Explicit Alignment of Requirements and Architecture in Agile Development

Sabine Molenaar[1(✉)], Tjerk Spijkman[1,2], Fabiano Dalpiaz[1], and Sjaak Brinkkemper[1]

[1] Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands
{s.molenaar,f.dalpiaz,s.brinkkemper}@uu.nl
[2] fizor., Soest, The Netherlands
tjerk@fizor.io

Abstract. [**Context & Motivation**] Requirements and architectural components are designed concurrently, with the former guiding the latter, and the latter restricting the former. [**Question/problem**] Effective communication between requirements engineers and software architects is often experienced as problematic. [**Principal ideas/results**] We present the Requirements Engineering for Software Architecture (RE4SA) model with the intention to support the communication within the development team. In RE4SA, requirements are expressed as epic stories and user stories, which are linked to modules and features, respectively, as their architectural counterparts. Additionally, we provide metrics to measure the alignment between these concepts, and we also discuss how to use the model and the usefulness of the metrics by applying both to case studies. [**Contribution**] The RE4SA model employs widely adopted notations and allows for explicitly relating a system's requirements and architectural components, while the metrics make it possible to measure the alignment between requirements and architecture.

Keywords: Requirements Engineering · Software Architecture · User stories · Alignment · Metrics · Case study · Agile RE

1 Introduction

Requirements and design are interdependent and cannot be conducted as separate activities [28]. The Twin Peaks model describes how requirements and architecture are defined concurrently, yet being separate activities, with the former guiding the latter and the latter constraining the former [28]. Extending Nuseibeh's model, the Reciprocal Twin Peaks model [22] focuses on agile development and discusses why the synergy between requirements and architectural elements matters. Throughout the development process, one has to manage a continuous flow of requirements, as well as a continuously changing architecture.

Since software engineering is essentially a social activity among collaborating humans [36], communication within and across the various disciplines of software

N. Madhavji et al. (Eds.): REFSQ 2020, LNCS 12045, pp. 169–185, 2020.
https://doi.org/10.1007/978-3-030-44429-7_13

engineering (requirements analysis, architectural design, development, testing, etc.), is of primary importance [25]. In Requirements Engineering (RE), flawed communication within the development team is a common cause of project failure [15]. Furthermore, client wishes and needs change continuously, leading to volatile requirements that are hard to cope with [13,39].

While RE is still mostly rooted in a written set of requirements, the lack of proper documentation is a serious problem in Software Architecture (SA), which creates high risks of architectural drift and erosion, as well as increased costs and a decrease in software quality [34]. Inaccurate or missing documentation leads to difficult to maintain software. To make matters worse, the impact of new requirements are uncertain and reuse of components is nearly impossible [21].

The challenge that we tackle in this paper is how to keep RE and SA aligned in the context of agile development. While both Nuseibeh [28] and Lucassen [22] identified challenges and explained how RE and SA can support each other, they did not specify *how* to tackle them. What makes the problem hard is that a good solution should not increase stakeholders' workload or costs, in line with the principles of ubiquitous traceability [9]. Furthermore, Cleland-Huang *et al.* [10] identified seven challenges concerning the Twin Peaks model, of which we aspire to address five: lack of in-depth communication between requirements analysts and architects, lack of requirements/architectural knowledge, lack of architectural visualization and explicit traceability between the two domains.

As a solution, we present explicit concepts and relationships that link functional requirements and functional architectural components in order to achieve alignment, among other purposes. While this solution requires some upfront work, aimed at creating or recovering the architecture and linking the requirements, we expect it to decrease rework in the subsequent development phase. Furthermore, to minimize the extra effort, we make use of notations that are widely adopted in agile development and in software architecture. Specifically, we make the following contributions:

- We present a refined version of the RE4SA model [32], which includes notations and relationships for linking RE and SA in agile development;
- We introduce metrics that allow quantifying the relationship between the two domains. While meant for RE4SA, the metrics can be applied more in general to other notations for expressing requirements and architectures;
- We report on two case studies that apply RE4SA for the purpose of architecture discovery and architecture recovery, respectively.

The rest of the paper is structured as follows. Section 2 discusses background work. In Sect. 3 we present the RE4SA model, followed by the alignment metrics in Sect. 4. Section 5 illustrates how the model and its metrics can be applied in practice, using two case studies. Limitations, expected benefits and future work are discussed in Sect. 6, followed by the conclusion in Sect. 7.

2 Background

The rise of agile development created new challenges for the RE and SA disciplines. Requirements documentation changed from long, detailed specifications to less detailed documentation and increased face-to-face contact [8]. The most common notation for requirements in agile development is user stories, a concise notation that captures only the essential elements of a requirement [23]. Regarding the SA discipline, agile practices require the incremental, step-wise construction of a product's functionality, which calls for modular architectures that require minimal coordination with other modules and are easy to extend [12]. This dynamic context is the one within which this paper is positioned.

Keeping software artifacts aligned falls under the umbrella term of software traceability [9], which includes techniques for establishing and maintaining trace links between different artifacts like requirements, architecture, code, and tests. Among the open challenges that pertain to our work, *ubiquitous traceability* [17] is especially important, as it stresses the need of tools and techniques that minimize the required human effort to create and keep the trace links up to date.

Many automated tools exist for the automated establishment of trace links. Trace Analyzer [14] uses certain or hypothesized dependencies between artifacts and common ground and then considers nodes that contain overlapping common ground to establish a trace link. The common ground they use, however, is source code, which is unusable when the system is still under design. Zhang *et al.* [38] use an ontology-based approach to recover trace links, but only link the source code to documentation. Traceability links have also been explored in agile development, with a focus on establishing links between commits and issues [30].

The systematic mapping by Borg *et al.* [4] shows that the most frequently studied links in information retrieval-based traceability are the links between requirements and between requirements and source code. Other popular links are between requirements and tests, and other artifacts and code. Linking requirements and architectures is a less studied topic.

Tang *et al.* [33] study the creation of traces between requirements and architecture. They provide an ontology for annotating manually specifications and architectural artifacts, which are then documented in a semantic wiki. This wiki shows which architectural design outcome realizes which requirement, which decisions have been made, and the links to quality requirements.

Rempel and Mäder [31] are among the first ones to propose traceability metrics in the context of agile development. They propose graph-based metrics that link requirements and test cases. Numerous researchers in the field of software maintenance proposed metrics, starting from the seminal work by Pfleeger and Bohner [29]. Our work, however, focuses solely on metrics between requirements and architectures in the context of agile development for software products.

Recently, Murugesan *et al.* [27] presented a hierarchical reference model to capture the relationship between requirements and architecture. Their goals are similar to those of this research, but they focused on *technical* architectures.

Our work, instead, investigates *functional* architectures and suggests the use of specific artifacts to formulate more specific guidelines, as opposed to a generally applicable requirement-to-component connection model.

3 The RE4SA Model

To facilitate good communication within the development team and support consistency, we propose the Requirements Engineering for Software Architecture (RE4SA) model. Figure 1 shows the four core concepts of the RE4SA model, and an example for each of the concepts from a case study [32]. RE4SA was assembled on the basis of tight collaboration with industrial partners in the software products domain and combines artifacts that we often found employed in their agile practices [24,32]. Like the Twin Peaks model, RE4SA links the RE and SA domains. More specifically, it relates Epic Stories (ESs) [24] and User Stories (USs) [11] in the requirements domain, and modules and features from the functional architecture model [7]. The problem space, which describes the intended behavior through requirements, is related to the solution space that defines how such intended behavior is implemented, i.e., how the requirements are satisfied [2]. Note that the model is only concerned with horizontal traceability [18].

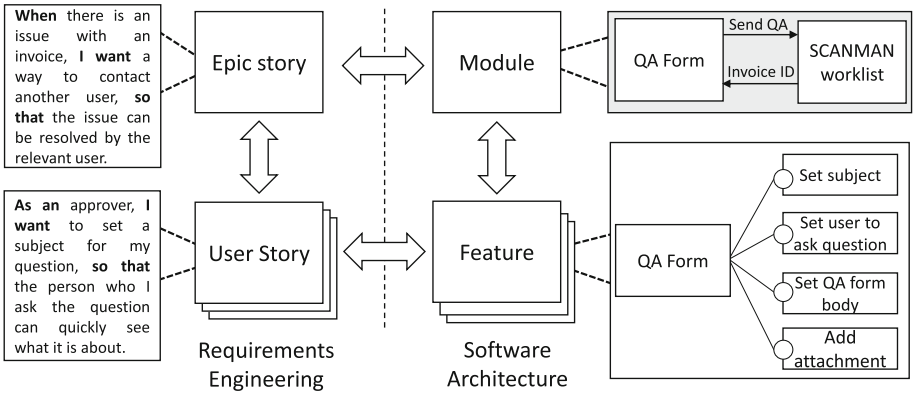

Fig. 1. The Requirements Engineering for Software Architecture (RE4SA) model.

3.1 Representing Requirements and Architecture

The concepts that are part of the RE4SA model encompass notations that are highly adopted in the industry, in an attempt to minimize the need for change and training of professionals. USs, for example, are often found to be among the requirements documents used in agile methods [20], and a US describes a requirement for one feature [23]. Features are often represented using feature

diagrams, a graphical language for organizing features hierarchically [19]. By focusing on the details, USs and features make it hard for the stakeholders to obtain an overview of the system that is necessary for clear and easy communication within the development team, thereby calling for a higher level of abstraction.

In practice, USs are grouped together using themes, epics or 'large USs' [35]. However, themes and epics tend to consist of one or a few words and thus lack the rationale that justifies why a requirement should be satisfied by the system [37]. Therefore, we propose the use of ESs [24], which make use of a clear template including both a motivation aspect and an expected outcome. From the architectural standpoint, we take the notion of 'module' from the functional architecture framework [7] as a grouping of features, that also allows for the visualization of usage scenarios through information flows [5].

3.2 Relationships Between the RE4SA Concepts

The RE4SA model supports the establishment of relationships between the four concepts in two ways: (i) Architecture Discovery (AD) is a top-down process that takes the requirements as input in order to create an architecture; (ii) Architecture Recovery (AR) is a bottom-up process that extracts the architecture from an implemented system [1]; then, the architectural components can be linked to the requirements. Figure 2 illustrates the four types of relationships between the concepts of RE4SA. The solid arrows indicate relationships in an AD process, while dashed arrows indicate an AR process. Furthermore, the relationships can be classified depending on whether they affect the *granularity* of the specification (refinement and abstraction) or they support the *alignment* between requirements and architecture (allocation and satisfaction).

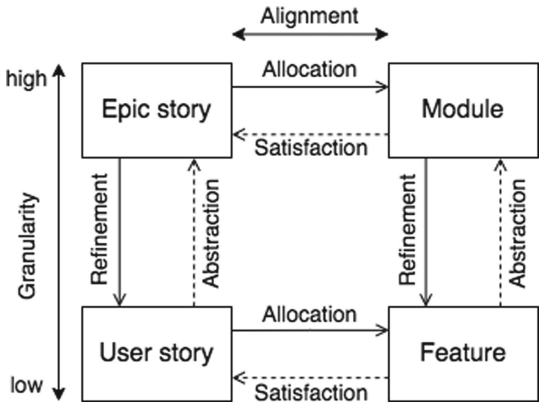

Fig. 2. Relationships between the RE4SA concepts.

Refinement. According to the SWEBOK guide "*decomposition centers on identifying the major software functions and then elaborating and refining them in a hierarchical top-down manner*" [6]. In an AD process, the major functions are described first, in ESs and modules, and subsequently refined into more specific functions and descriptions (here, in USs and features).

Abstraction. "*[. . .] refers to both the process and result of generalization by reducing the information of a concept, a problem, or an observable phenomenon so that one can focus on the "big picture"*" [6]. USs are grouped together using ESs, while features are bundled together based on similar functionality and placed in modules. The groupings of USs and features differ in the functionality they describe and the functionality they provide, respectively. The process of placing these sets of USs and features in ESs and modules we refer to as abstraction.

Allocation. The process of relating requirements to architectural components is "*the assignment to architecture components responsible for satisfying the requirements*" [6]. Since both requirements and architectural components exist on two levels of granularity, this relationship is included on both levels.

Satisfaction. The SWEBOK guide states that "*the process of analyzing and elaborating the requirements demands that the architecture/design components that will be responsible for satisfying the requirements be identified*" [6]. Therefore, we refer to this relationship from architectural components to requirements as satisfaction.

Since this paper investigates requirements-architecture alignment, we leave the study of refinement and abstraction to future research.

3.3 Architecture Discovery and Architecture Recovery

The AD process (solid arrows in Fig. 2) aims to design an intended architecture based on the requirements. It is advisable to start at the highest level of granularity, for the collection of ESs describe the functionality of the entire system, while USs specify the details of how such a high-level functionality is to be delivered. Once the requirements have been defined, they can be allocated to architectural components. W e suggest starting at the highest level: ESs are allocated to modules, then USs to features within the identified modules. Finally, it is useful to check if the features included in the software architecture are all represented in the requirements set. Features that cannot be linked to a requirement can indicate missing requirements or unnecessary features.

The goal of an AR process (dashed arrows in Fig. 2), instead, is to recover the implemented architecture from the system, using available documentation, such as source code and a run-time version of the system, and linking the recovered components to requirements. We suggest starting at the lowest level of granularity, and documenting the identified elements in a feature diagram. Different modules can then be defined to group the features.

Then, the architectural components can be linked to requirements by creating satisfaction links. We recommend starting at the highest level of granularity: the

ES-module alignment. If these relationships are established first, it should be easier to identify which feature satisfies which US, for the USs are abstracted to ESs. Optionally, missing ESs or USs can be formulated, if the module or feature they will be allocated to is still relevant and/or required. On the other hand, ESs or USs that cannot be allocated to an architectural component need to be assessed. If the functionality the requirement describes is not required or desired, the requirement can be removed. If the opposite is true, the implementation of the feature(s) that would satisfy the requirement can be added to the backlog.

4 Alignment Metrics

We introduce metrics that allow for quantitative investigation of the relationship between requirements and architecture through the lenses of the RE4SA model. To do so, we present the necessary formal framework the metrics build on. We use numbered definitions only for the core concepts of our framework.

Let $R = \{r_1, r_2, \ldots, r_n\}$ be a collection of requirements and $C = \{c_1, \ldots, c_m\}$ be a collection of architectural components. In the RE4SA model, a requirement can be either an Epic Story (ES) or a User Story (US), while a component can be either a module or a feature.

Since a requirement can denote multiple needs (e.g., using the conjunction 'and'), we introduce the function $needs : R \to 2^C$ that maps a requirement r to the needs it expresses. Formally, given a set of needs N, we have that for any $r \in R$, $needs(r) = \{n \in N.\ requested\_by(n, r)\}$, where $requested\_by(n, r)$ is true when n is expressed in the text of requirement r. In this paper, the identification of the needs that are requested by a requirement is left to human analysis.

We can now define the set $N_R = \bigcup_{r \in R} needs(r)$ as the collection of needs that are requested by individual requirements in the collection R.

Definition 1 (Alignment matrix). *A matrix A of size $|N_R| \times |C|$ such that $a_{ij} = 1$ if and only if the need $n_i \in N_R$ matches the component $c_j \in C$. Formally,*

$$a_{ij} = \begin{cases} 1, & \text{if } matches(n_i, c_j) \\ 0, & \text{otherwise.} \end{cases}$$

The alignment matrix is a key element of our framework that can be used to explore the mutual relationship between requirements and components. Based on the matrix, we define the function $allocation : R \to 2^C$ that returns the set of components that match the needs in a requirement. Formally, $allocation(r) = \bigcup_{n_i \in needs(r)} \{c_j.\ a_{ij} = 1\}$. Conversely, we define a function $satisfaction : C \to 2^R$ that returns all the requirements with needs matching a given component. Formally, $satisfaction(c_j) = \bigcup_{r \in R} \{n_i.\ a_{ij} = 1 \wedge n_i \in needs(r)\}$.

Based on the allocation function, we can partition the set of requirements into four non-disjoint subsets: $R = R_{not} \cup R_{under} \cup R_{exact} \cup R_{multi}$, defined as follows:

- $R_{not} = \{r \in R.\ allocation(r) = \emptyset\}$

- $R_{under} = \{r \in R.\ 0 < |allocation(r)| \wedge \exists n_i \in needs(r).\ (\sum_j a_{ij}) = 0\}$
- $R_{exact} = \{r \in R.\ \forall n_i \in needs(r).\ (\sum_j a_{ij}) = 1\}$
- $R_{multi} = \{r \in R.\ \exists n_i \in needs(r).\ (\sum_j a_{ij}) > 1\}$.

R_{not} is the set of requirements that are not allocated, R_{under} are those requirements with some but not all allocated needs, R_{exact} are those requirements with each need allocated to exactly one component, and R_{multi} are those requirements having at least one need allocated to multiple components. The four sets are not disjoint. For example, a requirement requesting needs n_1 and n_2, with n_1 matching components c_1 and c_2 and with n_2 matching no components would be both multi-allocated (because of n_1) and under-allocated (because of n_2).

Definition 2 (Allocation degrees). *The partitioning of R into R_{not}, R_{under}, etc. can be used to define metrics on the allocation degree of a set of requirements. We introduce three degrees, each in the $[0, 1]$ range:*

- *multi-allocation degree: $multi\_alloc_d = |R_{multi}|/|R|$*
- *exact allocation degree: $exact\_alloc_d = |R_{exact}|/|R|$*
- *under-allocation degree: $under\_alloc_d = (|R_{not}| + |R_{under}|)/|R|$*

The ideal case is one in which the exact allocation degree is close to 1 and the other two degrees are close to zero: in that case, indeed, each need in a requirement can be traced to exactly one architectural component. This situation is good because the needs are homomorphically mirrored in the architectural design, thereby facilitating the conversation between experts in either discipline. An exception to this case is when the system includes variability: in that case, it is desired to have a multi-allocation degree, for multiple components may be devised as alternative ways to fulfill one requirement.

Similar to the partitioning of requirements based on the allocation function, we can partition the set of components based on the satisfaction function. Specifically, the set of components is partitioned into two disjoint subsets: $C = C_{not} \cup C_{sat}$, where $C_{sat} = \{c \in C.\ satisfaction(c) \neq \emptyset\}$ and $C_{not} = C \setminus C_{sat}$.

Definition 3 (Satisfaction degree). *It defines the ratio of components that satisfy at least one need in a requirement as follows: $sat_d = |C_{sat}|/|C|$.*

When the satisfaction degree reaches the value of 1, all architectural components trace back to at least one requirement and, thus, their existence is justified. Unlike Definition 2, we do not include a notion of multi-satisfaction, for we are interested in assessing *whether* a component is justified or not, instead of *counting* how many needs the component accommodates.

To represent the combination of allocation and satisfaction, we introduce the metric of alignment which is a weighted arithmetic mean of the extent to which needs are allocated, and the extent to which components can be traced back to requirements. To do so, we first need to introduce the need allocation degree:

$$need\_all_d = \frac{|\{n_i \in N_R.\ (\sum_j a_{ij}) = 1\}|}{|N_R|}.$$

Definition 4 (Alignment degree). *It is a weighted arithmetic mean (with* $\alpha \in [0,1]$*) of the need allocation degree and the component satisfaction degree:* $align_d = \alpha \cdot need\_all_d + (1-\alpha) \cdot sat_d$.

In this paper, we set $\alpha = 0.5$ and give equal weight to the requirements and architecture perspectives. Similar to the debate on the β in the F_β-score [3], in-vivo studies are necessary to tune our parameter based on the relative impact of need allocation degree and component satisfaction degree. However, our experience with the software production industry reveals that early product releases include several implicitly expressed needs (e.g., printing, storage, menu interaction), thereby requiring a high $\alpha > 0.5$, whereas later releases focus on explicit (customer) requirements allocation with $\alpha < 0.5$.

The concepts and definitions above apply to the generic notions of *requirement* and *component*. In RE4SA, as per Fig. 2, we can reason about alignment at two granularity levels: *high* and *low*. The definitions and metrics can therefore be applied at either level:

- *high*: the set R contains ESs, C includes modules, N consists of outcomes from an ES, and the function *needs* returns the set of outcomes of an ES;
- *low*: R contains USs, C consists of features, N includes actions from a US, and the function *needs* returns the set of actions of a US.

5 The RE4SA Model in Practice

To assess the feasibility and usefulness of RE4SA and our metrics, we apply them to two case studies. The first presents an AD process, while the second illustrates an AR process. After introducing each case, we discuss the granularity relationships in Sect. 5.1, and analyze the alignment metrics in Sect. 5.2.

VP. The discovery case concerns a portal for vendors to manage their open invoices through an integration with the customers' ERP system. Following a requirements elicitation session with the customer, a list of USs was created and then grouped in themes. We defined ESs from the themes by rewording them and by splitting one of them into two (based on the word "and"). The SA was created by transforming the requirements into an intended architecture following the AD process described in Sect. 3.3. The software architect was allowed to include his interpretation of the requirements, e.g., by adding missing features and modules.

YODA. The recovery case regards a research application called Your Data (YODA, https://github.com/UtrechtUniversity/). A rich collection of USs was available, already grouped in themes. We used these one-word themes to formulate ESs. The functional architecture had to be recovered. As described in Sect. 3.3, this was done using a bottom-up approach. Using the implemented system, in this particular case a web application, all features were recovered by modeling every user-interactive element in the GUI as a feature.

5.1 Granularity: Exploring Refinement and Abstraction

Descriptive statistics of both cases are shown in Table 1, including the arithmetic mean for the granularity. The average number of USs in an ES is shown on the

Table 1. Descriptive statistics of both the Vendor Portal (VP) and YODA case.

Case	Level of granularity	Requirements						Comp.		Granularity	
		R	*Needs*	R_{not}	R_{under}	R_{exact}	R_{multi}	C	C_{sat}	μ*ES-US*	μ*M-F*
VP	*ES-module*	8	9	1	0	4	3	14	11	3.8	3
	US-feature	30	37	2	2	17	9	43	35	1	1
YODA	*ES-module*	12	12	0	0	12	0	12	12	8	12.6
	US-feature	96	102	3	3	84	6	161	66	1	1

top row, while the number of ESs a US is abstracted to, on average, is shown below that. The same is done for the averages of modules and features.

VP. This collection of requirements has an average of 3.8 US per ES. Analyzing our artifacts, we see that one ES only contains a single US, four modules have a single feature, and five modules only have two features. On average, a module has three features. This may indicate either the existence of few requirements per ES, high modularity, or non-detailed requirements. Due to the use of Scrum in the project, it is likely that the number of requirements will grow during development. The ES with a single US can indicate missing requirements, that it should actually be a US, or that it is expected to be extended in later phases. On the SA side, the aforementioned modules with one or two features should be analyzed as they can indicate missing features, modules to be extended, or an incorrect organization of features.

YODA. While all ESs contain at least two USs, thereby representing a proper refinement, three of them are larger than average. Regarding the modules, three contain less than two features, and one contains far more features than the average. The YODA development team can use these results to analyze their architecture and code. The larger-than-average module, for instance, may include too much functionality. In addition, the three modules with zero or one feature may lead the team to consider removing these modules or expanding upon them in the future. After speaking with the lead developer, it turns out that they have recently been working on 'simplifying' the largest module, since it was difficult to maintain and complex to use. On the other hand, they have been adding features to the modules that are relatively small.

5.2 Alignment: Studying Allocation and Satisfaction

The alignment metrics for both cases are presented in Table 2, including the ES-module alignment and the US-feature alignment.

VP. On both levels of granularity, the under-allocation degree shows that *13% of the requirements contain needs that are not addressed* by architectural components. The exact allocation degree is 0.50 for ES-M and 0.57 for US-F; roughly half of all requirements have each of their needs allocated to exactly one SA element. The remaining requirements are multi-allocated, with a degree of 0.38 for ES-M and 0.30 for US-F, which could indicate duplicate features or inefficient solutions. Only *around 80% of the components satisfy a requirement*; the remaining components are not explicitly justified by the requirements.

Table 2. The alignment-related metrics applied to the VP and YODA cases.

Relationship	Metric	VP		YODA	
		ES-M	*US-F*	*ES-M*	*US-F*
Allocation	$multi\_alloc_d$	0.38	0.30	0.0	0.06
	$exact\_alloc_d$	0.50	0.57	1.0	0.88
	$under\_alloc_d$	0.13	0.13	0.0	0.06
Satisfaction	sat_d	0.79	0.81	1.0	0.41
Alignment	$need\_all_d$	0.89	0.86	1.0	0.94
	$\boldsymbol{align}_d$	**0.84**	**0.84**	**1.0**	**0.68**

Since this is an AD process, we expect a high alignment degree, as the architecture is based on the requirements before taking implementation factors into account (as opposed to the AR process). The alignment degree is 0.84 on both granularity levels, indicating some discrepancies between the requirements and the architecture. Together with the multi-allocation degrees of 0.30 and 0.38, this seems to indicate *the requirements set is not sufficiently detailed.* The under-allocation degree indicates that the software architect either did not agree with certain requirements, or missed them during the AD process. The inexact allocation on the ES-M level can indicate an incorrect categorization of requirements, that the granularity of ES is not on a module level, or that the architect's categorization differs from that of the requirements engineer.

Figure 3 shows how USs can be allocated to features. The first US in the figure is multi-allocated, as it is linked to two features, specifically the need "*use password forgotten functionality*" is allocated to the features "*initiate password recovery*", and "*send password recovery email*". The other two USs are exact-allocated as they contain a single need and are allocated to a single feature.

The metrics from the VP case were discussed with the CEO of the company that developed the portal. He was *surprised by the low alignment score*, for the project was rather simple and the requirements were the basis for the architecture. The metrics were mentioned to be useful in highlighting potential issues with the requirements, and it was noted that the *requirements specification was not revisited after the SA creation.* Multi-allocation was seen as the most important allocation degree, as it *can indicate unnecessary costs*, while under-allocation was expected to be detected during use of the application, or denote

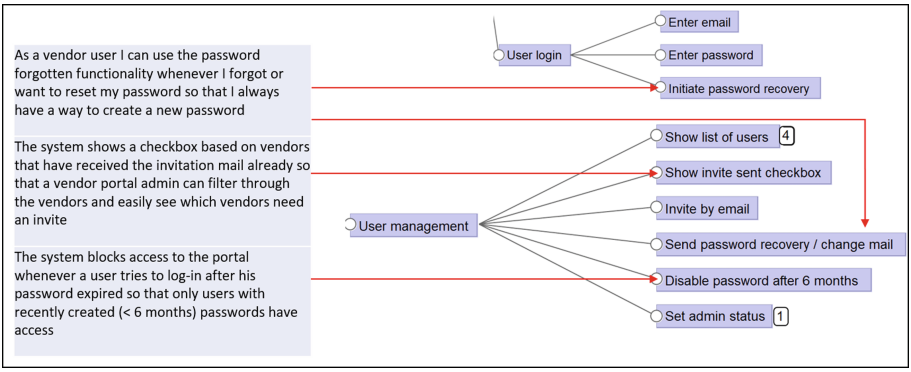

Fig. 3. Example of how USs were allocated to features.

missing features to add later. The modules that did not satisfy a requirement were judged to be a *result of missing requirements*. Finally, he mentioned the potential for making *agreements when outsourcing development*, e.g., requiring the architecture to have a 0.9 alignment degree with the requirements.

YODA. The ESs were allocated one-to-one to modules, while all modules satisfied exactly one ES; thus, these metrics are not further discussed. Nearly all USs were allocated to a feature in the architecture. Only three USs are missing completely and three others have not been fully implemented. The latter three USs contained two needs, of which only one was allocated to a feature. Regarding the features, instead, *not even half of the features satisfy at least one need*.

The missing satisfaction links may be due to a *granularity levels discrepancies*: the features are probably more specific than the USs. Also, since our feature recovery was based on exploring the GUI, some features (e.g., those related to navigation) might not need to be listed in a requirement.

According to the metrics, *not all requirements are currently allocated*: some features still need to be implemented. Moreover, since around 60% of the features do not satisfy a requirement, either the requirements are incomplete or unnecessary features exist. The lead developer explained that they *do not consider anything in retrospect*: when a US is considered completed, it is removed from the backlog. Thus, he was *unaware that six USs have not yet been fully implemented* in the system.

An example of how modules and features were recovered from the GUI is shown in Fig. 4. For the sake of brevity, the alternative features related to F2 and F3 were collapsed. The module satisfies an ES that was based on the "Metadata" theme: "When I am storing research data, I want to *include metadata about the content*, so that I can document my data." Only two of the features satisfy a US, features F3 and F4 (in Fig. 4) satisfy US3 and US4, respectively:

> US3: "As a researcher, I want to *specify the accessibility of the metadata* of my dataset, so that access can be granted according to policy [...]."

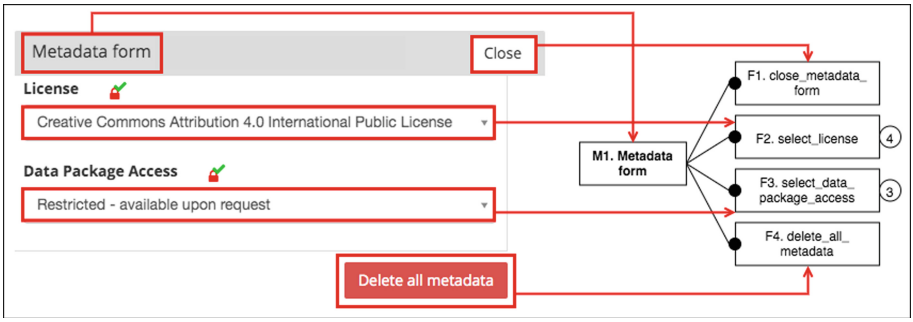

Fig. 4. Example of how architectural components were recovered from the GUI.

US4: "As a researcher, I want to be able to *discard existing metadata* and re-begin adding metadata, so that I can document a data package."

Therefore, F1 and F2 are part of the C_{not} count, while F3 and F4 are considered part of the C_{sat}.

YODA's lead developer expects the metrics to be useful, as they could help *foster the creation of trace links*, currently nonexistent. The situation is problematic when new colleagues join ("it takes approximately three months to get up to speed and be able to add something of value to the system") or when someone leaves the team, for their knowledge is lost. Also, team members often *do not know where features originate from.* To discover the rationale, the source code is checked to locate features; if unused, it is removed. This happens because the team sometimes adds features without defining the requirements first. Moreover, he expects under-allocation to be useful during development, e.g., during or at the end of every sprint, to *check whether all requirements were satisfied and if they were satisfied in full.* Finally, the multi-allocation metric may help identify duplicate features; the user stories often have overlap, causing the team to implement the same feature twice. The developer stated they are planning on *using the metrics in their next sprint aiming to improve their work efficiency and quality.*

6 Discussion

We present expected benefits from the use of RE4SA in practice, and present the validity threats to our study.

Expected Benefits. RE4SA can improve requirements-architecture communication in agile development product teams, which include product managers and product owners, through (1) simple communication means, (2) clear structural guidelines, and (3) consistent domain terminology. Combining the two granularity levels of the RE4SA model provides a shared context view of the software for the functional and technical experts. Functional experts tend to employ a

high-level overview (ES-module), while technical experts are mostly focused on the detailed level (US-feature) [32].

The objective of the RE4SA model, however, is not limited to improving communication. Gayer *et al.* [16] argued for the need of dynamic architecture creation. This architecture allows for traceability that can make software more maintainable, changeable and sustainable. The alignment relationships in RE4SA support traceability, with little documentation and effort required.

We also surmise that RE4SA helps reason about the system, for all stakeholders know which parts of the system are being discussed. In addition, when requirements are changed (modified, added, or deleted), it is apparent which other parts of the system are affected, due to the explicit relationship between concepts. Obviously, some effort is required to maintain the artifacts updated.

The RE4SA model and its metrics can be utilized for communication outside of the development team as well, such as when interacting with clients. One expected benefit is the ability *provide proof for contractual obligations*, which could also be applied to ensuring requirements alignment when outsourcing development. Using the alignment metrics, a company can prove that its system complies with the contractual requirements they and the client agreed on for the project. Furthermore, the company can provide *feedback on its progress* in percentage of realized functionality or satisfied requirements. At times, customers will have requirements for a software product that form a risk to the maintainability of the product. In these cases, the architecture can be used to *visualize the risks* of these particular elements and ensure that the customer is aware and agrees to the risks before the requirement is accepted as part of the project.

Finally, RE4SA may support release planning. The architecture highlights feature dependencies, while the requirements show the priorities. Using both perspectives, the developers can determine the *top-priority* features and, optionally, the pre-requisite features. When customers have a customized version of a software product, the architecture of the new release can be compared to the architecture of the customer [32]. Through this comparison, *incompatibilities can be detected*, allowing for better planning in an upgrade project for a new release.

Validity Threats. Concerning *construct validity*, the formulation of ESs presents some difficulties; in RE practice, ESs are formulated using the US template (epics) or as themes. Although our re-formulation did not present particular difficulties, we need to acknowledge that the ES notation we suggest is not mainstream yet. All other concepts of RE4SA (user stories, modules, features) are adopted by the industry. An *internal threat* in using the RE4SA model is *determining the 'right' levels of granularity*. While USs should describe a requirement for exactly one (atomic) feature [23], this is often unfeasible or inefficient and a US might describe a composite feature instead. For example, a US like "As a user, I want to select a language." would result in one feature 'select language'. Depending on the chosen granularity level, this feature may either be atomic, or be a composite one that is refined into separate features to switch to each supported language. To minimize this threat, we used the same levels of granularity and metrics for both cases. *Conclusion validity* is indirectly affected by

the granularity problem: should we have employed a different granularity level, the conclusions we have drawn may have differed. Regarding *external validity*, we considered only two case studies; nevertheless, the metrics are applied to real-world examples of documentation and cover common software applications.

7 Conclusion

In this study on requirements and architecture alignment, we presented the RE4SA model [26] that supports communication within the development team. We formalized the links between the four core concepts in RE4SA and we provided metrics to quantify the alignment between RE and SA. The results of these metrics can be used to analyze and improve the alignment. The metrics were applied in two industry provided cases and allow for detection of improvements in both the architecture and the requirements.

The results presented in this paper and in previous work regarding RE4SA [26,32] provide initial evidence on the suitability of our model for experimentation in practice. In particular, the AR process detailed in Sect. 3.3 allows for the RE4SA model to be used even if currently no architecture artifacts are in place.

This paper paves the way for various research directions. Firstly, we would like to study whether the linguistic structure of the artifacts, e.g., the specific words used, can help relate requirements with architectural components, and support the proper positioning of new functionality within an existing architecture. Moreover, using the sentence structures in USs, it might be possible to extract feature names from USs automatically. Secondly, evolution in agile environments [10] is a notable challenge that could benefit from the use of RE4SA. By capturing software changes introduced in extension, customisation and modification of a product in the architecture, the evolution of the product becomes visible and manageable. Utilizing the alignment relationships can be used to ensure that both the requirements and architecture stay up to date. Thirdly, we intend to apply the RE4SA model and its alignment metrics to additional cases, aiming to validate them and to determine best practices. One of the first steps in this direction is to formalize metrics for the granularity relationships, in the same manner as for the alignment relationships as presented in this paper. Finally, it is important to investigate how quality requirements are represented in agile development and how they are mapped to quality aspects in architectures.

Acknowledgements. We would like to thank Remmelt Blessinga, Abel Menkveld and Thijs Smudde for their contributions to an earlier version of this work.

References

1. Ali, N., Baker, S., O'Crowley, R., Herold, S., Buckley, J.: Architecture consistency: state of the practice, challenges and requirements. Empirical Softw. Eng. **23**(1), 224–258 (2018)
2. Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Technol. **8**(5), 49–84 (2009)

3. Berry, D.M.: Evaluation of tools for hairy requirements and software engineering tasks. In: Proceedings of the RE Workshops, pp. 284–291 (2017)
4. Borg, M., Runeson, P., Ardö, A.: Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. Empirical Softw. Eng. **19**(6), 1565–1616 (2014). https://doi.org/10.1007/s10664-013-9255-y
5. Bosch, J.: Software architecture: the next step. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 194–199. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24769-2_14
6. Bourque, P., Fairley, R.E., et al.: Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press (2014)
7. Brinkkemper, S., Pachidi, S.: Functional architecture modeling for the software product industry. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 198–213. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15114-9_16
8. Cao, L., Ramesh, B.: Agile requirements engineering practices: an empirical study. IEEE Softw. **25**(1), 60–67 (2008)
9. Cleland-Huang, J., Gotel, O.C., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: Proceedings of the FOSE, pp. 55–69 (2014)
10. Cleland-Huang, J., Hanmer, R.S., Supakkul, S., Mirakhorli, M.: The twin peaks of requirements and architecture. IEEE Softw. **30**(2), 24–29 (2013)
11. Cohn, M.: User Stories Applied. Addison-Wesley Professional, Boston (2004)
12. Coplien, J.O., Bjørnvig, G.: Lean Architecture. Wiley, Hoboken (2011)
13. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. Commun. ACM **31**(11), 1268–1287 (1988)
14. Egyed, A., Grünbacher, P.: Automating requirements traceability: beyond the record & replay paradigm. In: Proceedings of the ASE, pp. 163–171 (2002)
15. Fernández, D.M., et al.: Naming the pain in requirements engineering. Empirical Softw. Eng. **22**(5), 2298–2338 (2017). https://doi.org/10.1007/s10664-016-9451-7
16. Gayer, S., Herrmann, A., Keuler, T., Riebisch, M., Antonino, P.O.: Lightweight traceability for the agile architect. Computer **49**(5), 64–71 (2016)
17. Gotel, O., et al.: The quest for ubiquity: a roadmap for software and systems traceability research. In: Proceedings of RE, pp. 71–80 (2012)
18. Gotel, O., et al.: Traceability fundamentals. In: Cleland-Huang, J., Gotel, O., Zisman, A. (eds.) Software and Systems Traceability, pp. 3–22. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4471-2239-5_1
19. Hubaux, A., Tun, T.T., Heymans, P.: Separation of concerns in feature diagram languages: a systematic survey. ACM Comput. Surv. (CSUR) **45**(4), 1–23 (2013)
20. Inayat, I., Salim, S.S., Marczak, S., Daneva, M., Shamshirband, S.: A systematic literature review on agile requirements engineering practices and challenges. Comput. Hum. Behav. **51**, 915–929 (2015)
21. Lindvall, M., Muthig, D.: Bridging the software architecture gap. Computer **41**(6), 98–101 (2008)
22. Lucassen, G., Dalpiaz, F., Van Der Werf, J.M., Brinkkemper, S.: Bridging the twin peaks: the case of the software industry. In: Proceedings of the TwinPeaks, pp. 24–28 (2015)
23. Lucassen, G., Dalpiaz, F., van der Werf, J.M.E., Brinkkemper, S.: Improving agile requirements: the quality user story framework and tool. Requirements Eng. **21**(3), 383–403 (2016)

24. Lucassen, G., van de Keuken, M., Dalpiaz, F., Brinkkemper, S., Sloof, G.W., Schlingmann, J.: Jobs-to-be-done oriented requirements engineering: a method for defining job stories. In: Kamsties, E., Horkoff, J., Dalpiaz, F. (eds.) REFSQ 2018. LNCS, vol. 10753, pp. 227–243. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77243-1_14
25. McChesney, I.R., Gallagher, S.: Communication and co-ordination practices in software engineering projects. Inf. Softw. Technol. **46**(7), 473–489 (2004)
26. Molenaar, S., Brinkkemper, S., Menkveld, A., Smudde, T., Blessinga, R., Dalpiaz, F.: On the nature of links between requirements and architectures: case studies on user story utilization in agile development. Technical report UU-CS-2019-008, Department of Information and Computing Sciences, Utrecht University (2019). http://www.cs.uu.nl/research/techreps/repo/CS-2019/2019-008.pdf
27. Murugesan, A., Rayadurgam, S., Heimdahl, M.: Requirements reference models revisited: accommodating hierarchy in system design. In: 2019 IEEE 27th International Requirements Engineering Conference (RE), pp. 177–186. IEEE (2019)
28. Nuseibeh, B.: Weaving together requirements and architectures. Computer **34**(3), 115–119 (2001)
29. Pfleeger, S.L., Bohner, S.A.: A framework for software maintenance metrics. In: Proceedings of Conference on Software Maintenance, pp. 320–327 (1990)
30. Rath, M., Rendall, J., Guo, J.L.C., Cleland-Huang, J., Mäder, P.: Traceability in the wild: automatically augmenting incomplete trace links. In: Proceedings of the ICSE, pp. 834–845 (2018)
31. Rempel, P., Mäder, P.: Estimating the implementation risk of requirements in agile software development projects with traceability metrics. In: Fricker, S.A., Schneider, K. (eds.) REFSQ 2015. LNCS, vol. 9013, pp. 81–97. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16101-3_6
32. Spijkman, T., Brinkkemper, S., Dalpiaz, F., Hemmer, A.F., van de Bospoort, R.: Specification of requirements and software architecture for the customisation of enterprise software. In: Proceedings of the RE Workshops, pp. 64–73 (2019)
33. Tang, A., Liang, P., Clerc, V., Van Vliet, H.: Traceability in the co-evolution of architectural requirements and design. In: Avgeriou, P., Grundy, J., Hall, J., Lago, P., Mistrík, I. (eds.) Relating Software Requirements and Architectures, pp. 35–60. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21001-3_4
34. Venters, C.C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., Nakagawa, E.Y., Becker, C., Carrillo, C.: Software sustainability: research and practice from a software architecture viewpoint. J. Syst. Softw. **138**, 174–188 (2018)
35. Wautelet, Y., Heng, S., Kolp, M., Mirbel, I., Poelmans, S.: Building a rationale diagram for evaluating user story sets. In: Proceedings of the RCIS, pp. 1–12 (2016)
36. Whitehead, J.: Collaboration in software engineering: a roadmap. In: Proceedings of FOSE, pp. 214–225 (2007)
37. Yu, E.S.: Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of ISRE, pp. 226–235 (1997)
38. Zhang, Y., Witte, R., Rilling, J., Haarslev, V.: An ontology-based approach for traceability recovery. In: Proceedings of the ATEM, pp. 36–43 (2006)
39. Zowghi, D., Nurmuliani, N.: A study of the impact of requirements volatility on software project performance. In: Proceeding of the APSEC, pp. 3–11 (2002)