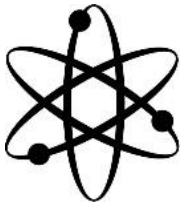

Atom Free IT
for true business agility

The Architecture Reasoning Model

v20 8-8-2019, under construction

Robert Deckers, Arne Laponin, Trupti Manik, Ani Megerdoumian

Abstract

System architects need to reason about a system for many purposes, such as making design decisions, justifying choices, and finding flaws. The architecture reasoning model (ARM) supports reasoning about a system in its environment along three dimensions. Along the application dimension is reasoned about the system's meaning and value for the customer. Along the design dimension is reasoned about the resources of which the system is built. Along the process dimension is reasoned about what the processes and organization are for the development and maintenance of the system. Along all three dimensions, statements are made about the system environment, the system itself, or the boundary between system and environment. Any statement in a particular dimension must be related to statements in the other dimensions. To define how the system fits into its environment, boundary statements relate system statements to environment statements. An architecture reasoning about a system is formed by a set of related statements that are made consistent by eliminating contradictions and by adding statements to establish coherence.

1 Introduction

A good software architecture forms a bridge between customer needs, technology constraints, and the development organization and its processes. To consciously build that bridge, every architect has to understand and analyze a large amount of statements about the different relevant aspects. A statement is something that someone has said, written, or expressed in another way. Statements are recorded in various formats and sources, and originate from various stakeholders, including the architect. They also may vary a lot in size and complexity, ranging from a single business goal, to a set of requirements, to a complete technical standard. It is a difficult task to understand the relations between all the relevant statements, and to make decisions that fit those statements. A reasoned architecture decision should follow a clear path between stakeholder concerns, design constraints, and other information, and should help to clarify the tradeoffs between possibly conflicting interests.

Figure 1 shows the Architecture Reasoning Model (ARM) for a software system. The system can be a single application, the complete application landscape of an organization, an application family, or any other abstraction of a unit of software. The ARM is intended to establish a consistent architecture; internally, and in relation to its context. An architecture reasoning is formed by a collection of related statements. The essence of a reasoning is that the critical decisions about the system functionality and quality, the software design, and the development, are made consciously. The architect does this mostly in cooperation with various stakeholders. The ARM can be used to:

- Make reasoned architecture decisions, such that it is clear what a decision means for the system's application, the technical design, and the involved realization activities.
- Analyze the impact of changes in business needs, technology, development and maintenance process, or development and maintenance organization.
- Analyze and check the consistency of an architecture. Do the architecture statements address all the major stakeholder concerns and given constraints? Are there any architectural statements that have no clear goal? Are there contradicting statements for some aspects?
- Relate technology choices, business goals and project plans, for example in a workshop with various stakeholders.

This paper explains the structure, elements, and application of the ARM. Section 2 discusses the distinction between a system and its environment. Section 3 explains the three dimensions of the ARM. Section 4 explains how to set up a reasoning with the model. Section 5 shows the application of the ARM to an example from industry. Section 6 discusses related work. Section 7 discusses our conclusions about working with the ARM in practice.

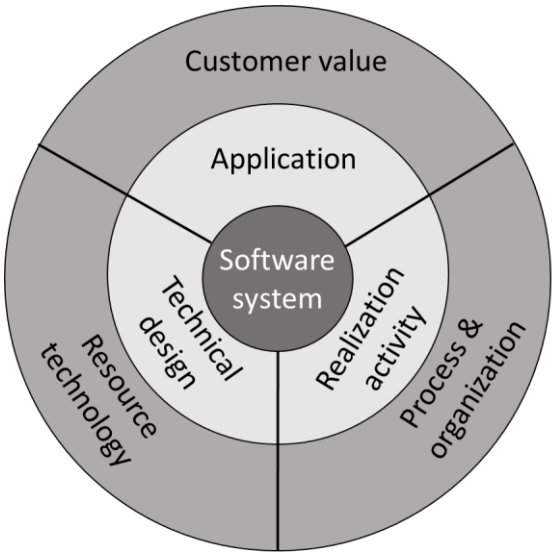

Figure 1 The architecture reasoning model.

2 A system and its environment

When defining an architecture for a system, the boundaries of the system must be clarified through an explicit statement of the system scope. This clarifies what aspects and things are part of the system and what can be seen as the system environment. Via an explicit system scope, we distinguish three levels of statements:

- **The environment level** that covers statements that are independent from the system, but the system has a direct or indirect relation with.
- **The system level** that covers the functions of the system, the conceptual design elements, and the main realization approach concepts.
- **The boundary** level that covers the statements that relate the system to its environment.

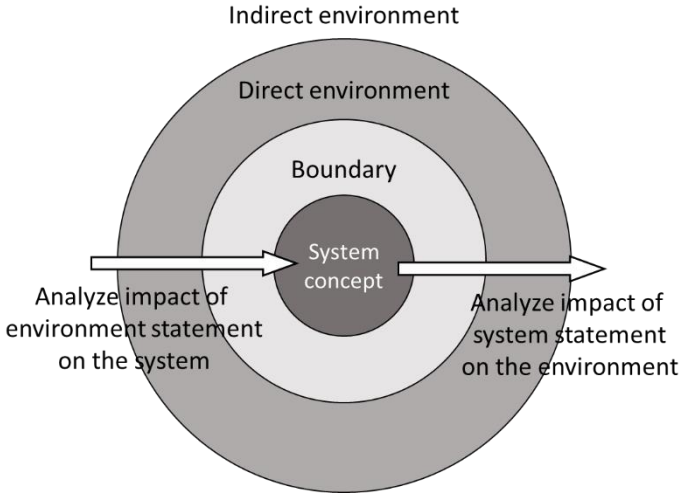

Figure 2 Reasoning between the system and its environment.

Figure 2 shows the different levels of statements. The architect analyzes via these levels about the relation between a system and its environment in two ways: environment statements may influence the system concept, and decisions about the system may impact the environment. These analyses only make sense if the stakeholders have a shared definition of the system scope. The architect is mainly responsible for the system statements, and typically defines the boundary statements together with other stakeholders. The architect is not primarily responsible for the environment statements, but often advises about them. The next sections explain the different levels in more detail.

2.1 Environment statements

To design a system that fits its environment, information is needed about that environment. System stakeholders make many environment statements, for example "our clients mainly reside in large cities", or "our IT infrastructure is based on Linux". The validity and importance of those statements will depend on the responsibility and role of the stakeholder in relation to the system (development). Some of the environment statements can be assumptions, for example "the amount of available COBOL programmers will decrease". The bigger the impact of an environment statement on the system is, the more important it is to let stakeholders validate that statement.

Environment statements may arise from goals and concerns of the stakeholders, developments and trends in the market, documentation about existing systems, platforms, technologies, or related projects and processes. They can be stated as facts, assumptions, or any other form of perception.

To assure that the system keeps fitting its environment, the environment statements have to be checked on their validity throughout the system's lifecycle. Of course, this is also applicable to new relevant statements that arise in the form of new requirements or constraints. A reliable validation of environment statements is required to ensure that the system fits its environment during all phases in its lifecycle.

2.2 System statements

System statements define the essence of the system independently from its environment. For example, "the main function of the system is to records and retrieve data", or "the system design consists of three software layers". The system definition is the basis for determining the possibilities and limitations of architectural decisions.

Often generic terminology that include concepts like function, design aspect, component, or development task, is chosen to formulate system statements. These concepts are also

related to each other. For example, functions are assigned to components, design aspects are relevant within components, or development tasks realize aspects of components. A coherent set of system definition concepts is a prerequisite for a consistent architecture.

2.3 Boundary statements

Boundary statements express how environment statements are linked to system statements, for example "the data storage layer is implemented with MongoDB" or "the online availability of the retrieve function enables that customers can always access their data". To obtain a consistent architecture, all relevant environment statements must be linked to a system statement, and vice versa. The boundary statements of a software system often address topics like the communication between the system and other systems, the relation between system qualities and customer requirements, how the systems uses platforms and standard components, or the way the system is developed and maintained. Sufficient and validated environment statements in combination with clear system statements form the basis of the system boundary specification.

To illustrate the boundary concept, we give an example about components. A component that is not managed and developed within the system scope belongs to the system environment. The way such a component is integrated and used in the system, is part of the system boundary. The statement that the system requires a certain external functionality that must be fulfilled by some component, is a property of the system itself.

3 Reasoning in three dimensions

An architecture must form a coherent set of statements. The architecture statements must fit statements that express the problem to solve, the technological resources, and the activities and artefacts of the process for development, maintenance, and management. Therefore, the ARM distinguishes the dimensions application, design, and process (see Figure 3), which are explained in the next sections.

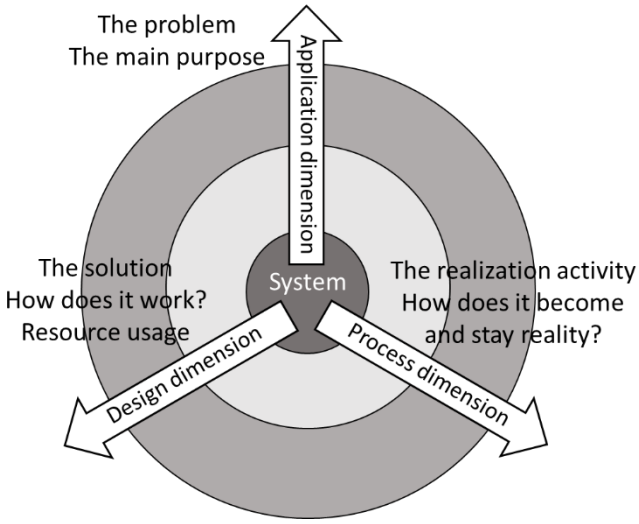

Figure 3 Application, design, and process.

3.1 Application dimension

Along the application dimension we reason about what the systems offers to the users and customers of the system, and how that impacts them. They give the system its main reason for existence. The system functionality and its quality must be suitable for the

targeted customer value. The application dimension addresses customer goals, use cases, system functions and the quality attributes of those functions. Figure 4 shows the outline of the application dimension.

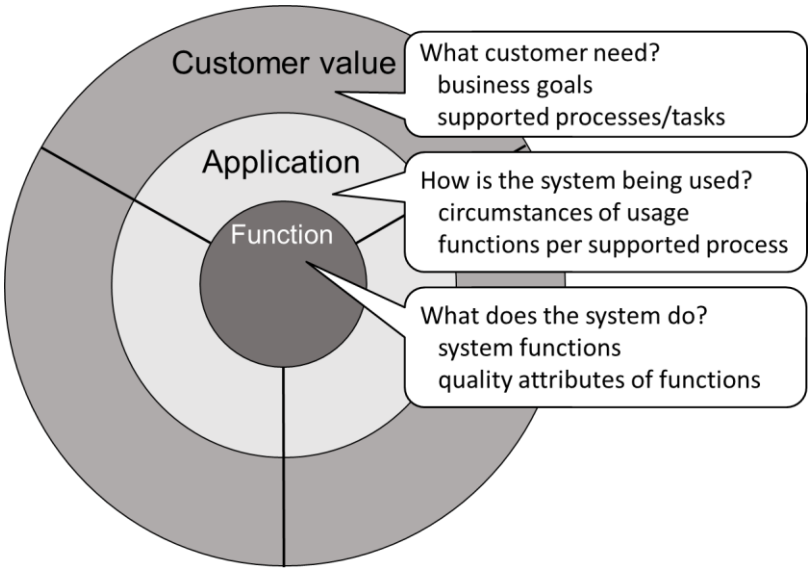

Figure 4 The application dimension for a software system.

The functional needs are the starting point for analyzing the desired system functionality and function quality. Types of functional needs include user task support, traceability of a business process, and information needs of business management. The system must provide the right features and quality to support these needs.

The application environment often imposes quality requirements on the systems in that environment. Most software intensive systems nowadays are not unique in terms of functionality. For example, each insurance company has an information system for customer administration, policy administration and claims handling. Although the functionality for each company is largely the same, the business benefits of different systems may vary substantially. These differences are found in functional qualities. The quality of a function is expressed via quality attributes like availability, usability and reliability. In comparison, The ISO/IEC 25010 standard [1] distinguishes attributes to express quality in use. The importance of a quality attribute is dependent on the context in which the functionality is implemented. This is determined by the business goals that the organization pursues with the (software) system.

3.2 Design dimension

The design dimension addresses how the system conceptually works and how it uses existing designs and other systems. The design dimension is about the division of the system into parts, like components and other design aspects, and their coherence.

The conceptual design a software system identifies design aspects, logical components, and conceptual design decisions that have to be implemented with technology from the system environment. The available resources comprise existing technologies like existing design patterns, IT-infrastructure, software libraries, and programming languages. The technical design is expressed in the designs and configurations of existing or to-be-acquired software and the connections between them. Figure 5 shows the outline of the design dimension.

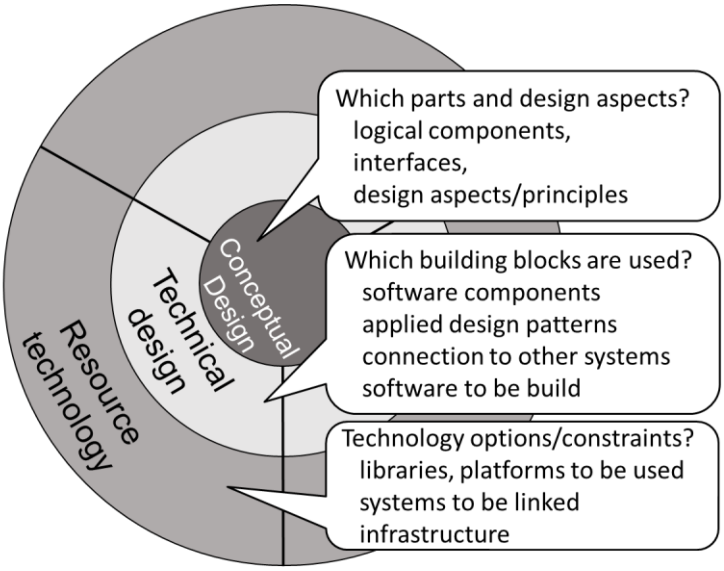

Figure 5 The design dimension for a software system.

The design dimension focuses on the stakeholders who build the system, install the system, and have requirements with respect to the system parts. Without them, the system could not be made. Typical stakeholders are programmers, testers, infrastructure architects, technology managers, and technology suppliers. To what extent each stakeholder is taken into consideration is partly determined by the architecture. For example, the IT procurement is more important when a bought component is integrated in the system than in the case the system is completely self-made.

Along the design dimension the architect usually considers the partitioning of the system. A partitioning typically defines the structural elements that together form the system, the responsibilities of each element, the interfaces to systems in the environment and between the elements themselves, the interaction between the elements, and the design patterns that the elements must adhere to.

3.3 Process dimension

Along the process dimension we reason about the way the system is developed and maintained, how that is planned and managed, and about the required expertise and organization to do that. The system development approach is executed in the process environment. For example, concepts such as the Project Start Architecture [2] and architectural guidelines are intended to integrate architecture in the realization process.

The system development approach must identify the main realization tasks, responsibilities, and the guidelines to apply during realization, for example the principles for testing and planning. Figure 6 shows the outline of the process dimension.

Figure 6 Process dimension for a software system.

The approach for development and maintenance must fit the capabilities, and processes of the development and maintenance organization. The decisions in the other dimensions of the ARM have their impact on the required expertise and tools used. For example, the choice for Java as the programming language requires programmers with knowledge of Java. The system elements from the other dimensions must be mapped onto the different steps and responsibilities in the process. For example, a release schedule based on use cases must be supported by a development schedule that is structured via those use cases.

4 Reasoning with the ARM

The starting point of a reasoning is an initial statement that needs to be analyzed. A reasoning clarifies the relation between the initial statement and other statements.

4.1 Statements

Statements exist in different forms and sizes as is shown in the examples in the previous sections. To create a consistent set of statements, the following properties must be considered for each statement:

- **The description** of the statement; either in natural language or a more formal (modeling) language.
- The **aspects** that the statement is about. In the different dimensions of the ARM different aspects play a role. Types of aspects include quality attributes such as reliability and availability, design aspects such as interfaces and error handling, and process aspects such as planning and project duration.
- The **scope** to which the statement is applicable, such as a specific organization, project, function, or service. The scope may also address a timeframe. A statement about the current situation can simply be prefixed with "Now". Not all statements will have the same scope. Some statements are valid for a larger number of systems. Some are only valid for a part of the system.
- The **relations** with other statements. For example, one statement is a specialization or decomposition of another statement.
- The **validity** of the statement expresses to what extent the statement is true. For example, it is an assumption, it is a requirement, an observation of the current

situation, or it is tested and agreed. This is important when decisions are questioned. Needs, requirements and assumptions could be revised for example. When designing a system, many statements will be about the desired situation.
- The **owner** of the statement. This is the organization or person who made the statement. He might be involved in discussions about the importance and validity of the statement.

Some of these properties might already be expressed in other specifications, like design models, or requirements documents, and can be referred to. Also, it is in practice not necessary to explicitly define all properties of all statements. Often properties are obvious to most stakeholders. Though, lack of clarity about any of the properties may lead to a faulty reasoning. For a particular reasoning it will be useful to split some statements and to combine others. This depends of course on the goal of the reasoning.

If not all the statement properties are known and it hinders reasoning, then the statement is made **open**. An open statement identifies the unclear statement property. An open statement can also be included when it becomes clear that statements about some aspects are missing. In this case, the open statement explicitly states the missing aspect. Consistency is difficult to achieve when a reasoning contains too many open statements. In such a situation, you should stop the reasoning and first obtain more clarity on the open statements.

Figure 7 shows the syntax for statements. The first is a **normal** (or closed) statement. The second is an open statement that indicates that the scope of the statement is unclear. The third is an open statement that indicates that a statement is required on the "programming language for applications."

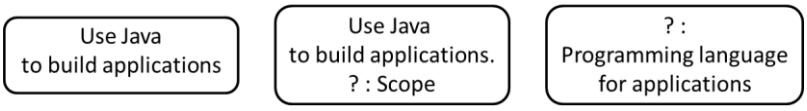

Figure 7 Notation for statements.

4.2 Consistency relations

A reasoning comprises a set of consistency relations between statements. A consistency relation between statements A and B can have three possible values (see Figure 8):

- The consistency between A and B is **undetermined**: A and B should be consistent, but it is not clear if they are. An explanation can be added for why the consistency cannot be determined or what would help to determine it.
- A is **consistent** with B. So, A and B do not contradict each other. For example, "the system is made with open-source technology" is consistent with "The system is programmed in MySQL". A rationale for why the relation is considered to be consistent can be added.
- A is **inconsistent** with B. A and B contradict in the scope of the system and its environment. For example: "It uses open source tools" conflicts with "The system is programmed in Microsoft Visual Studio.". An explanation can be added for why the relation is considered to be inconsistent or what would what help to solve the inconsistency.

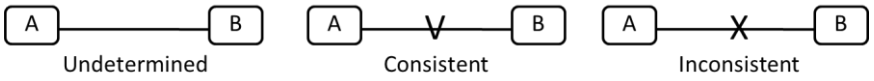

Figure 8 Possible values of the consistency relation.

4.3 Reasoning consistency

A reasoning begins with the identification of possible sources of statements, such as business plans, process documentation, development repositories, presentations, or meeting minutes. To get a complete reasoning, often statements in all three dimensions are required. One can simply not consider all sources of information and must deal with the abundance of statements in a pragmatic way. The architect must therefore have knowledge about all the sources or involve others who have it.

To set up a reasoning it must first be clear what the definition/scope of the system is, as indicated in section 2. The scope must be discussed with the involved stakeholders such that they have a shared understanding of the system concept. The scope discussion should address all three dimensions explicitly. After scope definition, the available statements can be positioned in the ARM. Figure 9 shows questions that help to position statements.

In practice, not all available statements will fit exactly in one of the seven demarcated sections of the model. Such statements are positioned on the border between two sections or split into statements that each fit one section. Also some statements will overlap. Such statements are connected to each other or they are split. Split statements must be linked together to preserve the semantics of the original statement. The aim is to form atomic statements. Newly introduced split or merged statements must still be acknowledged by the statement owner.

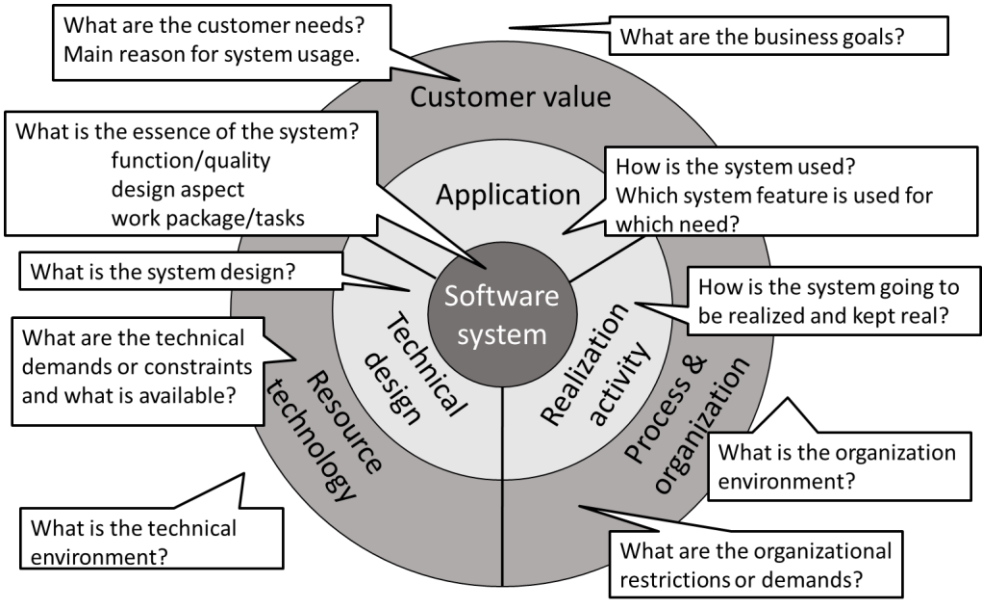

Figure 9 Questions that help to position the statements.

The ARM forms a framework to relate statements to each other. To ensure that a system will fit it's context, each environment statement must be connected to system statements via boundary statements, and each system statement must be associated environment statements via a boundary statements. To obtain a consistent architecture across all three dimensions, statements are associated as follows:

- For each statement in the application dimension it must be clear what design statements form the fulfilment of the application statement and how this is secured in the realization process.
- For each statement in the design dimension it must be clear which statements it serves in the application dimension and which process statements contribute to their realization.
- For each statement in the process dimension it must be clear what part of the design it concerns and to which statement in the application dimension it contributes.

4.4 Reasoning steps

A reasoning begins with the need to assess how well a particular statement matches other statements. That particular statement is called the start statement. Broadly a reasoning goes as follows:

1. Determine the clarity of the (start) statement.
    a. If not, make it an open statement and indicate which statement property needs to be clarified. The owner of the statement should provide that clarity. If a statement is reformulated, the owner must still recognize it as his statement.
2. Select the statements that are related to the start statement. For most statements this is clear, for some it is not. Guideline: when in doubt, include it. You can always ignore it later.
    a. First, look for statements in the same section of the model.
    b. Then look for statements in adjacent sections.
    c. Check if other sections also contain statements that are clearly linked. Possibly reposition statements due to new insights.
    d. If no related statements are found, then the statement should be discussed with respect to its validity, or new open statements are identified to make sure that the statement is not standing on itself.
3. Determine the consistency of the relations: consistent, inconsistent, or undetermined (see qqq).
    a. If an inconsistency is detected, an open statement can be added to indicate how the inconsistency can be solved. In case of an inconsistency in the same section, the adjacent sections often contain statements that cause the inconsistency. In case of an inconsistency between sections that are not adjacent, a statement in at least one of the intermediate sections is required.
    Determine if it makes sense to continue the reasoning, or if the inconsistency needs to be solved first.
    b. If the consistency is undetermined, make the statement open and indicate which property needs to be clarified (like in step 1a). The reasoning can continue in most cases. Though it is useful to decide about stopping or continuing with the reasoning.
4. Repeat the previous from Step 1 for each related statement until no new related statements are found. Stay focused on the start statement. It is better to create a separate reasoning if an inconsistency is detected that has little to do with the starting statement.
5. Remove statements that are not related at all. Apparently they are not relevant.

The result of the above steps is a judgement on the consistency between the start statement and the other statements. In case there are inconsistencies, the reasoning should offer a direction to solve them. This means that in all sections of the ARM, an appropriate statement must be made that is consistent with the rest. Some statements are so trivial that they can be skipped. Keep in mind that things that are trivial now, might be not so trivial in the future.

Comprehensibility of an architecture reasoning is in practice more important than accuracy or completeness. A reasoning should give the involved people insight into the rationale of particular choices. An explicit reasoning makes the most sense for the architecture decisions in which the most important stakeholder concerns are balanced.

5 An example: the Crowdocracy project

This section describes the use of the ARM in a project that developed an online platform for Crowdocracy. We explain shortly what Crowdocracy is, set the outline of the platform development, and show two reasoning examples. The objective of these examples is to clarify how the ARM is used to reason about the consistency of statements across various documents in a project. The ARM has been used in many other organizations since 2009. These organizations did not allow to publish their architecture decisions and other sensitive information. The Crowdocracy platform is open source and this enabled the publication of the reasonings presented in this section.

5.1 Crowdocracy

Crowdocracy is a process that involves the general public in developing policies for the issues that society faces. Members of communities, such as neighborhoods, cities, or countries, raise issues and develop policies together. Crowdocracy is described in *Crowdocracy: The End of Politics* by Alan Watkins and Iman Stratenus [3]. They tried out the Crowdocracy process with people from the Dutch tulip seller cooperation and saw that people feel involved and take more responsibility for the future of their organization. Watkins and Stratenus wanted to know if Crowdocracy also works for a large number of participants, and therefore wanted to have an online platform.

The Crowdocracy process consists of several steps: raise an issue that bothers the community, make a proposal to solve the issue, comment and integrate feedback, and finally vote. Any member of a community can participate. A participant can raise an issue that bothers him and/or his community. For example, a suburb of Rome may raise the issue of uncollected garbage on the city streets. After that, any other participant can come up with a proposal to solve the issue. A group of interested participants, the so called shapers, integrate all the different ideas from the community into a single cohesive proposal. Other participants give feedback on the proposal or suggest things that the shapers might have overlooked. Once the shapers think that the proposal is well-formulated, they call for a final vote, during which all community members may vote on the proposal. If the proposal passes, it is up to the members of the Executive branch to implement it. The whole process is monitored by neutral guardians, who cannot vote nor influence the participants and shapers. The guardians see to it that the community members treat each other respectfully and that the proposal progresses.

5.2 Development of the platform

The goal of the platform development project is to let people participate from different locations, at moments it suits them, via their own devices. The platform has to follow the Crowdocracy process and guard the users' privacy. The project is an initial development and after it's over the platform is handed over to another development team.

The platform was built with the Django REST framework for the backend and ReactJS for the frontend. The backend and the frontend communicate via a REST API. A uniform and simple UI-template was used so that the platform is compatible with a wide variety of access devices and is suitable for people with different levels of online skills.

5.3 Statements and system scope

The project handled and created several documents which contain the information used in our reasoning examples. The list of statements below were extracted from the project assignment, the project plan, the requirements document, the technology scan, the architecture description document, and the description of the Crowdocracy process. Many of the statements are not just the one-liners presented here, but refer to a whole document or another unit of specification. As a result, some marked consistencies in the reasoning examples might seem questionable, but that is only because we cannot provide the complete specifications here.

Before statements can be positioned in the ARM, a scope definition is needed. We see the scope of the system as follows: The platform should digitally support the Crowdocracy processes and roles, and people must be able to participate via their own device. The only restriction to the used technology is that there is no budget to buy software or user licenses. An initial version should be realized by the assigned development team within eight weeks by following a SCRUM based approach. After which the results will be handed over to another development team.

With the given scope definition we can position the statement. QQQ shows the result. As an example we given some explanation of some positionings:

- "Raising an issue is the first step of the Crowdocracy process" is clearly in the application dimension, and it is independent form the platform. So, it is in the Customer Value section.
- "Architects create the domain model" is a statement that combines the role of architect, which exists without this system, with the notion of a domain model, which is a system statement. Creating is clearly an activity. Hence the position is in the realization activity section.
- "API to connect the frontend and backend" is a statement independent from the system environment, and it states parts of the design. It clearly does not state something about functionality or how it is realized. Hence, it is positioned in the Conceptual Design section.

Customer Value:

1. The roles and the tasks of the Crowdocracy process are changeable.
2. Validate that the Crowdocracy process can be used by a distributed community.
3. Users should be able to participate in the Crowdocracy process with their own device.
4. Independence of thought: community members should not be swayed by the identity or status of other community members.
5. Raising an issue is the first step of the Crowdocracy process.

Application:

6. The platform should support multiple types of devices.
7. User story for raising an issue.
8. The platform must be usable for people with different online skill levels.
9. An Issue is described with a title, description, proposal, and author.
10. Consistency of the user interface enhances usability.
11. Domain model is based on the concepts in the Crowdocracy process (such as Issue).

Function:

12. UI has a form for raising an issue.
13. Same color schema on every page.
14. Same types of buttons, forms, and other UI elements used on every page.
15. The domain model captures the concepts from the Crowdocracy domain.

Resource Technology:

16. ReactJS runs on a browser.
17. ReactJS needs a library for state management.
18. ReactJS supports multiple devices.
19. UI-templates written in ReactJS are available.

Conceptual design:

20. The user form is filled in the frontend, processed in the backend, and stored in the database.
21. The frontend sends a filled form to the backend.
22. API to connect the frontend and backend.
23. Each domain entity from the domain model is put into a separate Django file.

24. Each public Django method corresponds to a HTTP request.
25. The frontend is decomposed according the ReactJS guidelines, such as a form becomes a component.
26. The database design is normalized to the third normal form.

Technical Design:

27. Django REST Framework for the backend.
28. The backend is programmed in Python.
29. PostgreSQL is the DBMS.
30. ReactJS is the frontend framework.
31. ReactJS UI-templates are used for the frontend.
32. State management of the frontend is realized using Redux.
33. Database schema design via an ER diagram based on the domain model.

Process & Organization:

34. The team had no experience in frontend development using modern web technologies.
35. Scrum user stories express value to the customer.

Realization Activity:

36. Acceptance test criteria define when a user story is realized (by the platform).
37. User Stories are based on Crowdocracy process steps.
38. The development team is divided into three sub-teams.
39. Every sub-team has a backend developer, frontend developer, and a tester.
40. Scrum is used by the development team.
41. Each sub-team is assigned a set of user stories.
42. "Raise an issue" becomes a user story.
43. The development period of the project is two months.
44. Developers review code against coding guidelines.
45. Backend developers document the API.
46. Architects create the database design.
47. Architects create the domain model.
48. Product Owner specifies the acceptance criteria of the user story.

Approach:

49. Database design is created manually.
50. Domain model is created manually.
51. Each user story becomes a separate development task.
52. Project consists of two-week-long iterations, which end with a demo for the client.
53. Each frontend developer writes frontend code.
54. After initial development, the project results are handed over to another development team.

5.4 Reasoning 1: is "raise an issue" predictably realizable?

The first reasoning that we present begins with the start statement "Is raising an issue predictably realizable?" (abbreviated as "raise an issue"). We want to analyze if the step of the Crowdocracy process called "raise an issue" is covered by our architecture and other project artefacts. By following the steps outlined in section 4.4, we will determine the consistency between the start statement and the other statements.

"Raise an issue" is located in the Customer Value section of the ARM. The next step is to identify all the statements that are related to the start statement, as depicted in *Figure 10*. One can derive from *Figure 10* that not all the statements are relevant to the start

statement. Throughout the rest of this reasoning more statements can become irrelevant and will be removed.

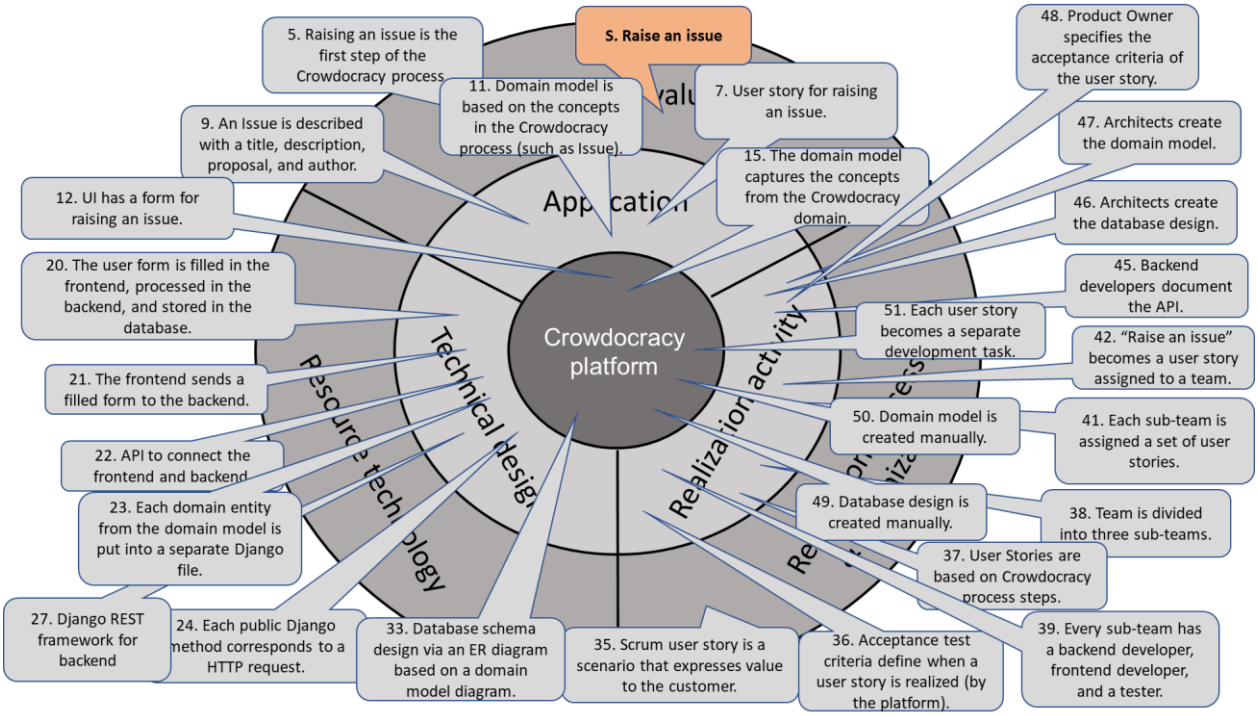

Figure 10. Starting statement with its related statements

After identification of all the related statements, the first iteration of the reasoning is conducted. The consistency between the start statement and the related statements is assessed. Once the consistencies between the statements have been assessed, we select another round of related statements to traverse the graph of related statements. We have chosen a breadth-first approach for exploring the neighboring statements, but a depth-first approach is also possible. The result of the first two iterations is visible in *Figure 11*. Statements 5, 7, 9, 11, and 15 make up the related statements of the first iteration, while 12, 23, 37, 50, 51, 47, and 48 are in the second iteration. Until now, most of the statements selected, are consistent with each other. Though the consistency between statements 11 (*Domain model is based on the concepts in the Crowdocracy process)* and 47 (*Architects create the domain model*) is undetermined. Namely, issue is clearly a Crowdocracy concept and it is unclear how it is used in the creation of the domain model. In the figure it is indicated that it could be solved by adding a statement about the method that the architects use to base the domain model on the Crowdocracy concepts.

We have used a drawing tool to present the reasoning flow. But, mostly we use a whiteboard and sticky notes to do the reasoning and make photos of it during the process. Of course, it would be better to have a dedicated to tool do the reasonings and also to store the reasonings in a repository that is based on the ARM concepts and connected to the development repository of the system.

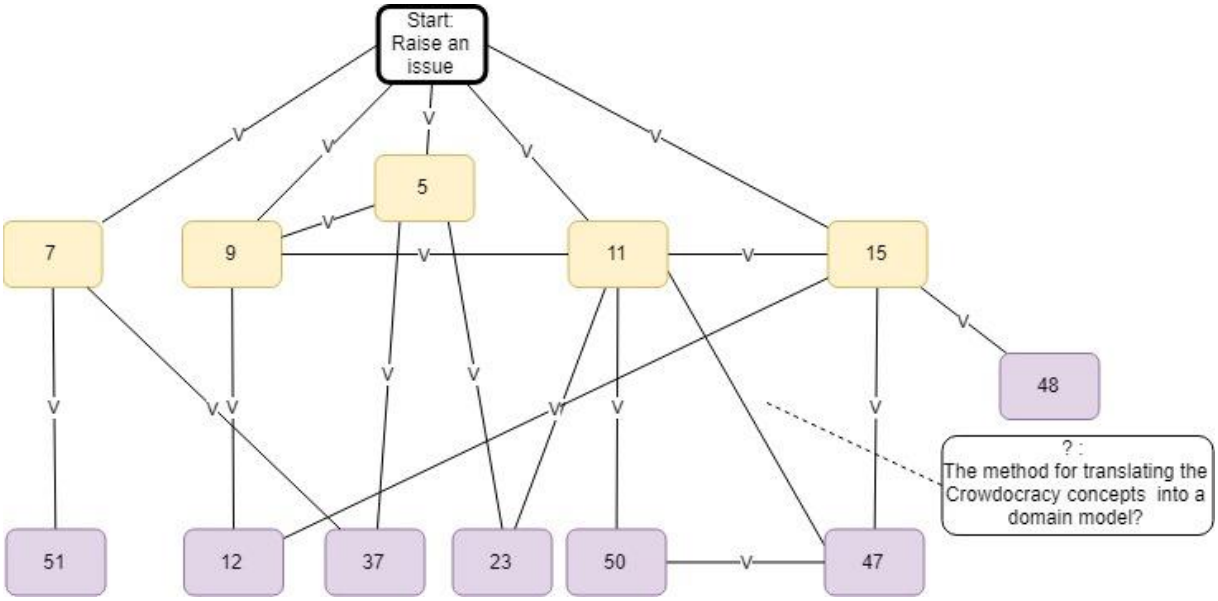

Figure 11. Reasoning after the first two iterations

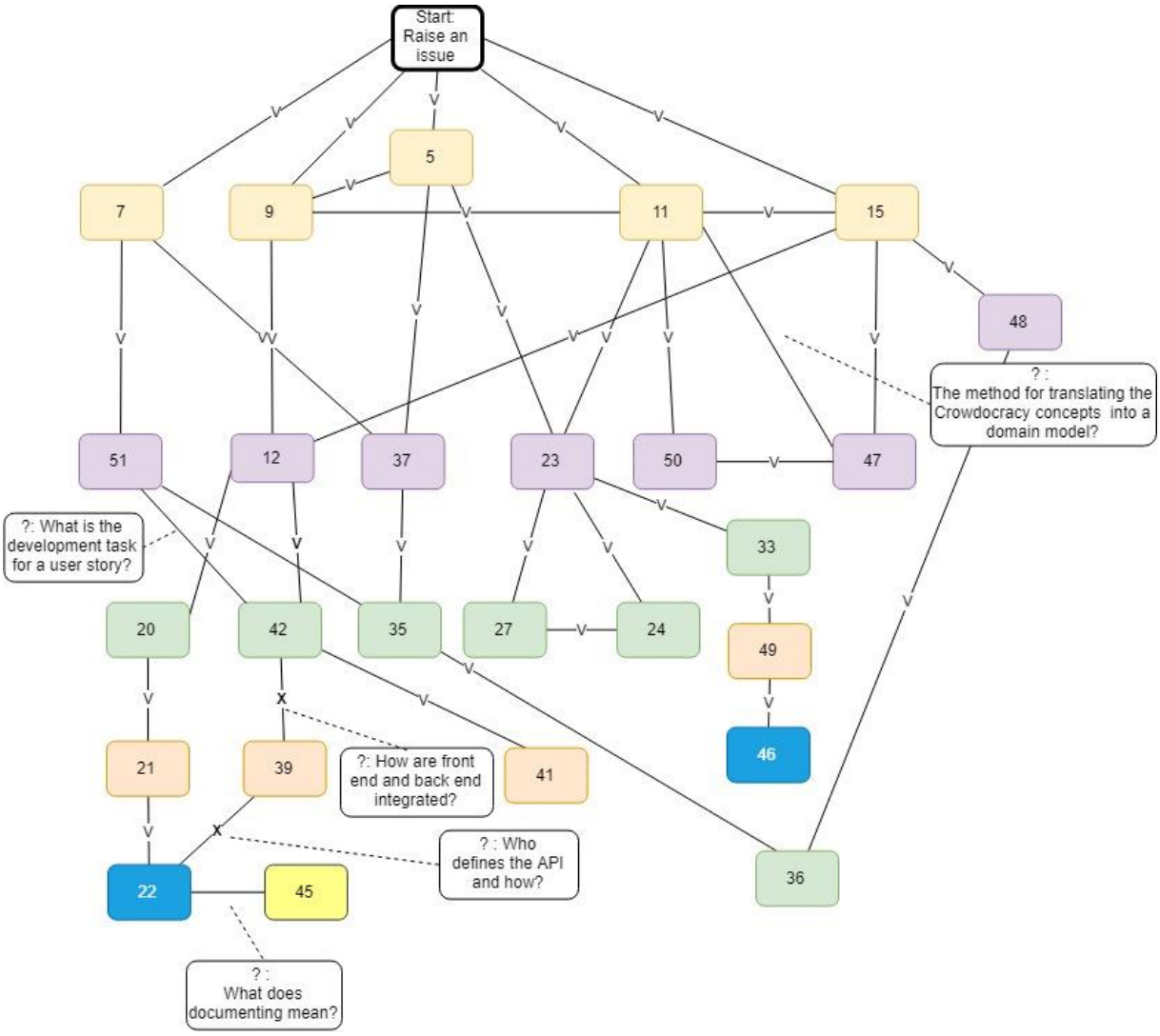

Figure 12. Reasoning after last iteration with inconsistencies and questions

In the next iteration, we discover an undetermined relation between statements 51 (*Each user story becomes a separate development task*) and 42 *("Raise an issue" becomes a user story).* It is unclear what exactly are the ingredients of a development task for a user story. In the next iteration, statement 39 (*Every sub-team has a backend developer, frontend developer, and a tester*) brings more clarity, assuming that a backend developer is responsible for the backend portion of the development tasks. Yet, it also introduces an inconsistency between statements 39 and 42, because it is unknown how the back end and front end are integrated into a coherent whole. The next iteration considers statement 22 that states that an API is used to connect the backend and the frontend. Though this resolves the previous inconsistency, it brings up the question of who defines that API and how. Statement 45 *("Back end developers document the API")* brings a bit more clarity, but also raises the question what documenting exactly means.

Figure 12 shows the reasoning after all the iterations, including the open statements. When all the questions have been answered and all the inconsistencies have been resolved then we can say that the user story "Raise an issue" can be predictably realizable. But with the current set of statements this is not the case.

5.5 Reasoning 2: is ReactJS a good technology choice?

The second reasoning starts from statement 30 *("ReactJS is the frontend framework")*. We want to analyze if this technology choice fits the architecture and other specifications. By following the process outlined in section 4.4, we determine the consistency between the start statement and the other statements.

The starting statement is located in the Technical Design section of the ARM circle. *Figure 13* shows the start statement with all the related statements.

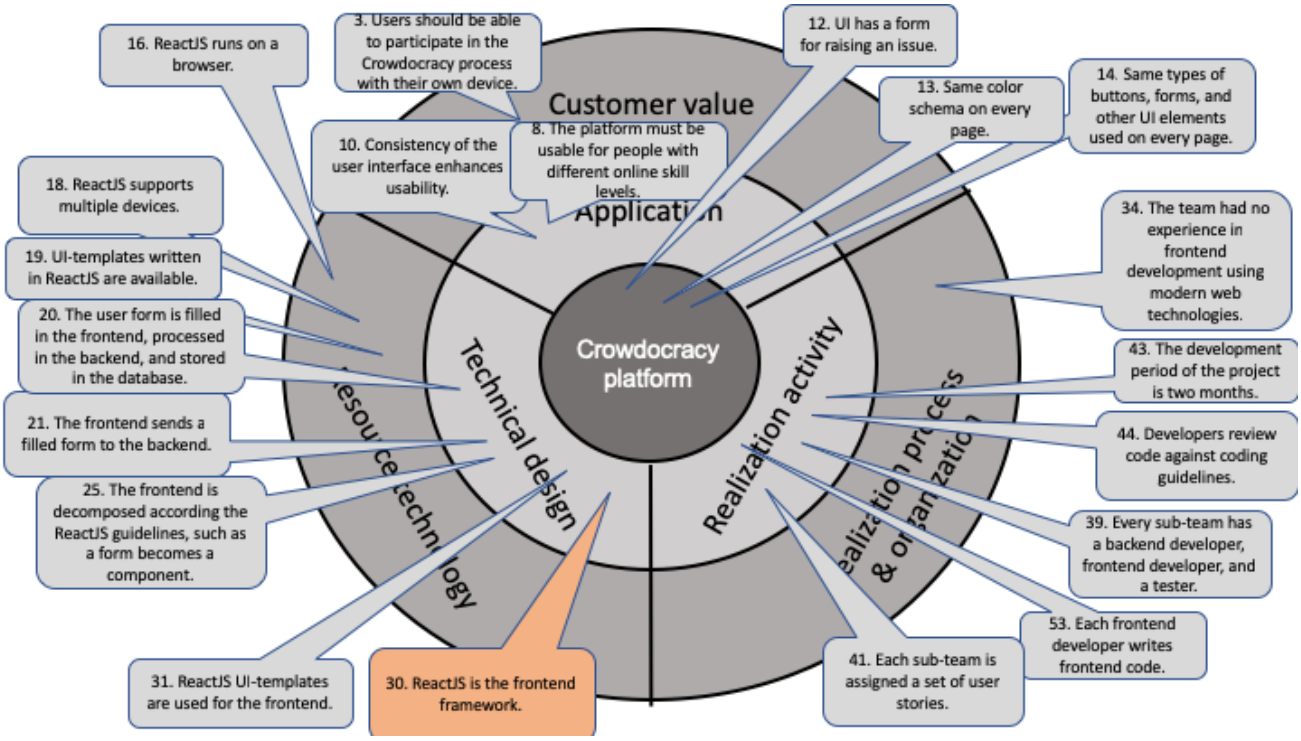

Figure 13. Statements that are considered to be related to the start statement.

In the first iteration, the consistency between the start statement and the statements 18, 31, 25, and 53 is assessed. As an example, statements 30: (*ReactJS is the frontend framework*) and 53: (*Each frontend developer writes frontend code*) are consistent, because we assume that the frontend developers write the frontend code in ReactJS. In the second iteration, depicted in *Figure 14. Reasoning after the second iteration.*, we did

not detect inconsistencies either. Statements of the second iteration, such as 18: (*ReactJS supports multiple devices*) and 3: (*Users should be able to participate in the Crowdocracy process with their own devices*) are consistent, because we checked that the supported device types suffice.

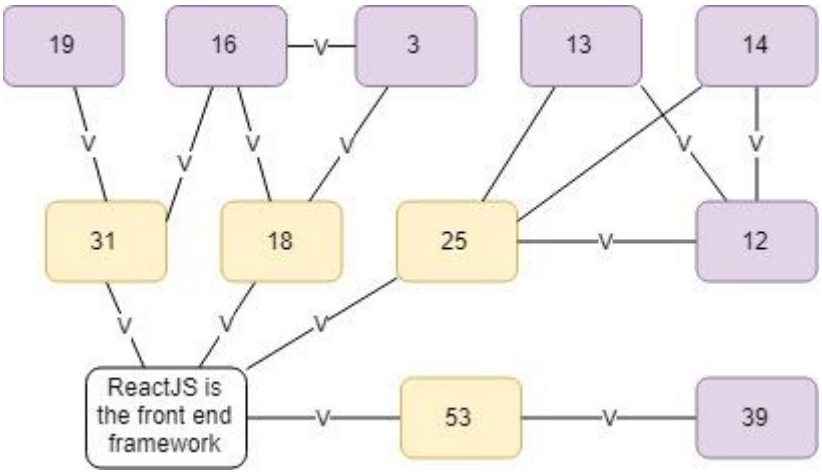

Figure 14. Reasoning after the second iteration.

When we explore the relations between statements during the third iteration, depicted in *Figure 15*, a number of questions arise. Statements 39 (*Every sub-team has a backend developer, frontend developer, and a tester*) and 41 (*Each sub-team is assigned a set of user stories*) rise questions about how the developers guard the consistency of the written code while switching between different user stories, and how frontend developers, who are spread between different teams, ensure that their code is written in a uniform way. This discussion leads us to an interesting discovery in relation to statement 54 *("After initial development, the project results are handed over to another development team.").* There are no statements about the documentation of the code or other project results. It this case it is advisable to start a new reasoning about statement 54, because it is probably a severe problem.

Additionally, there is an inconsistency between statements 39 (*Every sub-team has a backend developer, frontend developer, and a tester*) and 34 (*The team had no experience in frontend development using modern web technologies*). If the team does not have any experience in developing using modern web technologies, how are they going to finish the project in a predictable way. A possible solution would be to add statements about a training period or strict coding guidelines on the basis of the guidelines mentioned in statement 25 *("The frontend is decomposed according the ReactJS guidelines")*. Guidelines could help the team familiarize themselves with the best practices of the technology and thus create more uniform code. Additionally, the same measure would resolve the slight ambiguities between statements 25 and 13 (*Same color schema on every page*), and 25 and 14 (*Same types of buttons, forms, and other UI elements used on every page*), which we did not address in the first two iterations. ReactJS provide buttons, forms, and other UI elements of the same type, but it is up to the development team to ensure uniform usage across every page of the Crowdocracy system. Yet, a training period and coding guidelines are not mentioned in any of the statements.

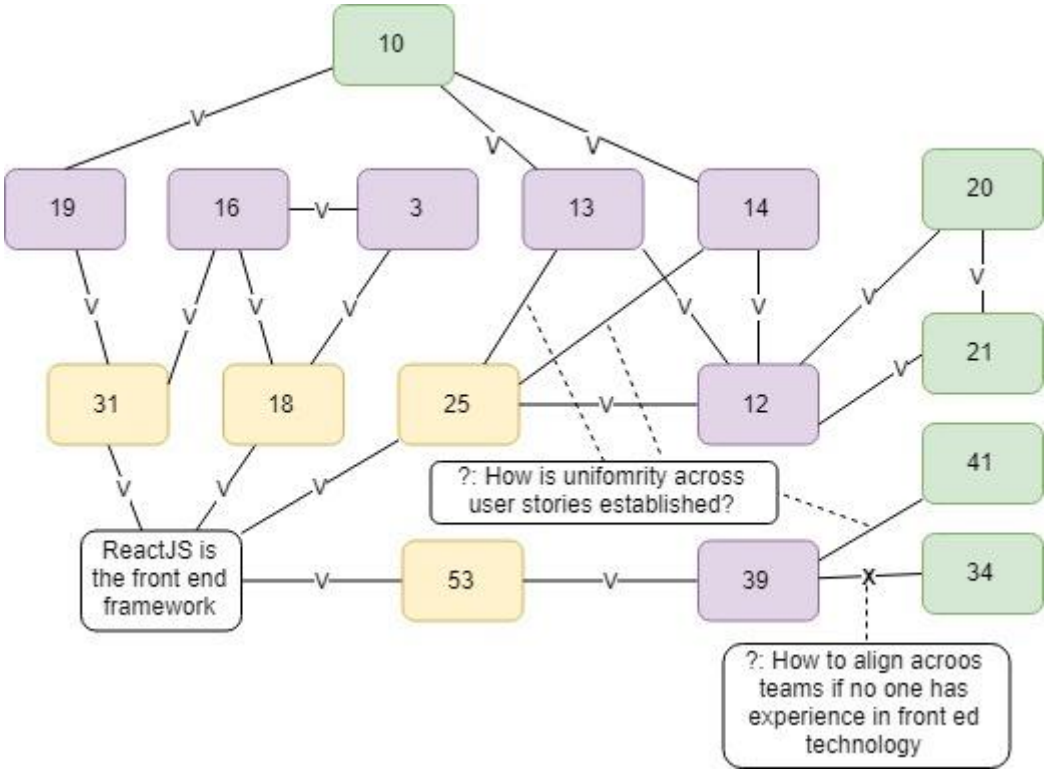

Figure 15. Reasoning after the third iteration.

Through this reasoning we discovered that there are several missing statements about how the team deals with lack of knowledge about front end development and how UI uniformity across teams is arranged. Based on these omissions, we conclude that the choice for ReactJS is not covered by the architecture.

6 Related work

This section discusses the relation of ARM with related work. The ARM is based on the definition of good architecture from DYA|Software [4], the CAFCR method [5], and the B2A Model [6]. The ARM is first published in a chapter from [4]. First we relate ARM to CAFCR [5], because CAFCR and ARM have the same origin and a similar structure. After that we compare ARM to approaches that also involve architectural reasoning. The approaches center around three themes: making architecture decisions in the development process, analyzing an architecture from a specific perspective, and capturing architecture knowledge. The conclusion is that ARM is offering a complementary perspective and can be combined quite easily with the other approaches.

The **CAFCR** [5] model is a decomposition of an architecture description into five views. The customer objectives view (what does the customer want to achieve) and the application view (how does the customer realize his goals with the system) capture the needs of the customer. The needs of the customer (what and how) provide the justification (why) for the design. The functional view describes the what of the product, including the quality attributes that concern the functionality. The how of the system is split into two separate views for reasons of stability: the conceptual view is maintained over a longer time period than the fast changing realization view. In general we can say that the application dimension of ARM is similar to the customer objectives view, application view, and functional view of CAFCR. Although we do not see the sections of the ARM model as architecture viewpoints, they could be treated as such. The conceptual view of CAFCR is similar to the system concept section of ARM. Though ARM explicitly

separates conceptual design from the realization approach. The realization view of CAFCR is in ARM spread over the sections technical design, resources technology, realization activity, and process and organization. ARM is symmetrical in its three dimensions and follows a bottom-up approach. CAFCR aims at a complete architecture and offers a framework to get there. The CAFCR reasoning flows from customer objectives to the realization view. The application of ARM is more targeted at how well a single statement fits the system and its context.

IBIS (Issue-Based Information) [7] is a method for problem mapping, issue solving, and capturing design decisions and was first published in 1970. IBIS provides a way to involve multiple stakeholders in reasoning. IBIS guides the identification, structuring, and settling of issues raised by problem-solving groups. The method follows these steps: determine the issues (formulated as questions), to find alternative positions (possible answers to the questions), and find arguments that are supporting or objective to the positions. Via the structured format of questions and answers, IBIS helps to capture the logic of decisions. Many other methods are derived from IBIS, like gIBIS (graphical IBIS) [8], Design Rationale editor (DRed [9], [10]), and PHI [11]. gIBIS, and PHI are using IBIS to create the actual design. PHI and DRed both add the concept of a repository to capture the decisions. The relation between ARM and IBIS, including its derivatives, could be seen as follows: the IBIS concepts questions, issues, positions can be seen as statements in the ARM sense. The relations between the IBIS concepts can be seen as a specialization of the consistency relation of ARM. ARM can help to clarify to position the responsibilities of stakeholders involved in an IBIS guided process. In a reasoning process via IBIS, the ARM sections can help to ask more specific questions. For example, instead of asking "what is the impact of this?", you can ask "what is the impact of this on the project plan and on the required skills of the developers?".

QoC (Questions, Options, Criteria) [12] is a method for design space analysis. QOC provides a technique to represent alternative design options and to reason about the choice between those options. QOC helps to expose assumptions, raise new questions, and challenge criteria. Besides that, it helps to find ways in which new options can capitalize on strengths and overcome the weaknesses of current options. QOC follows these steps: decompose the problem into questions, formulate already found options and create new options, decide about criteria on which the options are going to be judged, give a score to each option for each criterion, and interpret or revise unfitting criteria or choose the best one matching. Via QOC a record is built up, which includes ideas proposed during a project, previously identified task-relevant information, and missing assessments as dangling links. The process of constructing a QOC model helps to clarify vague and unarticulated ideas. The questions, options, and connections to the criteria can be shown in a tree view, which logs all reasoning and finally helps in deciding. ARM and QOC can be combined: QOC for choosing the statements and design decisions, and ARM can be used think of statements in all three dimensions and to analyze the consistency between the statements around an option or chosen decision.

SEURAT [13] is a system to capture and check the completeness and consistency of design rationales. Architectural knowledge is captured in a graph structure which is used to document and reason about architectural trade-offs. This method supports requirements traceability by incorporating functional and non-functional requirements into the argumentation of a rationale. Key elements are the decisions that need to be made, the alternative solutions for those decisions, and the arguments for and against them. Requirements are also included in the representation in two ways. Functional requirements appear on their own, including argumentation justifying the requirement, and in arguments for and against decision alternatives. Non-functional requirements, stored in an Argument Ontology, are associated with Claims that appear in arguments for and against Alternatives. This method has a tool to support the process.

The main anchor point of this method is to capture the rationale in a repository and to look for inconsistencies in the rationale and the decisions included in it. The rationale can

be analyzed to determine if there are requirements that are violated or that have not been used in the decision-making process at all. The same as with IBIS, SEURAT and ARM can be combined. The SEARAT concepts are types of statements in the ARM sense, and the relations between them are a specialization of the ARM consistency relation. The ARM sections can help in SEURAT to detect incompleteness and to explicitly distinguish stakeholder concerns from system (design) decisions.

ATAM (Architectural tradeoff analysis method) [14] is a method for evaluating architectures. It aims at identifying risks early in the life cycle, communication between stakeholders, clarifying quality attribute requirements, improving architectural documentation, and providing a documented basis for architectural decisions. The main driver of ATAM is assessing quality attributes such as modifiability, portability, extensibility, and integrability. The central concepts in ATAM are quality scenario, business driver, architectural approach, utility tree, core requirement, technical requirement, architectural property, and stakeholder. The stepwise process of this method is as follows: establish the process with the involved people, present and evaluate business drivers, present the architecture, identify architectural approaches, generate quality attribute utility tree, analyze and prioritize each architectural approach, brainstorm and prioritize scenarios, analyze architectural approaches (step 6 again with the added knowledge of the larger stakeholder community), and present the results. The main anchor point of this method is to reason about the appropriateness of the candidate architecture for the selected quality attribute. The main technique used in this method is questioning and measuring how the proposed architecture is going to achieve determined architectural scenarios. Similar to the discussion about the other approaches, the main ATAM concepts can be seen as statements in the ARM sense. Moreover, every section of the ARM can be associated with specific quality attributes. For example, usability fits the application section, portability fits in into the technical design section, and testability fits into the realization section. This helps to clarify the semantical link between system quality attributes and context quality attributes. For example, uniformity of user interface (function) helps in usability (application), which affects user efficiency (customer value). There are more scenario based evaluation methods. An overview is given in [15]. The relation of ARM with those methods is similar as the relation to ARM. ARM can help to position stakeholders and specific quality attributes and how they are related.

Building Up and Reasoning About Architectural Knowledge is the topic of [16]**.** This paper describes a use-case model for an architectural knowledge base, together with its underlying ontology of design decisions. The ontology comprises of types of decisions, attributes of decisions, relationships between decisions, and relationships with external artefacts. Architectural knowledge consists of architecture design as well as the design decisions, assumptions, context, and other factors that together determine why a particular solution is the way it is. The main point of this paper is to frame, represent, and exploit architectural knowledge, and to explore the different perspectives via use cases. The authors have modeled available architectural knowledge in a tool called the Aduna Cluster Map Viewer, which aims at ontology based visualization. Putting together ontologies, use cases, and tool support, they are able to reason about which types of architecting tasks can be supported, and how this can be done.

The authors suggest to capture architectural knowledge in a graph-like form, where the nodes represent design decisions, the vertices represent relationships between decisions, and the different use cases of architectural knowledge correspond to certain operations on this graph. Identified use cases are: 1.Incremental architectural review, 2. Review for a specific concern, 3. Evaluate impact of changes in design element or design decisions, 4. Get a rationale: given an element in the design, trace back to the decisions it is related to, 5. Study the chronology: find the sequence of design decisions taken over a time line, 6. Add a decision, 7. Spot the subversive stakeholder, 8. Clone Architectural Knowledge, 9. Integration: you want to integrate multiple systems and decide whether they fit, and 10. Detect and interpret patterns.

Unlike ARM, this method does not focus on how to spot conflicting design decisions, how to ensure consistency between design decisions, and how to find relations between design decisions. Using ARM, one can map decisions and elements to the specific ARM sections, which helps in knowing the right context of each decision. It is also possible to map the use cases themselves to the sections of the sections pf the ARM, which clarifies which stakeholders to involve.

In [17] and [18] the application of **AREL** (Architecture Rationale and Elements Linkage) [19] is discussed. AREL is an approach that incorporates design rationale templates and design-reasoning relationships. Designers can use AREL to reason about architectural design and reason how design elements relate to requirements and design models. Each design element can influence other parts of a system due to its behavior or constraints. These design elements will in turn become design concerns that influence other parts of a design. The main driver for this approach is to find causal relationships between design concerns, design decisions and design artefacts like as a database model, component model, or class design. AREL defines the following steps: specify design concerns, associate design concerns, identify design options, evaluate design options, backtrack decisions to revise design concerns. AREL, like ARM, helps to relate stakeholder concerns with design decisions. However, AREL does not explicitly mention how to identify conflicting design decisions. Furthermore, the ARM dimensions and ARM's separation between system and environment help to detect the completeness of coverage of concerns with design elements.

In [20] the authors stress the importance of **architecture decisions** and the conveying architecture decisions in **demystifying architecture**. To better convey architecture decisions, the authors suggests a template to document them. Architecture decisions should have a permanent place in software architecture development by adding architecture decisions to the ISO/IEC 42010 standard for architecture description [21]. The suggested template has the following main elements: decision, status, group, assumption, constraint, position, argument, implication, related decisions, related requirements, related artifacts, and related principles. In ARM all these concepts can be seen as statement types or values for the properties of statements. The template can be combined with ARM by either tagging the template concepts as properties of ARM statements or by adding the ARM positions of the template concepts.

In [22] the authors propose **a Model to Represent Architectural Design Rationale** to be used by architects during the architectural design definition stage. The model is proposed so that design decisions can be captured, and can be retrieved, analyzed and reused whenever necessary. The model includes concepts representing architectural artefacts, reasons, assumptions, and decisions and reasoning elements status. The model is defined in terms of UML classes to provide a clear information structure for design rationales and related concepts. The model allows to relate stakeholders' needs to requirements, to define alternative architectures, to define various architectural cases and solutions, and to define architecture design decisions. Using these definitions one can create a model for new requirements that are raised due to a made design decision. The model can be combined with ARM by seeing most concepts of the model as different types of statements. Some model elements like rationale could be additional information on a consistency relation between statements.

7 Conclusions

The ARM was introduced in Philips Research in 2003. Since then, it has proven itself useful in numerous discussions between managers, architects, and other stakeholders. In practice, the ARM is effective in workshops between stakeholders that have responsibilities in the different dimensions of the model. Often, much insight is already

gained by just positioning statements of important stakeholders. They often discover that they make statements outside their scope of control, and detect statements that have no clear justification in the context of the system.

ARM treats all statements initially as the same things. The properties of a statement make it either a decision or a given concern, constraint, or requirement, an option, or any other concept that is used by other reasoning approaches. The ARM can easily be used in combination with other reasoning methods to distinguish and involve different types of stakeholders and their concerns. Furthermore, the distinction between system and environment implies that there is inherent separation between stakeholder concerns and goals (environment), and design decisions (system or border). The distinction between system and environment also helps to position the scope of influence of the architecture and the architect. This might prevent that the architect makes decisions outside his scope of responsibilities, but also that external stakeholders force design decisions. By explicitly considering the owner of a statement, the ARM helps to involve the right people in a decision. An example, deciding to use MongoDB while the IT department has heavy investments in Oracle support, strongly impacts that IT department. If the IT department requires the use of Oracle, then the architect has to make a technical design that shows how.

ARM does not help you to come to design decision pro-actively. It helps to verify if a decision can be implemented by validating the consistency with the involved statements.

By distinguishing technical design from realization process, ARM forces to think about the realization of a design in the context of the development organization. Although one might argue that an architecture should not be determined by the development organization's characteristics, in practice many things can go wrong if a technical design cannot be implemented by the available people and their processes.

By explicitly reasoning along the application dimension (like in CAFCR), the ARM puts the customer's functional needs in a central position. These needs are the main reason of existence of a system are therefore very important.

The ARM does not offer explicit integration with ISO42010 and does not explicitly handle the design rationale as a separate type of statement. But on any consistency relation between two statements, a rationale can be written to express why the statements are consistent with each other.

The ARM currently lacks tool support, especially to record a reasoning. When the ARM is combined with an approach that has tool support, the sections of the ARM could be added as tags to the statements that are stored in the involved repository.

References

[1] „ISO/IEC 25010:2011: Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models," ISO, Geneva, 2011.

[2] M. v. d. B. M. van Steenbergen, Building an Enterprise Architecture Practice, Springer, 2006.

[3] A. Watkins en I. Stratenus, Crowdocracy: The End of Politics, Urbane Publications, 2016.

[4] R. S. R. Deckers, DYA Software, Architectuuraanpak voor bedrijfskritische applicaties, Kleine Uijl, 2010.

[5] G. J. Muller, CAFCR: A Multi-view Method for Embedded Systems Architecting. Balancing Genericity and Specificity, 2004.

[6] R. Deckers en F. v. d. Berk, The B2A Model, Architecture as a Business Enabler, Philips Electronics Nederland B.V., Atos Origin, Sioux B.V., 2003.

[7] D. Noble, N. Rittel en H. W.J., „Issue-Based Information Systems for Design," in *Computing in Design Education [ACADIA Conference Proceedings]*, Ann Arbor (Michigan / USA), 1988.

[8] J. Conklin en M. L. Begeman, „gIBIS: a hypertext tool for team design deliberation," in *Proceedings of the ACM conference on Hypertext*, Chapel Hill, North Carolina, USA, 1987.

[9] R. Bracewell en K. Wallace, „A tool for capturing design rationale," in *International Conference on Engineering Design*, Stockhom, 2003.

[10] K. W. M. M. D. K. Rob Bracewell, „Capturing design rationale," *Computer-Aided Design,* nr. volune 41, Issue 3, pp. 173-186, 2009.

[11] R. J. McCall, „PHI: a conceptual foundation for design hypermedia," *Design Studies,* nr. Volume 12, Issue 1, pp. 30-41, 1991.

[12] A. MacLean, R. M. Young, V. M. Bellotti en T. P. Moran, „Questions, Options, and Criteria: Elements of Design Space Analysis," *Human–Computer Interaction,* nr. vol. 6, numer 3-4, pp. 201-250, 1991.

[13] J. Burge en D. Brown, „Supporting Requirements Traceability with Rationale," in *Proceedings of International Symposium on Grand Challenges in Traceability (GTC)*, 2007.

[14] R. Kazman, M. Klein en P. Clements, „ATAM: Method for Architecture Evaluation," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2000.

[15] M. Ionita, D. K. Hammer en H. Obbink, „Scenario-Based Software Architecture Evaluation Methods: An Overview," in *Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering*, 2002.

[16] P. Kruchten, P. Lago en H. van Vliet, „Building Up and Reasoning About Architectural Knowledge," in *International conference Quality of Software Architectures*, Västerås, Sweden, 2006.

[17] J. H. a. R. V. A. Tang, „Software Architecture Design Reasoning: A Case for Improved Methodology Support," *IEEE Software,* nr. volume 26, no 2, pp. 43-49, 2009.

[18] A. V. H. Tang, „Software architecture design reasoning," in *Software Architecture Knowledge Management*, Heidelberg, Springer, 2009, pp. 155-174.

[19] A. Tang en J. H. Yan Jin, „A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software,* nr. volume 80, issue 6, pp. 918-934, 2007.

[20] J. Tyree en A. Akerman, „Architecture decisions: demystifying architecture," *IEEE Software,* nr. vol. 22, no 2, pp. 19-27, 2005.

[21] *ISO/IEC/IEEE 42010 Systems and software engineering - Architecture description,* New York: ISO/IEC/IEEE, 2011.

[22] M. C. Carignano, S. Gonnet en H. Leone, „A model to represent architectural design rationale," in *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, Cambridge, 2009.

[23] G. Abowd, L. Bass, R. Kazman en M. Webb, „SAAM: A Method for Analyzing the Properties of Software Architectures," Software Engineering Institute, 2007.

[24] H. van Vliet en A. Tang, „Decision making in software architecture," *Journal of Systems and Software,* nr. vol. 117, pp. 638-644, 2016.

[25] G. Yue, J. Liu, Y. Hou en Y. Luo, „Design Rationale Knowledge Management: A Survey," in *Cooperative Design, Visualization, and Engineering*, pringer International Publishing, 2018, pp. 245-253.

[26] G. C. R. K. P. K. Davide Falessi, „Decision-making techniques for software architecture design: A comparative survey," *ACM Comput. Surv.,* nr. vol. 43, number 4, pp. 1-28, 2011.