations manually, or by relying on the record of ambulance location by the part of the system which is located in each ambulance. An evolution requirement may indicate that control assistants should not manually record information, unless more than one of the automatic data recording components fails; this would exclude the second adaptation in the example.

9.2 Premises for the definition of the RPAS

The aim in this section is to relate the key ideas from existing research, to the terminology of Problem Spaces, Solution Spaces, the RDP, and the ROP.

9.2.1 Monitoring

The ability of an AS to adapt to changes requires that the system can detect changes. We introduce two terms, in order to talk about the extent of changes that the AS is designed to detect.

Definition 19. A **Monitored Variable** is a variable whose values the Adaptive System collects and whose changes of values can trigger adaptation.

Definition 20. The **Monitoring Scope** of an AS is the set of all of its Monitored Variables and, for each variable, the range of values that the Adaptive System can detect.

The Monitoring Scope describes what the AS is able to detect as change in its environment, and change in its own functionality. These changes are detected via

sensors. The variety and the specifics of the sensors that an AS has, determine its Monitoring Scope.

If something in the environment, or in the AS itself changes, but there are no Monitored Variables to reflect that change, then the AS will ignore it.

The Monitoring Scope reflects the *changes that were anticipated at design-time*. All other changes, which the Monitoring Scope cannot detect, remain as *unanticipated* changes. It is in this sense that we talk about *scope* in Monitoring Scope, as the scope of changes that have been predicted and considered as particularly relevant at design-time, regardless of how relevant they may actually prove to be at run-time.

The design of the Monitoring Scope involves choosing the Monitored Variables, based on the sensors that can be built into the AS, and the Depend relations between Monitored Variables and the Parameters in the Solution Space, and the Criteria in the Problem Space. Without these Depend relations, it is unclear why sensors would be used at all, or why and when the AS should adapt.

Monitoring the level of requirements satisfaction here means having Monitored Variables that are equal to some of the Criteria in the Problem Space. Monitoring the satisfaction of domain knowledge also means that the Monitoring Scope will include Monitored Variables that are equivalent to some Criteria, when these Criteria reflect domain knowledge. We may also have Monitored Variables that monitor Parameter values, as we want to know when failure happens, that is, when actual Parameter values are not those defined in the Solution.

9.2.2 Change

The Monitoring Scope is unlikely, in general, to be such that it enables the AS to detect all relevant changes in its functioning, its environment, and in the expectations of its stakeholders. This is the case as we cannot, at design-time, anticipate all that could potentially change, and to which the system should adapt.

Independently from the Monitoring Scope, there is what we call the *Change Scope*, denoting the variety of phenomena in, or outside the AS, which may occur, and to which the AS *may* need to adapt.

The Change Scope is not limited to phenomena that can cause the AS to fail to satisfy its design-time requirements, and/or to phenomena that put it at odds with its environment.

Stakeholders need to perceive the services that the AS delivers as being of high quality. People's evaluations of service quality reflect their own comparisons between expectations and experience with the service [48, 62, 49, 63, 39]. While design-time requirements are fixed, we may well have an AS that does achieve these requirements, but is perceived as being of low quality; stakeholders' expectations may have changed, enlarging the gap between what they expect, and their experience of the AS. The following definition of the Change Scope reflects this.

Definition 21. The **Change Scope** is the set of variables that describe the phenomena in the environment of the AS, and/or the system itself, are such that the values

of these variables can change independently from the system's operation, and these changes influence the stakeholders' perception of the quality of the AS.

At design-time and run-time, then, there can be many variables in the Change Scope that the system would ideally monitor. Their relevance may become apparent only after some phenomena occur at run-time and affect the stakeholders' perception of quality of the AS. In such cases, the engineers need to determine how to measure these phenomena, which sensors to use to collect measurements, and thereby add new variables to the Monitoring Scope.

In the terminology of the Monitoring Scope and the Change Scope, the design of feedback loops can be described as the task of identifying phenomena that can influence stakeholders' expectations, finding ways to measure them, adding these variables to the Change Scope. Next, it is necessary to determine how these variables are related to Criteria and Parameters. All such variables are candidates for becoming Monitored Variables.

It is unlikely that we can identify every variable in the Change Scope. Of those that we do manage to identify, we may also be able to make only some into Monitored Variables, due to, for example, there being no sensors that are capable to capture their values.

The conclusion that is important for the definition of the RPAS, is that the *design* part of the RPAS involves *finding and choosing variables in the Change Scope, that need to be made into Monitored Variables in the Monitoring Scope*, as it is unlikely that we can ensure that the Monitoring Scope fully covers the Change Scope.

9.2.3 Stability and Adaptation

We will use the term *event* to refer to any change of values of variables in the Change Scope; an event can be the result of a failure of a component of the AS, a drop in the level to which the AS satisfies a requirement, a change in the conditions in the operating environment of the AS, and so on.

The reason for having Monitored Variables in the first place, is because their changes of values result in changes of values of Criteria in the Problem Space and/or Parameters in the Solution Space.

The run-time of the AS is, then, a sequence of two kinds of time periods, called periods of *stability* and of *adaptation*.

Stability is any time period during the run-time of an AS, during which one of the following conditions holds:

- Values of Change Scope are not changing;
- Values of Change Scope variables are changing, but these variables are not Monitored Variables;
- Values of Change Scope variables are changing, and some or all of them are Monitored Variables. The changes result in new values for Criteria and/or Parameters. However, these changes are within some ranges that we judged tolerable at design-time.

As the values of Monitored Variables influence values of Criteria, they also influence the Problem Instance that the AS needs to be solving. As the values of Monitored Variables change, so does the Problem Instance to solve. At some time at the run-time, the AS should run according to the Optimal Specification that solves the Problem Instance applying at that time period.

Adaptation is the time period during the run-time of an AS, when the AS is running according to the Specification which is *not* the Optimal Specification for the Problem Instance that the AS should solve at that time.

To clarify this, suppose that T_1 denotes a time period, during which the Monitored Variables result in Criteria values which give one Problem Instance A, and the Optimal Specification X solves A. During T_1, the AS runs according to X.

Let T_2 be a time period which immediately follows T_1. T_2 starts with values of Monitored Variables that give the Problem Instance B. If $B \neq A$, and X does not solve B, it is necessary to find a new Specification Y which is the Optimal Specification for B.

Adaptation is the period during which the AS is searching for this new Specification, and changes behavior from running according to X, to run according to Y.

9.2.4 Relaxation

If the AS were to react to every change in Criteria and Parameter values, stability periods would be shorter, and more resources would be invested in adaptation.

Relaxation is used to allow the AS to run according to the same Specification in a broader range of conditions, and thereby potentially lengthen stability periods. We have three possible cases, depending on what changes for the AS, and relaxation can work in each case:

- If only Criteria values change, then the goal is no longer to solve the Problem Instance, denote it A, that was relevant in the last stability period, but a new Problem Instance, denote it B. If the Specification X was the Solution to A, and it is not the Optimal Specification for B, then adaptation would involve finding the Specification Y, which is the Optimal Specification, and thereby the Solution to B. Adaptation can be avoided, if we allow X to be the Solution to both A and B.
- If only Parameter values change, then the Problem Instance A to solve has not changed, but the AS is no longer running according to some Specification X from the last stability period, but according to a new Specification Y. There is no guarantee that Y is the Optimal Specification for A, but it can be the Optimal Specification for some other Problem Instance B. Adaptation would amount to finding a third Specification, which solves a Problem Instance C, whereby C is closer to A than B. To avoid adaptation in this case, relaxation would consist of allowing any one Specification from some set of Specifications to be the Solution to A.
- If both Criteria and Parameter values are changing, then adaptation will involve finding the Optimal Specification to the new Problem Instance. To avoid adap-

tation, relaxation would need to be such that, the new Problem Instance is considered sufficiently close to the one from the last stability period, and the new Specification as the Solution to the new Problem Instance.

In the cases above, and using the terminology of ROPs, adaptation amounts to the act of recomputing the Solution to a ROP, when the Problem Instance changes. Relaxation, in contrast, is the act of not triggering the computation of a new Solution to a new Problem Instance. At design-time, relaxation consists of changing the Criteria in the Problem Space, and changing Depend relations that link Parameter values to Criteria values. For example, suppose that we have formulated a ROP, and we want to relax it so as to allow more Specifications to be its Solution.

9.3 Problem Statements

RPAS is a double problem, one focused on design, the other on decision-making. They are defined as follows.

Definition 22. Requirements Design Problem for Adaptive Systems (RDPAS): **Given** the information about the stakeholders' expectations, the information about the environment of the system-to-be, and the predictions of changes to stakeholders' expectations and the environment, **define** (i) the sequence of ROPs that the AS is expected to solve, and (ii) the Monitoring Scope needed to detect some or all of the predicted changes.

Definition 23. Requirements Optimisation Problem for Adaptive Systems (ROPAS): **Given** a sequence of ROPs that the AS is expected to solve and the Monitoring Scope for the AS, **find** the set of Specifications and Evolution Requirements, such that, if the AS can run according to the Specifications, and while adapting satisfy the Evolution Requirements, then it will maximise the time that it runs according to the Optimal Specification in each stability period.

10 ROP and Mathematical Optimisation

Subclasses of the general ROP are made by restricting the properties of Criteria, Parameters, Depend relations, and the Decision Rule.

Restrictions can be intended to narrow down the informal reading of the Criteria, Parameters, and Depend relations, and thereby the informal interpretation of an ROP as of a problem statement meaningful from the perspective of the ZJ view of RE.

For example, the ZJ view distinguishes between requirements, domain knowledge, and specification. To capture this and define an ROP inspired by the ZJ view, the following would need to be done. All Criteria and Parameters need to be binary variables. The Criterion class should have two subclasses, requirement and domain knowledge. Specification should be the only subclass of the Parameter class.

Another kind of restrictions, independent from what one thinks RE is about, are on types of Criterion and Parameter variables, and the mathematical properties of Depend relations. They are interesting, because the general ROP is a subclass of the general mathematical optimisation problem, defined as follows.

Definition 24. Optimisation Problem [8]: A mathematical optimisation problem, or just an Optimisation Problem, has the following form:

$$\begin{aligned} &\textit{Minimise} \;\; f_0(x) \\ &\textit{subject to} \;\; f_i(x) \leq b_i, \textit{ for } i = 1, \ldots, n \end{aligned}$$

where the vector $x = (x_1, \ldots, x_n)$ is called the optimisation variable of the problem, the function $f_0(x)$ is called the objective function, the functions $f_1(x) \leq b_1, \ldots, f_m(x) \leq b_m$, are the constraint functions, and the constants $b_1, \ldots, b_m$ are the limits, or bounds, for the constraints.

Definition 25. Optimal Solution : The Optimal Solution to an Optimisation Problem is the vector x^* if it has the smallest objective value among all vectors that satisfy the constraints: that is, for any x' with $f_1(x') \leq b_1, \ldots, f_m(x') \leq b_m$, we have $f_0(x') \geq f_0(x^*)$.

ROP differs from the above in (i) having a different terminology, and (ii) equality in constraints. These differences still make the ROP a subclass of the General Optimisation Problem above. We can therefore reuse resolution techniques from mathematical optimisation [9, 8, 40, 47, 60, 31]. For example:

1. Depending on Criteria and Parameter variable types, we can have the following ROP subclasses:
   a. Binary ROP, where all Criteria and Parameter variables have binary value,
   b. Integer ROP, where all variables take an integer value. We can have this if we allow more than two levels of satisfaction of Criteria, and more than two configuration values for Parameters,
   c. Continuous ROP, if all Criteria and Parameters take real numbers as values,
   d. Mixed-integer ROP, if there are binary, integer, and continuous variables in the Decision Set.
2. Depend relation properties give another classification dimension, where we can distinguish:
   a. Linear Depend relations, where every relation is a linear function,
   b. Nonlinear Depend relations, where every relation is an arbitrary nonlinear function,
   c. General Depend relations, where some relations are linear, others nonlinear.

11 ROP and Decision Analysis

"Decision analysis is a logical procedure for the balancing of the factors that influence a decision. The procedure incorporates uncertainties, values and preferences in a basic structure that models the decision." [29]

The decision analysis procedure involves four steps [37, 38]. In step one, the aims are to specify the objectives to achieve by taking the decision, generate alternatives to choose from in order to achieve the objectives, and identify attributes used to measure the degree to which the objectives are achieved. Step two measures the uncertainty of the consequences of alternatives; the aim is to quantify uncertainty with a probability distribution of attribute values, with one probability distribution per alternative. Step three captures the relative desirability of value assignments to attributes. This gives a utility function, as a function over attributes. Step four ranks alternatives by their expected utility, by following the rule that the higher its expected utility, the more desirable the alternative.

The problem in decision analysis can be formulated as an Optimisation Problem, as follows.

Definition 26. Decision Analysis Optimisation Problem (DAOP) has the following form:

$$\textit{Maximise} \quad \sum_{i=1}^{i=n} (p(x_{j,i}) * U(x_{j,i}))$$

$$\textit{subject to} \quad \sum_{i=1}^{i=n} p(x_{j,i}) = 1, \textit{ for } i = 1, \ldots, m$$

where j denotes an alternative among m alternatives, i an attribute among n attributes, $x_{j,i}$ the value of the attribute i when alternative j is chosen, $p(x_{j,i})$ the probability of observing $x_{j,i}$ for i when choosing alternative j, and $U(x_{j,i})$ the utility of observing $x_{j,i}$ for i when choosing alternative j.

Definition 27. Optimal Solution to DAOP is an alternative j^* such that for any other alternative j', it is true that $\sum_{i=1}^{i=n}(p(x_{j',i}) * U(x_{j',i})) \leq \sum_{i=1}^{i=n}(p(x_{j^*,i}) * U(x_{j^*,i}))$.

The relationship between ROP and DAOP is that there are subclasses of ROP which are also subclasses of DAOP.

There are many such subclasses, and they depend on how ROP concepts are mapped to DAOP concepts. Here are the steps to follow, to make one such ROP subclass:

1. Let every alternative $j = 1, \ldots, m$ in the DAOP be a point in the Solution Space.
2. Let each attribute $i = 1, \ldots, n$ that is used to evaluate alternatives, be a Criterion in the Solution Space.
3. Add one Criterion to the Solution Space, and call it utility. Its values are utility values, interpreted informally in the same way utility values are in decision analysis. Note that the Solution Space now has $n + 1$ dimensions.

4. Add Depend relations, which map values of all non-utility Criteria, or combinations of these Criteria, to values of the utility Criterion. This is the same as saying, using terms common to RE, that different levels of satisfaction of requirements and domain assumptions that these non-utility Criteria amount to, result in different levels of utility. It is necessary to add these Depend relations, because in decision analysis, utility is the aggregate measure of the attributes used to evaluate alternatives, and these relations will reflect this. If such Relations were absent, it will look like the Decision Rule disregards all Criteria other than utility, or disregards all those Criteria whose values map to no utility value.
5. Constraints $\sum_{i=1}^{i=n} p(x_{j,i}) = 1$ for $j = 1,\ldots,m$ from DAOP should be carried over as Depend relations to the ROP.
6. The Decision Rule of the ROP should be equal to the Depend relation that corresponds to the objective function in the DAOP.

The resulting ROP subclass is a subclass of DAOP, in the sense that the only differences between DAOP and the ROP subclass are those of terminology and in the number of Depend relations, since it is necessary to add Depend relations that map non-utility Criteria values to the utility Criterion values.

12 ROP and Expected Utility Theory

"Expected utility models are concerned with choices among risky prospects whose outcomes may be either single or multidimensional. If we denote these various (say n) outcome vectors by x_i and denote the n associated probabilities by p_i such that the sum over $i = 1$ to $i = n$ of p_i equals 1, we then generally define an Expected Utility (EU) model as one which predicts or prescribes that people maximize the sum over $i = 1$ to $i = n$ of $F(p_i) * U(x_i)$. [...] Within this general EU model different variants exist depending on (1) how utility is measured, (2) what kind of probability transformations F(.) are allowed, and (3) how the outcomes x_i are measured." [Schoemaker:1982]

The DAOP introduced in the previous section is equivalent to the optimisation problem central to EU, so that the same remark as for DAOP applies: there is a subclass of SSOP which is also a subclass of the EU problem, and we can proceed to define that subclass in the way as for DAOP.

13 Conclusions

This paper argued that the Requirements Problem for Adaptive System is different from the DRP, and that it is not a subclass of the DRP. It then proposed a general definition of the RPAS. Finally, the paper related RPAS to mathematical optimisation in general, to decision analysis in management science, and to expected utility theory in economics.

Acknowledgements. The first author is funded by the Fonds de la Recherche Scientifique – FNRS (Brussels, Belgium). This work was supported in part by ERC advanced grant 267856, titled "Lucretius: Foundations for Software Evolution".

References

1. Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy goals for requirements-driven adaptation. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 125–134. IEEE, 2010.
2. Patrik Berander and Anneliese Andrews. Requirements prioritization. In *Engineering and managing software requirements*, pages 69–94. Springer, 2005.
3. Barry Boehm, Prasanta Bose, Ellis Horowitz, and Ming June Lee. Software requirements negotiation and renegotiation aids: A theory-w based spiral approach. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 243–243. IEEE, 1995.
4. Barry W Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, (1):4–21, 1984.
5. Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.
6. Barry W Boehm, John R Brown, and Myron Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605. IEEE Computer Society Press, 1976.
7. Barry W Boehm, Ray Madachy, Bert Steece, et al. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, 2000.
8. Stephen Poythress Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
9. Stephen Bradley, Arnoldo Hax, and Thomas Magnanti. *Applied mathematical programming*. Addison Wesley, 1977.
10. Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer, 2009.
11. Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Information systems*, 27(6):365–389, 2002.
12. Betty HC Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.
13. Jane Cleland-Huang, Raffaella Settimi, Chuan Duan, and Xuchang Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 135–144. IEEE, 2005.
14. Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993.
15. Robert Darimont and Axel Van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *ACM SIGSOFT Software Engineering Notes*, 21(6):179–190, 1996.
16. Alan Davis, Oscar Dieste, Ann Hickey, Natalia Juristo, and Ana María Moreno. Effectiveness of requirements elicitation techniques: Empirical results derived from a systematic review. In *Requirements Engineering, 14th IEEE International Conference*, pages 179–188. IEEE, 2006.
17. Neil A Ernst, Alexander Borgida, Ivan J Jureta, and John Mylopoulos. Agile requirements engineering via paraconsistent reasoning. *Information Systems*, 2013.
18. M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD*, page 50, Washington, DC, USA, 1998. IEEE Computer Society.

19. S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *IEEE Int. Req. Eng. Conf.*, pages 140–147, 1995.
20. Anthony CW Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *Software Engineering, IEEE Transactions on*, 20(8):569–578, 1994.
21. Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Pistore, Marco Roveri, and Paolo Traverso. Specifying and analyzing early requirements in tropos. *Requirements Engineering*, 9(2):132–150, 2004.
22. Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. In *Conceptual ModelingER 2002*, pages 167–181. Springer, 2003.
23. Joseph A Goguen and Charlotte Linde. Techniques for requirements elicitation. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 152–164. IEEE, 1993.
24. Orlena CZ Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101. IEEE, 1994.
25. Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: Rml revisited. In *Proceedings of the 16th international conference on Software engineering*, pages 135–147. IEEE Computer Society Press, 1994.
26. Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, 1996.
27. Andrea Herrmann and Maya Daneva. Requirements prioritization based on benefit and cost prediction: An agenda for future research. In *International Requirements Engineering, 2008. RE'08. 16th IEEE*, pages 125–134. IEEE, 2008.
28. Ann M Hickey and Alan M Davis. A unified model of requirements elicitation. *Journal of Management Information Systems*, 20(4):65–84, 2004.
29. Ronald A Howard. *Decision analysis: Applied decision theory*. Stanford Research Institute, 1966.
30. Anthony Hunter and Bashar Nuseibeh. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(4):335–367, 1998.
31. Michael Jünger, Thomas Liebling, Denis Naddef, George Nemhauser, William Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence Wolsey. *50 Years of Integer Programming 1958–2008*. Springer, Berlin, 2010.
32. Ivan Jureta, John Mylopoulos, and Stéphane Faulkner. Analysis of multi-party agreement in requirements validation. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 57–66. IEEE, 2009.
33. Ivan J Jureta, Alexander Borgida, Neil A Ernst, and John Mylopoulos. Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 115–124. IEEE, 2010.
34. Ivan J Jureta and Stéphane Faulkner. Clarifying goal models. In *Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling-Volume 83*, pages 139–144. Australian Computer Society, Inc., 2007.
35. Ivan J Jureta, John Mylopoulos, and Stephane Faulkner. Revisiting the core ontology and problem in requirements engineering. In *International Requirements Engineering, 2008. RE'08. 16th IEEE*, pages 71–80. IEEE, 2008.
36. Joachim Karlsson, Claes Wohlin, and Björn Regnell. An evaluation of methods for prioritizing software requirements. *Information and Software Technology*, 39(14):939–947, 1998.
37. Ralph L Keeney. Decision analysis: an overview. *Operations Research*, 30(5):803–838, 1982.
38. Ralph L Keeney. *Decisions with multiple objectives: preferences and value trade-offs*. Cambridge University Press, 1993.
39. William J Kettinger and Choong C Lee. Zones of tolerance: alternative scales for measuring information systems service quality. *MIS Quarterly*, 29(4):607–623, 2005.

40. Mark W Krentel. The complexity of optimization problems. *Journal of computer and system sciences*, 36(3):490–509, 1988.
41. John Krogstie, Odd Ivar Lindland, and Guttorm Sindre. Towards a deeper understanding of quality in requirements engineering. In *Advanced Information Systems Engineering*, pages 82–95. Springer, 1995.
42. Julio Cesar Sampaio do Prado Leite and Peter A Freeman. Requirements validation through viewpoint resolution. *Software Engineering, IEEE Transactions on*, 17(12):1253–1269, 1991.
43. Emmanuel Letier and Axel Van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 53–62. ACM, 2004.
44. Sotirios Liaskos, Sheila A McIlraith, Shirin Sohrabi, and John Mylopoulos. Integrating preferences into goal models for requirements engineering. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 135–144. IEEE, 2010.
45. John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering, IEEE Transactions on*, 18(6):483–497, 1992.
46. Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *Software Engineering, IEEE Transactions on*, 20(10):760–773, 1994.
47. Christos H Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *Journal of computer and system sciences*, 43(3):425–440, 1991.
48. Anantharanthan Parasuraman, Valarie A Zeithaml, and Leonard L Berry. A conceptual model of service quality and its implications for future research. *The Journal of Marketing*, pages 41–50, 1985.
49. Leyland F Pitt, Richard T Watson, and C Bruce Kavan. Service quality: a measure of information systems effectiveness. *MIS quarterly*, pages 173–187, 1995.
50. Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *Software Engineering, IEEE Transactions on*, 27(1):58–93, 2001.
51. W. N. Robinson. A requirements monitoring framework for enterprise systems. *Requir. Eng.*, 11(1):17–41, 2006.
52. William N Robinson, Suzanne D Pawlowski, and Vecheslav Volkov. Requirements interaction management. *ACM Computing Surveys (CSUR)*, 35(2):132–190, 2003.
53. Guttorm Sindre and Andreas L Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.
54. V. E. S. Souza, A. Lapouchnian, K. Angelopoulos, and J. Mylopoulos. Requirements-driven software evolution. *Computer Science - Research and Development*, pages 1–19, 2012.
55. V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness Requirements. In Rogério Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 133–161. Springer, 2013.
56. Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.
57. Axel Van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *Software Engineering, IEEE Transactions on*, 24(11):908–926, 1998.
58. Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, 2000.
59. Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty HC Cheng, and J-M Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 79–88. IEEE, 2009.
60. Mihalis Yannakakis. Expressing combinatorial optimization problems by linear programs. *Journal of Computer and System Sciences*, 43(3):441–466, 1991.
61. Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.

62. Valarie A Zeithaml, Leonard L Berry, and Ananthanarayanan Parasuraman. *Delivering Quality Service: Balancing Customer Perceptions and Expectations*. Free Press, New York, 1990.
63. Valarie A Zeithaml, Leonard L Berry, and Ananthanarayanan Parasuraman. The behavioral consequences of service quality. *The Journal of Marketing*, pages 31–46, 1996.