Understanding and Comparing
Architectural Analysis Models

Reinder Kamphorst

supervised by

Anton Jansen, ABB Corporate Research (formerly University of Groningen, The Netherlands)
Paris Avgeriou, University of Groningen, The Netherlands
Albert-Jan Boonstra, Netherlands Institute for Radio Astronomy (ASTRON)

17 July 2009
Abstract

An analysis model provides insight into the properties (e.g., the cost) of a system if it were built according to a certain design, with certain requirements. It performs calculations that take as input some pre-determined properties of the system. The output is a set of values that represent resulting properties of the system and the resources needed to realize the system. Results of analysis models are used to resolve design issues: they provide part of the rationale for design decisions.

At the start of the development of a system, a simple analysis model is sufficient. At later stages, more complex analysis models are needed. Thus, multiple analysis models with different levels of complexity, different approaches and different levels of detail are in use during the life cycle of a system. This also occurs when two systems are integrated into a larger system: the analysis models of the former two may differ significantly from their corresponding parts of the latter.

Multiple analysis models that yield different results are a problem, because it becomes unclear which result a design decision should be based on. Understanding each of the analysis models and identifying their key differences requires great effort. This problem is encountered by ASTRON (an institute that develops radio telescopes) in the joint development of the Square Kilometre Array (SKA) with multiple other institutes. Tooling is desired that helps analysts to understand each of the analysis models, and to identify differences between them more easily and quickly.

This master’s thesis looks into what information (knowledge) an analyst needs to understand an analysis model, how this knowledge is acquired, and how knowledge of multiple analysis models can be used to identify differences. Subsequently, it determines what is needed to make understanding analysis models and comparing them easier, faster and less error-prone.

The pyQAM tool is an implementation of the found solutions in the context of the ASTRON case. It enables an analyst to analyze an analysis model in Python more easily and quickly by extracting data from the analysis model and visualizing it. PyQAM stores the data into a repository for further analysis in combination with Architectural Knowledge extracted by other tools. PyQAM is validated in a controlled experiment.

PyQAM is designed as part of the Knowledge Architect, a tool suite for capturing Architectural Knowledge (AK) from artifacts (documents, source code, ...) belonging to a system and its design. The purpose of the Knowledge Architect is to capture AK and share it among stakeholders of a system design. AK embedded in analysis models is captured in terms of a domain model, and can be shared among all stakeholders of the system.
Acknowledgements

This master thesis has provided me with many joyful hours of work, although it took a long time to finish and at times it was hard to keep on going. Nevertheless, here it is! And I couldn’t have done it without the help of the following people.

First of all, I want to thank Anton Jansen for the effort he has put into supervising me, even when he had to leave his new job in for Sweden, and I started on my own full-time job here in Groningen. Our bi-weekly chats over the Internet have helped me to keep focused and to not give up the hope that I would eventually finish this thesis.

Thanks go to the SEARCH group at the University of Groningen; in particular, to Paris Avgeriou, for giving lots of advice, for being very patient and for giving me compliments when I needed them. To Tjaard de Vries, whose thesis has served as an example and whose insights have helped me do a better job.

ASTRON has provided me with a case study and the time and means to work on it, which I am grateful for. In particular, I would like to thank Albert-Jan Boonstra, for supervising me during my internship at ASTRON, and Ronald de Wild, for the many chats we had in the office we shared. And Dess Small from JIVE, who has been a major help in commuting between Groningen and Dwingeloo by letting me ride with him – and being great company during those rides.

I want to thank my family and friends, for they have also been very supportive: Martijn Loot, for doing a final and very detailed review; my uncle Jan, for giving me advice on statistical matters; and my parents, brother and sisters for tolerating me while talking on and on about computers and software. I am especially grateful to my mother, who has home-schooled me and my siblings during the years we lived in Africa, and has made it possible that I have come this far.

Finally, I thank my wife Kirsten, for making it possible for me to work long hours on this thesis, for being endlessly patient, and for encouraging me whenever I needed it. For loving and caring, and for never doubting me.

Reinder Kamphorst
Groningen, 17 July 2009
Contents

1 Introduction ............................................. 1
   1.1 Context: ASTRON .................................. 2
       1.1.1 The Square Kilometer Array .................. 2
       1.1.2 The Aperture Array ......................... 3
       1.1.3 Analysis Models ............................. 3
   1.2 Context: GRIFFIN .................................. 4
   1.3 Thesis outline .................................... 5

2 Problem Statement .................................. 7

3 Analysis ............................................. 9
   3.1 Introduction .................................... 9
   3.2 Causes for different analyses ................... 9
   3.3 Finding differences ............................. 10
       3.3.1 Knowledge to understand an analysis model .. 10
       3.3.2 Knowledge to compare two analysis models  .. 13
       3.3.3 Aspects of comparing analysis models ...... 14
       3.3.4 Significance of values ..................... 15
       3.3.5 Prerequisites for comparison ................ 17
   3.4 Easier, less error-prone and quicker ............ 17
       3.4.1 Tool: Knowledge Architect and pyQAM ...... 18

4 pyQAM ............................................. 21
   4.1 Introduction ................................... 21
   4.2 Stakeholders ................................... 21
       4.2.1 GRIFFIN .................................. 21
       4.2.2 ASTRON .................................. 22
   4.3 Requirements ................................... 23
       4.3.1 Use cases .................................. 23
       4.3.2 Requirements ............................... 25
   4.4 Design .......................................... 28
       4.4.1 Components ................................ 28
       4.4.2 Allocation ................................ 29
       4.4.3 Logical design of pyQAM ................. 31

5 Capturing Analysis Model Data ..................... 33
   5.1 Introduction ................................... 33
   5.2 Excel workbook .................................. 33
   5.3 SKAcoast ....................................... 35
       5.3.1 What to capture ............................ 36
       5.3.2 Strategy ................................... 37
       5.3.3 Design and implementation ................. 43
   5.4 SKAsim ........................................ 49
       5.4.1 Capturing data from SKAsim ............... 49
   5.5 Domain models ................................... 50
# CONTENTS

5.5.1 Analysis Function, System Parameter, Number ................................................. 52
5.5.2 Value, Analysis Output, Analysis Result ......................................................... 53
5.5.3 Component, Component Instance ................................................................. 53
5.5.4 Scenario ............................................................................................................ 53
5.5.5 Artifact ............................................................................................................ 54

6 Visualizing Analysis Models .................................................................................. 55
  6.1 Introduction ......................................................................................................... 55
  6.2 Common principles ............................................................................................ 56
  6.3 Graph visualization ............................................................................................ 56
    6.3.1 Dependency graph of System Parameters ..................................................... 56
    6.3.2 Adding Analysis Functions ......................................................................... 58
    6.3.3 Adding Numbers ......................................................................................... 60
    6.3.4 Adding Components and Component Instances ........................................ 61
  6.4 System Parameter Lists ..................................................................................... 61
    6.4.1 Reverse dependency analysis (top-down) ..................................................... 63
    6.4.2 Forward dependency analysis (bottom-up) ................................................... 63
  6.5 Table of Numbers ............................................................................................. 63
    6.5.1 Tables in Excel .......................................................................................... 64
    6.5.2 Table of Numbers: one step ....................................................................... 66
    6.5.3 Table of Numbers: two steps ..................................................................... 67
    6.5.4 Table of Numbers: multiple steps ............................................................... 68

7 Comparing Analysis Models .................................................................................. 71
  7.1 Introduction ......................................................................................................... 71
  7.2 Mapping domain models ................................................................................... 71
  7.3 Mapping semantics ............................................................................................ 72
    7.3.1 Deriving mappings for System Parameters ............................................... 72
    7.3.2 Deriving mappings for Analysis Functions ............................................... 73
  7.4 Visualization ....................................................................................................... 74
    7.4.1 Side by side view ....................................................................................... 74
    7.4.2 Combined view ("diffl") ............................................................................ 74

8 Validation .............................................................................................................. 77
  8.1 Introduction ......................................................................................................... 77
  8.2 Use cases and requirements ................................................................................ 77
  8.3 Experiment .......................................................................................................... 79
    8.3.1 Experiment Setup ....................................................................................... 79
    8.3.2 Quantitative results .................................................................................... 85
    8.3.3 Qualitative results ...................................................................................... 92
  8.4 Conclusion .......................................................................................................... 93

9 Related Work ......................................................................................................... 95
  9.1 Introduction ......................................................................................................... 95
  9.2 Architectural Knowledge ................................................................................... 95
  9.3 Architecture Evaluation Methods ....................................................................... 97
  9.4 Reverse Engineering .......................................................................................... 98
  9.5 Model Based Development ............................................................................... 100

10 Conclusions, Future Work .................................................................................. 103
  10.1 Conclusions ...................................................................................................... 103
  10.2 Future Work ..................................................................................................... 105

Bibliography ............................................................................................................. 107
Chapter 1

Introduction

One of the questions that lacks a definitive answer in the field of systems and software engineering, is how knowledge about a system should be retained, transferred, and maintained. This knowledge is the knowledge gathered by people who work with the system during its design, implementation and maintenance stages. People generally don’t work on a system during its entire lifetime; for example, a system designer typically leaves a project once a system is complete and running. Unless the system designer documented everything she knows, valuable knowledge about the system leaves the project once the system designer does. If at some point the system needs to be modified, either clumsy (re)design decisions are made, or a lot of time is invested to reinvent lost knowledge. This is not only true for system designers who leave a project: people who implement, maintain or use a system also gather knowledge that is valuable to everyone who has to work with the system.

The loss of knowledge is called knowledge evaporation [1] and is an almost inevitable fact of life. The one solution to it is that everyone involved with a system writes every bit of knowledge about it down. However, if not collected in a structured manner, the body of information may become impenetrable. Moreover, people generally don’t like spending time on writing down every little fact about the system they work with.

Knowledge evaporation is not the only problem. On large development projects, tasks are divided among members of a team; each member has his or her own responsibilities and is an expert on only part of the system. This is generally a good thing: some systems are just too large or complex for one person to know every detail about. A consequence is that if one of the experts doesn’t record a detail that only she knows about, this detail is lost and is not easily recovered. If another expert is responsible for the design or implementation of a different part of the system that influences or is influenced by the unrecorded detail, the completed system may contain flaws.

To reduce this knowledge evaporation, we can explicitly gather Architectural Knowledge during the process of system design, implementation, and possibly even during maintenance (when the system has been deployed). Architectural Knowledge consists of a description of the design and the Design Decisions made. A Design Decision describes not only a decision that is made, but also its rationale, the requirements it fulfills, the considered alternatives to the decision, additional requirements that follow from the decision and prescriptions for subsequent design decisions.

A reason to choose for one alternative over the other in a design decision is because it causes certain properties of the system, called quality attributes, to be better. For large systems, the amount of design information to consider when reasoning about quality attributes is huge. So huge, that even if well-ordered and complete, it is difficult to give estimates for quality attributes such as cost, throughput, reaction time, and others. A solution is to develop analysis models of the system design: quantitative estimates for a small number of quality attributes are input to such an analysis model, which then calculates values for other quality attributes. If one or more of the calculated values are unacceptable, either the input values (estimates for quality attributes) can be tweaked, or the design can be changed. An analysis model can be used as an argument in discussions about design issues, and often plays a vital role in the reasoning behind design decisions.

In some cases, the system is so large that it is developed by multiple engineering teams or even multiple organizations. This is the case with the Square Kilometer Array (or SKA, see section
1.1. Context: ASTRON

ASTRON is the Netherlands foundation for radio astronomy. It is an institute that develops and maintains equipment for astronomers, and does astronomical research. Its main line of development is in the field of radio astronomy: it has developed three important radio telescopes in western Europe: the Dwingeloo radio telescope (not operational anymore for scientific research), the Westerbork Synthesis Radio Telescope (WSRT) and the Low Frequency Array (LOFAR); it also maintains and operates these. A new development on this path is the development of the Square Kilometer Array, in which ASTRON is a participating institute.

1.1.1 The Square Kilometer Array

The Square Kilometer Array (SKA) project is an international effort to construct the world’s most sensitive radio telescope, scheduled to be operational around the year 2020. The project is a collaboration of 27 institutes, spread over 19 countries. The SKA will be composed of a large number of antennas, half of them in a core station with a diameter of 5 km and the other half of them spread out over stations at greater distances from the core; the total diameter of the entire system will be about 3000 km. The data that is generated by all these antennas can be computationally combined into one picture of (a portion of) the sky.

Three types of antennas may be used for the SKA:

1. Dipole arrays, to observe radio wavelengths from about 100 MHz to about 300 MHz
2. Small parabolic dishes, to observe wavelengths from 500 MHz to 25 GHz
3. Aperture array tiles to observe wavelengths from 300 MHz to 1 GHz
Chapter 1. Introduction

1.1. Context: ASTRON

For an impression of each of the antennas, see figure 1.1. Of course, these are not actual SKA antennas: the SKA itself has not been built yet. In figure 1.1a, a dipole antenna from ASTRON's LOFAR project is depicted; 1.1b shows a parabolic dish from the Allen Telescope Array; 1.1c is a prototype aperture array tile at ASTRON.

SKA is currently being designed. A group of SKA participants have received funding from the European Union to conduct the European Square Kilometer Array Design Studies (SKADS) and subsequently for the PrepSKA project. These projects will eventually produce a SKA development plan and design specification.

1.1.2 The Aperture Array

ASTRON is one of the institutes with expertise on the Aperture Array (AA) (figure 1.1c), one of the types of antenna to be used for the SKA. This type of antenna is very new: it has not yet been used on a large scale in existing radio telescopes. The Aperture Array consists of a collection of tiles, each of them hosting about a hundred antenna elements. Each tile sits on the ground, receiving radio signals with its elements. The signal from one element alone is not very useful, as it receives radiation from all directions. But when all signals from a tile are combined, so-called beam forming can be applied: it is possible to measure radiation that comes from specific directions, and to create a picture of that direction in the sky; albeit one that is not of a very high quality. The resolution is improved by using a clusters of tiles at large distances from each other; their results can be combined through interferometry and aperture synthesis into a picture of high resolution. Interferometry is not new; it is quite common in today's radio telescopes (such as the WSRT mentioned earlier).

In contrast to "classical" radio telescopes, which consist of dishes that can be pointed in a direction to do measurements of one spot in the sky, AA tiles do not contain moving elements. They are pointed to the sky electronically through beam forming techniques, which makes them cheaper to maintain. The cost of a classical dish is determined by the steel structure and the mechanics that are needed to support the dish and to point it to the sky; AA tiles are small and don’t need to be pointed mechanically, which makes them potentially cheap to produce. A possible downside of the AA tile is that it may have a higher energy consumption: instead of the "beam" being formed analogously by a parabolic dish, the beam needs to be digitally composed of the signals that are received. However, whether this really results in a significantly higher energy consumption is yet to be determined.

1.1.3 Analysis Models

The Aperture Array is developed in a cooperation of ASTRON and a number of other institutes. Each of the institutes have their own ideas on which configuration to use, how to estimate the associated cost, etcetera. At some point, the institutes have to reach a compromise on these matters.

To compare notes, a cost model is composed by members of the institutes involved. Such a cost model computes the cost of a system design at a certain point in time, but also other quality attributes that are important to consider. Because not only the cost is modeled, the more generic term analysis model will be used in this thesis.

The outcomes of the analysis model for different system designs are an important source of information that is used when the ultimate decisions about the design of the AA are made. By adopting one analysis model that all stakeholders agree on, the institutes can uniformly compare their designs. For the aperture array, three analysis models have been made, each of them with improvements on the previous.

The first analysis model that was created in Microsoft Excel ("Excel model" for short). Excel is a spreadsheet program, providing an intuitive way of presenting an analysis model in tables. The Excel model is divided in a number of sheets in one workbook; each sheet covers part of the system. One sheet gives a summary with results from the other sheets. A big advantage of Excel is that every domain expert that is involved knows how to work with it. A disadvantage of the Excel model is that it easily becomes a mess: Excel gives the user a lot of freedom in specifying a calculation.
1.2. CONTEXT: GRIFFIN

After the Excel model, the SKAcost analysis model was introduced. SKAcost models the system design in an object-oriented Python program: components in the design are classes in the analysis model. SKAcost is a more structured approach to model the aperture array than the Excel model: for example, it allows parts of the analysis model to be re-used, and it automatically performs sanity checks. However, it requires engineers that work with it to be able to use Python in order to do analysis with it: if a system design to be modeled contains a new component, this component needs to be implemented in SKAcost by creating a class for the component.

The third model, SKAsim, is a Python program that separates the system design from the engine that performs calculations. Most components a system design consists of can be provided as XML files that describe the component; these XML files can be generated with a user interface that is easy to use. The obvious advantage is that users of this model don’t have to be Python experts: they simply define the system design they want to analyze by generating the XML files with an easy to use user interface, and click a button to calculate the results.

Although ASTRON is one of the leading institutes on aperture array development, the analysis models (especially the Python models) have mainly been the product of another institute. The first (Excel) analysis model was made with input and monitoring by ASTRON. The knowledge captured in the Excel model was used to implement the AA design in SKAcost, which had already been used for other SKA related designs. This happened with little contribution of ASTRON; the same is true for the transfer from SKAcost to its successor, SKAsim.

Still, it was important for ASTRON to be able to validate the SKAcost and SKAsim analysis models; after all, important decisions were to be made based on one of these. To do this validation, domain experts would have to spend precious time delving into the Python code of the models, to identify differences with the Excel model and to point out possible flaws. To enable them to do this in a timely fashion and without errors, ASTRON wanted a tool that could produce visualizations of all the models, and compare them to each other.

1.2 Context: GRIFFIN

GRIFFIN stands for GRId For inFormation about architectural knowledge. It is a project sponsored by the JACQUARD program of the Netherlands Organization of Scientific Research (NWO), and is hosted at two universities, the University of Groningen (RuG) and the VU University at Amsterdam (VU). A number of knowledge institutes participate in GRIFFIN; ASTRON is one of these.

GRIFFIN research at the RuG has spawned a tool suite that has been baptized the Knowledge Architect tool suite. It makes use of the Sesame semantic repository to store architectural knowledge. Currently, the tool suite features the following tools:

- The Word Client, which captures knowledge entities and the relations between them from system design documents made with Microsoft Word
- The Knowledge Explorer, which can visualize architectural knowledge stored in the repository
- The Excel Client and Excel Knowledge Extractor, which visualize and capture knowledge entities in analysis models made with Microsoft Excel
- The tool of which the development is described in this thesis: the Python Client or pyQAM (python Quantitative Analysis Model) tool, the tool developed for analysis models made with the Python programming language such as SKAcost and SKAsim.

GRIFFIN’s interest was similar to ASTRON’s, albeit in a more general way: they wanted a tool that could capture the same information the Excel client does, and store it into the same repository as the data captured with Excel. Data from Python analysis models being stored in a similar format next to data from Excel analysis models, it would become possible to research ways in which two analysis models from entirely different sources can be compared; more so because the three analysis models provided by ASTRON are supposed to be modeling the same system design.
1.3 Thesis outline

The two chapters following this one, Problem Statement and Analysis, respectively elaborate on the problems found and the analysis thereof.

In the three subsequent chapters, the solution presented in this thesis and implemented in the pyQAM tool is rolled out. Capturing Analysis Model Data describes how data from each of the three analysis models is captured; the largest chunk of this chapter is on capturing data from Python analysis models, as the Excel tool is already extensively described in a previous master thesis [2]. Visualizing Analysis Models then shows the considered visualizations of single analysis models: that is, visualizations that take only one analysis model into account. Comparing Analysis Models is about visualizations that can show two analysis models at a time, showing differences and similarities. Unfortunately, the amount of work spent on capturing data from Python models and visualizing it took more time than expected; therefore, Comparing Analysis Models contains the fruit of many hours of thought and literature research, but is not backed by an implemented prototype.

The pyQAM tool has been tested in a validation experiment with both domain experts and laymen on the SKA Aperture Array design; this experiment and its results are described in the Validation chapter. Related Work gives an overview of work of others that is related to the work described in this thesis; finally, Conclusions and Future work concludes this thesis and gives an overview of possible future developments.
Chapter 2

Problem Statement

An analysis model's purpose is to provide insight into the properties (e.g., the cost) of a system if it were built according to a certain design. It performs a calculation that takes as input some properties of the design that are assumptions; these are either fixed values that cannot be changed, or desired system attributes\(^1\). Its result is a set of values that represent other properties calculated from these assumptions.

Analysis models can come in many forms: the level of detail needed, the complexity of the design to model, and the preference of the analysis model authors are among the reasons to choose a particular form. For example, when we consider a recipe (the system design for a meal), a grocery list might suffice as analysis model; whereas if we want to build a bridge, a list of things to buy is not detailed enough, and will become too long (complex) to manage, to change, or to check for errors. In the latter case, a more sophisticated analysis model is needed; one that allows to derive amounts of materials needed from a handful of assumptions, calculates how the bridge performs with respect to certain quality attributes, and is set up in such a way that errors surface quickly.

It is not always easy to decide on how elaborate the analysis model needs to be for a system design, and in what form it should be implemented. At the start of the development of a system, a simple analysis model is often enough; while at a later stage, the increased number of known details calls for more sophistication in the analysis model. Thus, multiple analysis models that use different approaches and levels of detail may be in use during the life cycle of a system. These may not give the same results; these differences need to be justified.

A similar problem occurs when two different systems are developed independently, and then are selected to become subsystems of a bigger system. Not only do the subsystems need to be integrated into the new system, their analysis models also need to be integrated into the new system's analysis model. The subsystems may use different approaches in their analysis models, and the enclosing system might yet use another approach. It is then almost inevitable that the latter yields results that are different from results of the former two; again, these differences need to be justified.

In both cases, there are two (or more) analysis models of which the key differences need to be identified. In order to find these differences, an analyst needs to gain sufficient knowledge about each of the analysis models, and then somehow compare the two analysis models to each other. How analysts tend to do this, and how we can help them do this better and more efficiently, is the subject of this thesis. The problem decomposes into the following research questions.

Research Question 1 To justify differing results for analysis models that analyze the same system, one has to investigate the internals of both models. The goal is to identify the root differences: the key differences between two analysis models that cause one or more of their results to be different. There can be a number of causes for these root differences: examples are the aforementioned difference in analysis detail and difference in approach. The first research question to be answered is:

\(^1\)The word *assumptions* encompasses more than just values that are input to an analysis model. However, in this thesis an *assumption* refers to a value that is input to an analysis model, unless stated otherwise.
What are the causes for different analyses of the same system?

**Research Question 2** Comparing analysis models to find the so-called key differences is not a trivial task. Not only can analysis models have different designs and approaches, they can be in entirely different formats; an example is the case where one analysis model is defined with formulas in spreadsheet and the other is implemented in an object-oriented program. Even with a firm grasp of both formats and designs of the analysis models, it can be a time-consuming task to determine the important differences between analysis models. This yields the second research question:

*How does one find key differences between different analyses of the same system?*

**Research Question 3** The answer to the second research question consists of a number of techniques and approaches that are generally employed by analysts. The bigger and more complex an analysis model is, the longer it takes for an analyst to apply these techniques, and the more errors she makes. Therefore, the last and most important research question is:

*How can we make finding differences between different analyses of the same system easier, less error-prone and quicker?*
Chapter 3

Analysis

3.1 Introduction

Three research questions have been formulated in the last chapter. In this chapter, these are decomposed into sub-questions and further analyzed.

Section 3.2 elaborates on the causes for differences between analysis models, subject of the first research question. The second research question, on how one finds these differences, is treated in section 3.3.

The last research question, involving how we can help an analyst to do a better job at comparing analysis models, was answered with the development of a software tool called pyQAM. The ideas that led to the development of pyQAM are discussed in section 3.4; pyQAM itself is discussed in subsequent chapters.

3.2 Causes for different analyses

What are the causes for different analyses of the same system?

The word *causes* can be interpreted in a number of ways. For example, a particular value or formula that differs between analysis models can be the cause of results being different. A cause can also be external to the analysis model itself, lying in the decisions made prior to and during the creation of the analysis model. The former kind of cause is what an analyst tries to find when comparing two analysis models; the latter is what this section is about.

**Level of detail** At the start of development of a system, most of its details have not yet been established. An analysis model created at the start of development of a system is typically less elaborate and detailed than one created at a later stage. This is reflected in the results of these analysis models; the results of the former are more coarse-grained than the latter’s results. An example is given by the SKADS case (see section 1.1.2): three analysis models were developed in different stages of the project. The first of these, the Excel analysis model, is less detailed than the later SKAcost and SKAsim analysis models.

**Assumptions** If different analysis models make different assumptions, these are obvious causes for different results. Assumptions change during the development of a system because they are better known; therefore, assumptions of "earlier" analysis models are different from "later" models. However, one might also be comparing two analysis models that model different scenarios; in that case, different sets of assumptions reflect different desired properties for each of the scenarios for the system design. Different results for the analysis models then reflect properties that need to be optimized by the system design.

**Results of one are Assumptions of the other** One analysis model might have as input some values that are calculated values in another analysis model. In other words, the analysis models
not only makes different assumptions, but they make assumptions about different parts of the system that is modeled. This issue is related to the next, about different formulas being used: in this case, parts of the calculations of one model are inverse to the ones in the other.

### Formulas

Two analysis models can calculate the same thing, using entirely different ways to do so. One possibility is that there are different formulas to calculate the same value; but mostly, the formulas or calculations employed by an analysis model are subject to what is possible in the format in which it is implemented. For example, a Python program can have program loops, whereas this is not possible in Excel.

## 3.3 Finding differences

*How does one find key differences between different analyses of the same system?*

In order to find differences between analyses of the same system, first of all, an analyst has to have a certain level of knowledge of the individual analysis models. What knowledge one needs to gain about an individual analysis model, is the first issue in finding the answer to this research question. The second issue is which knowledge is needed to find differences between the analysis models. Summarizing:

1. **What knowledge is needed to understand an individual analysis model?**
2. **What knowledge is needed to compare two analysis models that one understands?**

These two questions are about the knowledge one needs to gain to be able to understand and compare analysis models. *How* this knowledge is acquired, is the subject of the next question:

3. **What are the aspects of acquiring knowledge of an individual analysis model and of comparing two analysis models?**

When comparing two analysis models, a lot of differences are bound to surface. Most of these differences are insignificant. These differences can be stated in terms of corresponding values used and calculated in the analysis models that are different; if a value in an analysis model is significant and it differs from its corresponding value in the other analysis model, the difference is significant. Therefore, in order to have a comparison of two analysis models produce a meaningful result, we need an answer to the following question:

4. **What are significant values in an analysis model?**

Finally, with the answers to these four questions, it is possible to state the skills and knowledge the analyst needs to perform a successful comparison of analysis models:

5. **What are the prerequisites for successfully comparing analysis models?**

The following five subsections elaborate further on each of these questions.

### 3.3.1 Knowledge to understand an analysis model

*What knowledge is needed to understand an individual analysis model?*

An analysis model consists of calculations that calculate quantitative values for quality attributes of the modeled system design. Each calculation or formula takes as input a set of values, and produces one or more values. Values that are output of one calculation can be input to another calculation.

A value that is involved in calculations of an analysis model also has a **semantic value**: that is, it has a meaning to someone interpreting the analysis model. Distinct values in the analysis model can have the same semantic value. For example, when the analysis model models a component that occurs multiple times in the design, the calculation for this component may be repeatedly executed, each time involving a set of different values. Thus, there are multiple values in the analysis model (either assumptions or results from calculations) that have the same semantic value.
The semantic value of a value is closely related to the calculation it is produced by, if any. If the same calculation is executed twice, result values from the first run have semantic values corresponding to the semantic values of the results of the second run. Furthermore, a calculation connects the semantic values of its output values to the semantic values of its input values: the former are dependent on the latter.

The knowledge needed to understand an individual analysis model thus consists of:

- Semantic values in the analysis model
- How these semantic values depend on each other through calculations.

A domain model that formally defines these in the larger context of a system development process is described by Tjaard de Vries [2]. The next section describes the parts of this domain model that encompass the knowledge needed to understand an individual analysis model.

### 3.3.1.1 Domain Model

![Diagram](image.png)

Figure 3.1: Knowledge Entities in the Excel Client

In figure 3.1, the basic building block of the domain model is depicted. Every piece of knowledge to express is represented with a Knowledge Entity (KE). Every KE is part of a fragment of an artifact, the Artifact Fragment. An Artifact Fragment contains information about a part of an artifact that supposedly contains a definition for the KE; as such information is different for each type of artifact, there are Word Fragment, Excel Fragment and Python Fragment subclasses. In turn, an Artifact Fragment is part of an Artifact; hence, a KE is part of the Artifact the Artifact Fragment it is part of.

The KE class is the super class of all kinds of information that is contained in an artifact. Figure 3.2 on the following page shows which subclasses of KE there are, and which relations they have; the ones that are greyed out are the ones that represent quantitative analysis, the kind of analysis that is present in analysis models. Note that in this figure, every class is a subclass of KE except Namespace.

In the remainder of this chapter and all the next chapters, whenever an entity from the domain model described here is referred to, its name will be capitalized and in italic font; e.g. Analysis Function, System Parameter.

**Analysis Model** The Analysis Model class is the class that represents an analysis model. It contains all KEs that represent something in that analysis model. Typically, an analysis model calculates quality attributes for a design scenario, which is represented by the analyzes relation from Analysis Model to Scenario. Furthermore, an Analysis Model lives in a certain Namespace, to be able to reason about KEs from different sources.
3.3. FINDING DIFFERENCES

CHAPTER 3. ANALYSIS

Figure 3.2: Domain Model for the Excel Client
CHAPTER 3. ANALYSIS 3.3. FINDING DIFFERENCES

**Number** Every value that is used or calculated in an analysis model is represented by the *Number* class in the domain model. Every *Number* is part of the *Analysis Model*.

A *Number* can have different roles in the analysis model: it can be a value that is input to the model, the output of a calculation, or a special output that is considered to be an important result of the analysis model. These roles are represented by respectively *Value* (not to be confused with the word “value”, that is used in a broader sense), *Analysis Output* and *Analysis Result*; these are all subclasses to *Number*.

Because the *Scenario* that is analyzed by the *Analysis Model* determines what the input values to the analysis model are, there is a *sets* relation from *Scenario* to *Value*. An *Analysis Result* is a point of connection between the quantitative and qualitative parts of the domain model: it influences one or more design alternatives with its *pro* and *con* relations to *Alternative*.

In practice, a *Number* is referred to by its semantic value. For example, if the kinetic energy is calculated of a person weighing 80 kg that travels at 2 meters per second, the *Number* that represents the value 80 is referred to as "person's mass", the *Number* for value 2 is referred to as "person's velocity". Multiple *Numbers* may have the same semantic value, for example when there is more than one calculation in the analysis model that takes "person's mass" as its input. The semantic value of a *Number* is represented by *System Parameter*. *Number* has an *is of type* relation to *System Parameter*.

**System Parameter** A *System Parameter* represents *Numbers* with a certain meaning, such as "person’s velocity" in the previous example. Elaborating on this example, there might be multiple persons for whom the kinetic energy needs to be calculated; each of these persons has an associated "person’s velocity". For each of the persons, there is a *Number* representing the "person's velocity"; however, all these numbers are of the same *type* (semantic value), i.e., multiple numbers can have a *typed by* relation to the same *System Parameter*.

A *System Parameter* can be aggregated into a *Component*, which is also considered a *System Parameter* in the domain model (*Component* is a subclass of *System Parameter*). *Component* represents a grouping of *System Parameters* that follows from the analysis model. For example, an analysis model might consist of several independent parts that are tied together by another part; each of these parts maps to a *Component*.

**Analysis Function** The *Analysis Function* entity represents calculations in an analysis model. The domain model defines that *Numbers* that are calculated, and are *typed by the same System Parameter*, are always calculated by the same *Analysis Function*; and one *Analysis Function* takes a set of *Numbers* as input that are always typed by the same set of *System Parameters*. Hence the relation calculated by from *System Parameter* to *Analysis Function* and takes as input from *Analysis Function* to *System Parameter*.

**Other Knowledge Entities** Although the entities that are whitelisted in figure 3.2 are not unimportant and do interact with the entities just discussed, their description is left out here because they are not relevant to the quantitative analysis. For full description of the domain model, see [2].

The domain model described in this section is that used in the development of the Excel Client. In order for it to be usable for the Python Client (pyQAM), it needed some adjustments. The differences between the Excel domain model and the Python domain model are described in section 5.5 on page 50.

3.3.2 Knowledge to compare two analysis models

*What knowledge is needed to compare two analysis models that one understands?*

Having the knowledge needed to understand two analysis models, what else is needed to be able to compare these two analysis models? Somehow, the analyst has to map the parts of one analysis model to the other analysis model. In terms of the domain model, she needs to map *System Parameters* from one analysis model to *System Parameters* of the other analysis model; and *Analysis Functions* from one model need to be mapped to *Analysis Functions* of the other.
This mapping is exactly the knowledge needed to compare two analysis models. A part of one analysis model that doesn’t have a mapping to the other analysis model, is a difference; a part that does have such a mapping, is a part that both analysis models have in common.

### 3.3.3 Aspects of comparing analysis models

*What are the aspects of acquiring knowledge of an individual analysis model and of comparing two analysis models?*

There are many ways in which an analyst can analyze and compare analysis models, and many skills that can be used in the process. No two analysts have exactly the same skill set, and no two analysts approach a comparison problem in the same way. *Aspects* of comparing analysis models are the approaches and techniques that can be taken to compare analysis models.

In some cases, the analyst has prior knowledge that comes in handy when comparing two analysis models. For example, when she has worked on similar analysis models before, she might know where to look or what to compare to quickly find results. If the analyst is an expert on a knowledge domain that is particular to the system design and the analysis model, she might have another advantage. However, in this section, we are interested in the aspects of acquiring knowledge about the analysis models at hand without any prior knowledge.

This section lines out two key aspects of comparing and understanding analysis models that can be employed by any analyst, and which are used as an example for the solution presented in section 3.4: dependency analysis and mapping of semantics. With these, the knowledge described in sections 3.3.1 and 3.3.2 can be acquired from analysis models.

### 3.3.3.1 Dependency Analysis

![Dependency graph of a calculation](image)

One of the first things an analyst does, especially if she lacks domain knowledge (see 3.3.5), is to find out the dependencies in an analysis model. A *dependency* occurs when one value is the result of a calculation that takes another value as input; the former is a dependency of the latter. The relation they have is a *dependence*. This is illustrated in figure 3.3: the value $a$, which is the result of the formula $b \cdot c$, depends on values $b$ and $c$. When a value changes, most or all of its dependencies also change.

Taking into account the kind of calculation, the name a calculation may carry, the names of variables involved or annotations that are present, the analyst can get a good understanding of the semantics of the found values and dependences. Each value corresponds to what is called a *Number* in the domain model of section 3.3.1.1; each dependence runs through an *Analysis Function*. Intuitively, the analyst assigns *Numbers* to *System Parameters*. Thus, the knowledge described in section 3.3.1 can be acquired through dependency analysis.

Dependencies are transitive: if a first value depends on a second value, and the second value depends on a third, the first value also depends on the third. In figure 3.3, $a$ depends on $f$, $g$ and $d$ because $b$ does; because $d$ depends on $h$ and $i$, $a$ also depends on $h$ and $i$. In fact, $a$ depends on all the leaf nodes depicted in the figure; these are the *assumptions* that the end result $a$ depends
on. Dependency analysis involves tracing these chains of direct dependences to determine indirect dependencies.

Through dependency analysis, an analyst can find the analysis results that are influenced by an assumption of the analysis model; this is an analysis result in itself. Dependency analysis can also be part of a more complex routine to gain more knowledge about the internal structure of an analysis model. Typically, the latter case involves more localized dependency analysis: not an entire dependency chain is analyzed, but only the portion corresponding to the part the analyst tries to understand.

Dependency analysis can be done in two directions: starting with an assumption to follow the dependences to the results it influences, or starting at a result to determine the assumptions it depends on. The former is called forward dependency analysis, the latter reverse dependency analysis. Note that this convention is a bit counter-intuitive when it comes to figure 3.3: forward dependency analysis goes in the direction opposite to the direction of the edges in that figure (and in other figures). This is because the edges have the semantic meaning depends on, and not has dependency.

Forward dependency analysis is employed when the analyst tries to gain more knowledge about the structure of the analysis model: with it, she can find out how an assumption propagates through the analysis model. Reverse dependency analysis is more suited for quick answers: typically, it is used when one result of the analysis model is subject of discussion and the analyst wants to find out how that result is calculated, ignoring parts of the model that are irrelevant to that particular result.

3.3.3.2 Mapping semantics

Using information like annotations, names of variables and names of calculations, an analyst can match System Parameters from one model to System Parameters in the other model. In most cases this requires that the analyst has knowledge that cannot be derived from the analysis model itself: for example, she must be familiar with what a calculation is for, instead of only what it looks like.

When some of the System Parameters and Analysis Functions have been mapped, other mappings may be derivable from these mappings. For example, when an Analysis Function’s input System Parameters in one analysis model have been mapped to their counterparts in another analysis model, and its output System Parameters have also been mapped, it is possible to map the Analysis Function to its counterpart(s) in the other analysis model. Thus, only a subset of the semantics needs to be mapped by an expert analyst; if enough mappings have been made, other mappings can be derived. See sections 7.2 and 7.3 for further elaboration on this topic.

3.3.4 Significance of values

What are significant values in an analysis model?

The significance of a value is determined by how crucial a role it will play in discussions about the system design, and in the rationale behind system design decisions. Generally, an analyst is an expert in the knowledge domain on which the analysis model operates; she knows certain rules of thumb to guess which values are important and which are not. Experience with previous projects or previous versions of the analysis model benefit the analyst’s ability to determine the significance of a value. When in doubt, a black-box approach can be used: the analyst can change one of the input values (assumptions) to determine which results change, and how much they change. This gives the analyst a feeling of how significant an assumption is to an end result. However, even though these methods may work, they are hardly a systematic way of determining how significant values in an analysis model are.

Values in an analysis model can be divided into three categories:

1. Assumptions
2. Intermediate results
3. End results
An assumption is a value that is input to the analysis model. Results are values that are produced by calculations in the analysis model. Some, if not most, of the results are only in the analysis model to be fed into other calculations: these are the intermediate results. Other results are end results. Note, however, that a result that is fed into another calculation can still qualify as an end result: being an end result is solely dependent on how crucial it is to design decisions.

Assumptions and end results are the only ones that really matter; the intermediate results are insignificant. End results and assumptions are not all equally significant: one end result may play a more crucial role when a design decision is to be made than another, one assumption may have greater influence than another.

The significance attached to an end result or assumption, or a classification of results into intermediate and end results, is not universal. Each person on a design team may have different opinions on these matters, depending on their domain of expertise, the stage in which the project is, personal taste, and so on. One reason for having design discussions is precisely because a team needs to reach consensus on which attributes of the system design are more important than others.

Returning to the analyst's task, she has to determine from the analysis model:

- Which results are intermediate results and which are end results, and
- Which end results are more significant than others.

Of course, the results of this analysis are from the point of view of the analyst; others might produce a different analysis, as explained in the last paragraph. However, the goal of this analysis is not to give a definitive answer to how significant each value in the analysis model is; it is to determine which parts of the analysis model are most likely to yield interesting results in further analysis.

3.3.4.1 Determining a significance "rating"

Although significance of values in an analysis model is a subjective matter, it is still possible and desirable to have a method of finding a likely ordering on significance of these values. This method should be generic, so it is applicable to analysis models in general. This means that it must consider only properties that can be derived from the structure of an analysis model; it should not consider properties that come from the knowledge domain that is the context of the analysis model's calculations.

A property of a value that is derivable from the analysis model, is its place in the dependency graph (discussed in section 3.3.3.1). An important thing to realize is that a value has two roles in the dependency graph: it influences its dependencies, and it is influenced by its reverse dependencies. For both of these roles, a different approach is needed to determine the value's significance.

This poses a problem: which one of the two roles yields the significance "rating" we are looking for? Recalling the classification of values into assumptions, intermediate results and end results, an answer presents itself. Assumptions only influence other values, as they are not calculated; therefore, their significance is determined by their influence on other values. If an end result is not used in any calculation, it has only the role of being influenced by other values. Intermediate results are by definition not significant.

The problem persists for end results that still influence other values in the model: is such a result significant because it has a very deep reverse dependency tree, or is it significant because it influences a lot of other values? And, unfortunately, classification of results into intermediate and end results is subjective; therefore, we have to treat every result as an end result, assigning a low significance to results we think are intermediate results. An analyst has to consciously select the role on which to determine significance when using this method, and be aware of the fact that a significance determined by the other role might be completely different.

Below are four approaches that were determined to work well when determining significance of values in an analysis model. The first takes a shot at a classification of results into intermediate and end results; the second and third summarize how significance is determined for each of the two roles, and the fourth gives a way of determining the significance of a value based on the significance of its (reverse) dependencies.
CHAPTER 3. ANALYSIS

3.4 EASIER, LESS ERROR-PRONE AND QUICKER

Analysis model design  Probable end results may be derived from the analysis model design, for example, if analysis model's design is divided into components. When a result of a component is not re-used in the component itself (while it may be used in other components), it is likely to be an end result.

Influence on other values  When the role of influencing other values is important, significance of a value can be measured by the number of other values it influences, directly and indirectly. For assumptions, this is the only significance rating available; significance of other values can also be rated by the depth of the reverse dependency tree.

Depth of the reverse dependency tree  When influence of other values on a value is important, significance of (end) results can be measured by longest dependency chain (i.e., the depth of the reverse-dependency tree) it is influenced by.

Significance of (reverse) dependencies  When significance of one value has been determined, this significance can be used to determine significance of values it influences or is influenced by.

3.3.5 Prerequisites for comparison

What are the prerequisites for successfully comparing analysis models?

Following from the previous sections, the prerequisites for successfully comparing analysis models are, that the analyst:

1. Needs to have enough knowledge particular to the knowledge domain of the system design in order to be able to interpret the semantics of values; in other words, she needs to have enough knowledge to identify and map System Parameters,

2. Needs to have enough knowledge particular to the knowledge domain of the system design in order to be able to interpret what calculations do; i.e., she needs to have enough knowledge to identify and map Analysis Functions,

3. Needs to be able to perform dependency analysis on the format of any of the analysis models to be compared. For example, one analysis model might be created in Excel while the other is written in Python; to compare them, the analyst needs to be able to perform dependency analysis on both.

4. Needs to have enough knowledge particular to the knowledge domain of the system design in order to be able to determine which values are significant in the analysis model and which aren’t.

3.4 Easier, less error-prone and quicker

How can we make finding differences between different analyses of the same system easier, less error-prone and quicker?

Interpreting easier as requiring less prerequisites, the first part of this research question can be re-formulated as:

1. How can we minimize the number of prerequisites for finding these differences?

Section 3.3.3 mentions two activities in acquiring knowledge about one analysis model and comparing two analysis models to each other, those two being dependency analysis and mapping semantics. Therefore:

2. How can we make dependency analysis less error-prone and quicker?

3. How can we make mapping semantics less error-prone and quicker?
3.4. EASIER, LESS ERROR-PRONE AND QUICKER

CHAPTER 3. ANALYSIS

Minimizing the number of prerequisites  The prerequisites for an analyst to perform analysis and comparison on analysis models that need to be minimized, are (sec. 3.3.5):

1. Enough knowledge to identify and map System Parameters,
2. Enough knowledge to identify and map Analysis Functions,
3. Able to perform dependency analysis on the format of any of the analysis models
4. Able to determine which values are significant in the analysis model

Unfortunately, the first two prerequisites can’t be eliminated entirely; there simply is no substitute for domain knowledge. Dependency analysis can compensate for the lack of knowledge to identify and map System Parameters and Analysis Functions.

Which brings us to the dependency analysis itself: the analyst has to be able to do dependency analysis on the formats of any of the analysis models that are to be compared. This prerequisite can be eliminated by converting each of the analysis models to the same format or visualization; preferably, this format or visualization makes dependency analysis on the analysis model as easy as possible.

The visualization can also give suggestions on which values are significant and which are not, using approaches from section 3.3.4.1. This alleviates the need for the last prerequisite.

Making dependency analysis less error-prone and quicker  Dependency analysis, as described in section 3.3.3.1 is an activity that doesn’t require knowledge other than is embedded in the analysis model itself; apart from the knowledge about the format in which the analysis model is implemented. This is a task that can be carried out by a software tool. Results of the software analysis can be visualized in different ways, depending on the preference of the analyst.

Making mapping semantics less error-prone and quicker  Section 3.3.3.2 stated that once sufficiently many Analysis Functions and System Parameters are mapped between analysis models, other Analysis Functions and System Parameters can be derived from these mappings. The first task of defining the first mappings between analysis models remains an expert human’s task; but deriving other mappings from these can be performed by a software tool.

3.4.1 Tool: Knowledge Architect and pyQAM

The results of the analysis presented in this chapter have been used to create a software tool to facilitate analysts in analyzing analysis models. This tool focuses on analysis models in Python, and is called pyQAM (short for Python Quantitative Analysis Model). The pyQAM tool is part of the Knowledge Architect suite [2], and is to be used used in combination with a similar tool for Excel analysis models to compare an Excel analysis model to it’s Python counterpart. The next chapter elaborates on pyQAM’s requirements and design.

The following chapters describe problems and solutions found in implementing the results from this analysis:

- Chapter 5, Capturing Analysis Model Data, describes how dependency analysis can be performed in a software tool on a Python analysis model in order to extract the knowledge described in section 3.3.1. The Excel Knowledge Extractor does the same job for analysis models created in Excel (see [2]). Output of both the Excel tool and the Python tool are in the same format.

- Chapter 6, Visualizing Analysis Models, describes a number of visualizations that can be created with the output of either the Excel or Python tool. With these visualizations, an analyst can perform dependency analysis with less errors and with more speed.

- Chapter 7, Comparing Analysis Models, elaborates on the mapping of System Parameters and Analysis Functions, described in section 3.3.3.2, as a basis for a number of comparative visualizations that enable analyst to compare analysis models with less errors and with more speed.
Chapter 8, *Validation*, presents the results of an experiment to validate the implementation of the visualization of individual analysis models.
3.4. EASIER, LESS ERROR-PRONE AND QUICKER
Chapter 4

pyQAM

4.1 Introduction

ASTRON’s main interest lies in an answer to the last research question: How can we make finding differences between different analyses of the same system easier, less error-prone and quicker? The answers to this question have been put to work in a software solution called pyQAM, of which the use cases, requirements and design are described in this chapter.

Firstly, the perspective of each of the stakeholders of pyQAM is described. A number of use cases is formulated; from perspectives and use cases, functional and non-functional requirements follow. Finally, the chosen solution is presented in the form of a high level design. Chapters 5, 6 and 7 give detailed views on some of its aspects.

4.2 Stakeholders

Some of the stakeholders to the tool to be developed have already been briefly described in the introduction chapter; the obvious remaining stakeholder is yours truly. Now follows a description of the problem at hand from the perspective of the GRIFFIN and ASTRON stakeholders.

4.2.1 GRIFFIN

![Architecture of GRIFFIN’s Knowledge Architect tool suite](image)

GRIFFIN’s Knowledge Architect suite revolves around a Sesame repository. This repository is a so-called semantic repository which can store data expressed in a domain model (see section 3.3.1.1).
4.2. STAKEHOLDERS

In figure 4.1, a high-level architecture is depicted which shows different components of the tool suite. The Sesame repository is part of the Knowledge Architect Server; this component also includes a web service through which the Sesame repository can be accessed. The Knowledge Architect Explorer can access the data stored in the repository and visualize it interactively.

The remaining components are parts of tools that are generally referred to as clients. These clients are responsible for capturing data from artifacts (design documents, code files, ...) that contain knowledge to store in the repository. Capturing this data is often not possible without a little augmentation: the artifacts need to be annotated. This is what the client component indicated with Knowledge Architect Annotator in the figure is meant to do. The Knowledge Architect Extractor is the part of a client tool that captures data conforming to the domain model from an artifact.

The Annotator component is depicted inside a File Authoring Program. An artifact is generally created and edited with software specific to that artifact (examples are Excel and Word files, which are generally edited by programs from the Microsoft Office suite). In those cases, the most convenient and easy to realize solution is to build a plug-in extension to the authoring program that enables the user to annotate the artifact.

The Extractor is inside a box named Version Management System (SVN). The Knowledge Architect is designed to be used in existing processes of system and software development; a version management system such as Subversion (SVN) typically plays a role in such a process. Subversion can be configured to invoke extensions as part of its commit transaction; that is how the Extractor components are activated. This way, the Extractors don’t have to be invoked explicitly on files managed by a version management system; they are automatically invoked whenever anyone commits them.

As mentioned in the introduction chapter, before the Python Client was developed, there were two client tools provided by the Knowledge Architect:

- The Word Client that can capture knowledge from system documentation written in Word format
- The Excel Client that can capture knowledge from analysis models created with Excel

The Word Client does not adhere entirely to the general architecture in figure 4.1; essentially, it does not contain an Extractor component. The Excel Client, however, does.

The Python analysis tool required by ASTRON was to be part of the Knowledge Architect tool suite; therefore, an Extractor and an Annotator component were needed. Furthermore, the Knowledge Architect Explorer did not (yet) feature required visualizations fit for quantitative knowledge (the kind contained within analysis models).

4.2.2 ASTRON

The timeline in figure 4.3 shows a rough schedule for the development of the SKA. At the time of writing, ASTRON is participating in PrepSKA depicted between ’08 and ’12; at the time of the...
development of the Python Client, the predecessor of PrepSKA, called SKADS, was in its final stages.

PrepSKA starts at the end of the Concept Design phase of the project. These are phases in which important design decisions are made. Especially on the Aperture Array, a new concept that is to fulfill a crucial role in the SKA and on which ASTRON is one of the leading institutes, decisions are to be made that will influence the quality of this billion-euro project. Needless to say that these decisions need to be well-founded, not in the least to get the funding for it to be developed and built.

As explained in the introduction, the outcome of a commonly accepted analysis model will be used to decide on design issues in the Aperture Array. Three analysis models have seen the light, the Excel model, SKAcost and SKAsim, the last of which will probably be the authoritative one when the final decisions are made. The Excel model is well-understood within ASTRON; the latter two less so.

For these reasons, ASTRON needs a tool that:

- Is easily transferable, deployable and maintainable
- Is lightweight, i.e., the tool should work on an ordinary workstation
- Is easy to use and understand;
- Allows as many functions as possible to be performed offline, i.e., on a stand-alone workstation without the requirement of a running server

### 4.3 Requirements

From the informally discussed perspectives of stakeholders in the last two sections, a formal set of use cases and requirements that have been used in the development of the Python Client are stated below.

#### 4.3.1 Use cases

The primary two use cases identified for the Python Client are "Respond to comments" (UC1) and "Compare concepts" (UC2). The former refers to a situation where a member of an institution other than ASTRON sends comments (criticism, remarks, statements) about a concept or design to an ASTRON member. The latter is more or less the reverse case: here, an ASTRON member compares two design concepts; say, one from ASTRON and one from another institute; and may end up with comments to send to the other institute. These two use cases were designated as the most important two to be supported by the Python Client.
4.3. REQUIREMENTS

The numbering of the use cases doesn’t reflect their importance, rather the order in which they have been conceived. The two use cases that were named are split up into use cases that support them:

- **UC1: Respond to comments**
  - UC1.1: Create an analysis model
  - UC1.2: Annotate an analysis model
  - UC1.3: Analyze an analysis model

- **UC2: Compare concepts**
  - UC2.1: Compare analysis models

These use cases, together with the requirements established for the Excel Client, lead to the requirements for the Python Client (see section 4.3.2).

### 4.3.1.1 Respond to comments (UC1)

Two scenarios are possible in this use case: one where there is not yet an analysis model for the criticized concept or component, and one where there is.

The steps in the first case (no analysis model available):

1. Create / modify analysis model (UC1.1)
2. Annotate the analysis model just created (UC1.2)
3. Perform analysis (UC1.3)

In case there already is an analysis model, the first step(s) may be skipped.

**Create / modify an analysis model (UC1.1)** This corresponds to, in the Python case, writing or modifying a Python program that calculates attributes for the system design. What this means in terms of requirements for the Python analysis tool, is that each time a model has been modified, the user must be able to re-capture it with the tool.

**Annotate an analysis model (UC1.2)** For best results with the tool, the analysis model needs to be annotated. The tool must provide a way to do this, preferably an easy one.

**Analyze an analysis model (UC1.3)** The tool must provide the user with the ability to create one or more visualizations of one model, facilitating the analysis task.

### 4.3.1.2 Compare concepts (UC2)

Again there are two scenarios, one where there is already an analysis model for both concepts, and a second where there isn’t. The full set of steps is:

1. Create / modify analysis model(s) (UC1.1)
2. Annotate analysis model(s) (UC1.2)
3. Perform comparison (UC2.1)

The first step(s) may be skipped, as in UC1.

**Compare analysis models (UC2.1)** Actually an extension of UC1.3, the user is provided with an visualization of the differences and similarities of two analysis models.
4.3.2 Requirements

Functional requirements were gathered from two sources: the use cases lined out above, and the requirements already stated for the Excel client. Table 4.1 gives an overview of the requirements for the Python Client. In the last column, the corresponding use cases and Excel requirements are stated. See figure 4.4 for how the requirements relate to use cases.

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Dep.</th>
<th>Origin / Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>Represent the analysis model in a visually attractive manner</td>
<td>R11</td>
<td>Excel-R03, UC1.3</td>
</tr>
<tr>
<td>R02</td>
<td>Enable the user to fold/unfold abstractions in the visualization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R03</td>
<td>Clearly indicate descendants and precedents of a selected system parameter</td>
<td>R01</td>
<td>Excel-R07, UC1.3</td>
</tr>
<tr>
<td>R04</td>
<td>Provide a way to perform parameter analysis by hand</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R05</td>
<td>Deduce semantically enhanced formulas from the model’s formulas</td>
<td>R08, R09</td>
<td>Excel-R19, UC1.3</td>
</tr>
<tr>
<td>R06</td>
<td>Provide a way to group formulas in abstractions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R07</td>
<td>Provide a way to store the point of origin of an input: who defined it and why?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R08</td>
<td>Provide a way to annotate formulas with unit information</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R09</td>
<td>Provide a way to annotate formulas with semantic information and mathematical symbols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>Provide a way to annotate abstractions with rationale</td>
<td>R06</td>
<td>Excel-R05, UC1.2</td>
</tr>
<tr>
<td>R11</td>
<td>Extract formulas from a model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td>Build a dependency graph from formulas</td>
<td>R10</td>
<td>Excel-R02, UC1.1</td>
</tr>
<tr>
<td>R13</td>
<td>Visualize the differences between two analysis models</td>
<td>R11</td>
<td>Excel-R23, UC2.1</td>
</tr>
</tbody>
</table>

Table 4.1: Functional requirements

In addition, the Python Client had to satisfy some non-functional requirements. These are summarized in table 4.2.

4.3.2.1 Requirements following from UC1.1

Requirements R06, R11 and R12 fall in the category of UC1.1. R6 states that the user should be able to split the analysis model into "abstractions" in order to hide complexity. R11 and R12 require from the Python Client to provide a way to capture information from an analysis model.

4.3.2.2 Requirements following from UC1.2

Requirements R07, R08, R09 and R10 state the additional information that the Python Client should be able to process. This additional information falls into two categories:

1. Meta-information: R07 (who defined something and why) and R10 (rationale for abstractions)
2. Semantic enhancements: R08 (unit information) and R09 (semantics, mathematical symbols)
<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Origin / Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF01</td>
<td>Lightweight</td>
<td>Excel-NF01, ASTRON</td>
</tr>
<tr>
<td>NF02</td>
<td>Easily deployable and maintainable</td>
<td>Excel-NF02, ASTRON</td>
</tr>
<tr>
<td>NF03</td>
<td>Easy to use and understand</td>
<td>Excel-NF03, ASTRON, GRIFFIN</td>
</tr>
<tr>
<td>NF05</td>
<td>Robust (not crash)</td>
<td>Excel-NF05, ASTRON</td>
</tr>
<tr>
<td>NF07</td>
<td>Extensible</td>
<td>Excel-R18, ASTRON</td>
</tr>
<tr>
<td>NF08</td>
<td>Stand-alone</td>
<td>Excel-R31, ASTRON</td>
</tr>
<tr>
<td>NF09</td>
<td>Part of Knowledge Architect</td>
<td>GRIFFIN</td>
</tr>
</tbody>
</table>

Table 4.2: Non-functional requirements

4.3.2.3 Requirements following from UC1.3

Requirements R01, R02, R03, R04 and R05 all have to do with the analysis of an analysis model. From the point of view of the Python Client, they are mostly the visualization requirements, except for R04 and R05. R04 states that the tool must leave enough freedom for a user to do analysis by hand. R05, about semantically enhanced formulas, enables the user to understand formulas in the analysis model more easily and quickly.

4.3.2.4 Requirements following from UC2.1

The remaining requirement, R13, states that two analysis models must be visualizable in one visualization.

4.3.2.5 Non-functional requirements

The non-functional requirements are desired quality attributes of the Python Client. Four of these requirements had priority (key drivers):

1. **NF03: Easy to use and understand**
   In order for the Python Client to be accepted and used by the ASTRON engineers, a lot of effort has been put into making it usable and user-friendly.

2. **NF08: Stand-alone**
   Actually, NF08 is implied by both NF02 (easily deployable and maintainable) and NF03 (easy to use and understand). It is separately mentioned, however, because it poses an important problem in combination with the next non-functional requirements.

3. **NF09: Part of Knowledge Architect**
   The Python Client needs to be part of the Knowledge Architect tool suite. Recalling the description of the overall architecture of the Knowledge Architect, a problem arises with respect to NF08: ASTRON wants the Python Client to be stand-alone, while GRIFFIN wants the Python Client to fit into a client-server architecture through NF09.

Halfway the development of the Python Client, SKAsim emerged as a successor to SKAcost, the original Python analysis model that was to be analyzed. Because SKAsim was intended to be the authoritative analysis model, considerable effort has been put in a fourth non-functional requirement:

1. **NF07: Extensible**
   That is, extensible in the sense of enhancing the Python Client to be able to analyze analysis models with a different architecture than the ones that are already understood by the tool.

The remaining non-functional requirements are either generalizations of these four, or had less priority.
CHAPTER 4. PYQAM

4.3. REQUIREMENTS

Figure 4.4: Use cases vs. requirements
4.4 Design

Requirements and use cases have ultimately led to the design lined out in this section. Ultimately, because development of the Python Client was an iterative process; experiences gained in the implementation influenced the design. For details and reasons why certain choices were made, see chapters 5, 6 and 7.

4.4.1 Components

(a) Python Client Annotator component

(b) Python Client Extractor component

Figure 4.5: Components of the Python client

The overall design of the Knowledge Architect (see figure 4.1) dictates that the Python Client is composed of two components: an Annotator component and an Extractor component. The Annotator component aids the user in annotating an analysis model, enhancing it with meta-information and preparing it for the capturing process performed by the Extractor component. The Extractor component extracts data and stores it into a repository during a commit action to a version management system.

For the Python Client, the situation is slightly different than for the Excel Client: for one, there is not a designated file authoring program for Python. Therefore, the Annotator is not a plug-in as it is in the Excel case; it is a stand-alone program called pyQAMui (python Quantitative A nalysis M odel user interface), which is a front-end to a library called pyQAM. The pyQAM library provides functions to capture Python analysis model data, callable from any Python program. The pyQAMui program enables the user to produce so-called configurations for pyQAM, which are in turn Python programs that invoke an analysis model with pyQAM in a certain configuration. A simplified depiction of the Annotator and the Extractor is shown in figure 4.5.
In fact, a *configuration* produced by pyQAMui is part of the Extractor component for the Python Client, tailored to one analysis model with one set of input parameters (which constitutes a *scenario*). When this program is invoked, it runs the analysis model and outputs captured information about the analysis model. The output of the script can be fed into a component that can visualize it, such as the `pyqam2dot` program in figure 4.6, that generates GraphViz dot files. The output can also be fed to a component that can insert the data into a Sesame repository (`PyQAM2Sesame` in figure 4.6). The data can then be visualized with components that read their data from the Sesame repository (*SysparamListHtml, AssumptionListHtml, NumberTableHtml*).

PyQAMui can invoke the configuration script it generates, and feed the output into another program (such as `PyQAM2Sesame`). Thus, PyQAMui can not only act as the Annotator, but it is also capable of calling the Extractor, removing the need for the user to manually invoke the script and feeding its output to `PyQAM2Sesame`.

### 4.4.2 Allocation

![Diagram of allocation](image)

Figure 4.7: Full deployment of the Knowledge Architect tools. On a workstation, analysis models are annotated and prepared for analysis; a versioning source repository system (in this case Subversion) captures Excel and Python data from submitted artifacts and sends it to the Sesame Repository. The Word Client communicates directly with the server via its web service (SOAP) interface.

There are two seemingly conflicting (non-functional) requirements that have to do with the allocation of the Python Client’s components to different hardware components (in practice, workstation and server). They are:

- **NF08:** Stand-alone
- **NF09:** Part of Knowledge Architect

A closer look at figure 4.7 shows why these requirements are conflicting. This is a more detailed view of the allocation of Knowledge Architect components on workstations and server that was described in section 4.2.1. Here, all clients are mentioned explicitly and the Knowledge Architect Server is depicted separately from the Sesame Repository, to show that the Excel Client and Python Client interact in a different manner with the repository than the Word Client does. As is obvious from the figure, this set-up requires a server to be configured and maintained, which conflicts with NF08.
4.4. DESIGN

Figure 4.8: Deployment of the Knowledge Architect tools with no source repository. Annotation of analysis model and capturing of data happens on a workstation, the captured data is sent to the central knowledge repository.

Figure 4.9: Workstation-only deployment of Knowledge Architect tools. In this case, the Sesame repository is hosted on-demand (i.e., the repository logic is included in the client software and executed when needed) on the workstation itself. Note that the Word Client cannot communicate with a Sesame repository directly; it needs the web service exposed by the Knowledge Architect Server, and therefore is of little use in a workstation-only set-up.
One option is to remove the Subversion component; the Extractor components of the Python Client and the Excel Client need to be invoked manually. We then get the allocation scheme of figure 4.8. This allocation is more acceptable to ASTRON, as they are not (yet) using Subversion to store versions of the analysis model; manually invoking the Extractor component is not a problem. Now, the only remaining obstacle is to eliminate the requirement of a separate server to store the knowledge captured by the clients. However, even if this server component is not eliminated, it does not have to conflict with the stand-alone requirement: server and client programs can run on one workstation, giving the user the impression that the Python Client is a stand-alone tool.

The solution chosen by ASTRON is to have no server process running at all. The Sesame repository logic can be compiled into a client (this repository logic is referred to as ElmoConnector in figure 4.6 on page 28), allowing an Extractor program to access a repository on-disk, i.e., the operations otherwise done by a server are now done by the client programs. The Python Extractor component is still able to access a Sesame repository via a server process, so the previous two allocations are still possible. Because the Python Extractor shares the code to access a Sesame repository with the Excel Extractor, the latter benefits from this modification, too. The Word Client, however, can only communicate with a SOAP web service; because a web service still requires a separate server process, the Word Client cannot be deployed "fully stand-alone".

4.4.3 Logical design of pyQAM

![Module lay-out of pyQAM (simplified). A configuration script contains annotations and specifications (defined through pyqam.spec) of how to capture data from the analysis model. An analysis model itself can also specify annotations; it has to import the pyqam.annot module to do so. The configuration script uses the pyqam.alyze module to generate output defined by pyqam.model. This data can be read for further analysis, for example by making use of the PyQAMReader module (which can be used in a Java environment).](image)

The heart of the Python Client is the pyQAM library that consists of a number of modules that each carry out a specific task. A schematic overview of the modules is in figure 4.10; this is a somewhat simplified view, the details are covered in section 5.3.

The module where all functions come together, is the pyqam.alyze module. This module reads configuration information by using the pyqam.spec module, sets up the analysis engine and uses the domain model defined in pyqam.model to construct a model.

In the figure are two entry points, the configuration script (probably generated by pyQAMui) and PyQAMReader. The configuration script makes use of pyqam.spec to specify a configuration,
and then calls `pyqam.alyze` to do the analysis; it then obtains the result from `pyqam.alyze` and outputs it. `PyQAMReader` uses `pyqam.model` to decipher the output generated by the configuration script.

**4.4.3.1 Configuration subsystem: `pyqam.annot` and `pyqam.spec`**

The configuration subsystem allows for two types of configuration: one is by annotations in the analysis model, the other is by configuration directives in a configuration script. The former is used primarily to alleviate the task of the model analyst; if an analysis model is often captured with pyQAM, it might be a good idea to specify some configuration within the model so that it does not have to be entered every time. The configuration specified through `pyqam.spec`, however, always overrides annotations in the analysis model.

**4.4.3.2 Model: `pyqam.model`**

The `pyqam.model` module is a Python-native implementation of the domain model. It contains a collection of classes that each represent one of the classes in the domain model. It is possible to store data expressed through `pyqam.model` in a file, or to serialize it into a stream of data, so that other programs can consume it. Every program that needs to read or write this data, must use `pyqam.model` to do so.

**4.4.3.3 Analysis engine**

The analysis engine is a collection of modules that attach hooks to the Python analysis model, and provide the `pyqam.alyze` module with trace information. The specifics of these modules are the subject of section 5.3 and further.
Chapter 5

Capturing Analysis Model Data

5.1 Introduction

The analysis models in Excel, SKAcost, and SKAsim roughly analyze the same system design; each of them does it in a different way. This chapter is about the strategies followed to extract structural information from each of these analysis models. The structural information should be represented in such a way that it is easier to see what calculations are in the analysis model, and that it becomes possible to compare them to each other. There is one section for each of the analysis models; each of these explains in more detail:

- how the analysis model works,
- what kind of structural information it contains,
- which steps are taken to extract it.

*Structural information* is information in terms of the domain model of section 3.3.1.1 on page 11. The relevant parts have been depicted again in figure 5.1 on the following page. For a discussion on the domain model differences between the Excel and Python cases, see the last section of this chapter.

5.2 Excel workbook

One of the reasons for choosing Excel as the format for an analysis model, is that the Excel program is available everywhere; everyone knows how to work with it; and Excel offers an intuitive interface to write down calculations, check them and modify them. Excel is also very flexible: it doesn't enforce structural rules, making it easy, for example, to create small calculations on the side to illustrate the analysis model or to prove a point. This flexibility is also a downside: the analysis model can become a big mess with chunks of seemingly unrelated calculations scattered all over the workbook. Furthermore, it is difficult to enforce structural rules to the analysis model, making Excel limited when it comes to re-use and error checking.

An Excel *workbook* is a collection of Excel *sheets*. A sheet contains a *grid*, and in each *cell* of the grid the user can put data. This data can be static (e.g. a label indicating what is in another cell), but it can also be a value calculated from data in other cells: a formula with references to other cells can be entered, of which the calculated result is displayed. The formula is re-calculated automatically when referenced cells change.

The SKADS Excel analysis model uses multiple sheets in one workbook. Generally speaking, each sheet calculates values for a part of the system design. There is one “main” sheet which gives a summary of results; the main sheet also presents a table in which input values (“assumptions”) to the analysis model can be tweaked. Each sheet presents a view on the analysis model.

The cells in an Excel workbook and the references between them can be represented in a graph. Each node of the graph represents a cell with a formula or value, and each directed edge (arrow)
Figure 5.1: Domain model for the Excel analysis model
between nodes represents a reference or dependency from one node to another. From the graph, a
data structure according to the domain model in section 3.3.1.1 can be constructed.

This is exactly what the Excel Client does. A plug-in for Microsoft Excel calculates the depen-
dency graph for all formulas in the workbook. The tool also allows for annotations to be added to
cells; with these, extra information can be added to cells to label them, group a number of cells or
to uniquely identify them. The Excel plug-in can display the dependency graph with a number of
enhancements, such as expanding and collapsing groups of cells ("abstractions") and highlighting
the dependency tree for one node. When the Excel workbook with annotations is saved to a file,
a separate Java utility, the Excel Knowledge Extractor, can extract data from it, and save it to a
repository for further analysis. For a more detailed discussion on the Excel Knowledge Extractor,
see [2].

5.3 SKAcost

```python
class PersonAndBike:
    "\n    __init__( self , personMass , bikeMass , velocity ):
        person = MassAndVelocity(personMass , velocity)
        bike = MassAndVelocity(bikeMass , velocity)
        self .KE = self .calcSystemKE(bike .KE, person .KE)
    
    def calcSystemKE( self , personKE , bikeKE):
        return bikeKE + personKE

class MassAndVelocity :
    "\n    __init__ ( self , mass , velocity ):
        self . velocity = float ( velocity )
        self .mass = float (mass)
    
    def calcPartKE( self ):
        self .KE = .5 * self .mass * \\
        ((self .velocity * 1000/3600) ** 2)
```

Figure 5.2: Example analysis model in Python using the SKAcost principle

SKAcost is a Python program in which, among others, the Aperture Array design is modeled.
The program can be invoked from the command-line or from a web interface. It is possible to pass
custom input parameters to SKAcost in both situations. SKAcost prints a report in plain text
stating the input values and the calculated values.

In SKAcost, the system design is modeled into an object oriented program. Components of
the system design are mapped to classes, each of which can be seen as an analysis model for the
component it represents. A component class can be instantiated multiple times with different parameters, providing a nice way of modeling variations of a component in the system. It is easy to aggregate components into one parent component: the parent component class simply needs to reference component class instances of the components it contains. Thus, the entire analysis model is aggregated into one super-component; this super-component is instantiated when the program is invoked.

A component class instance (object) calculates a number of values when it is instantiated. These values are stored in attributes (members) of the object. Another object, often a parent object (i.e., the former has a part-of relation with the latter), can then use these stored values in its own calculation.

An example of how this works is in figure 5.2. This example shows an analysis model in Python of a person and a bike traveling at a certain velocity. When `PersonAndBike` is instantiated (in Python, the constructor of a class is the `__init__` method), `PersonAndBike` instantiates the component `MassAndVelocity` twice to calculate the mass and velocity of person and bike. After that, the `calcKE` method of `PersonAndBike` is called to calculate the kinetic energy of the entire system, using the values calculated by `MassAndVelocity`. The value for the kinetic energy is then stored in the `KE` attribute of the `PersonAndBike` object.

### 5.3.1 What to capture

Ideally, the data captured from the SKAcost model should be represented using the same domain model as the Excel tool does. Taking that as a starting point, a first attempt was to identify each of the classes depicted in figure 5.1 on page 34 in the Python model.

In the following, the words `method` and `attribute` are used to respectively indicate a class method and an object attribute. An `attribute` (in other languages sometimes referred to as a `member` or `property`) holds a value for that object. In most programming languages, all objects of a class have the same set of attributes (declared in the class), but in Python this is not true in most cases: object attributes are dynamically added during the lifetime of the object. To be clear, the term `class attribute` refers to an attribute that all objects belonging to that class have.

It may be clear from the previous section that `Component` from the domain model maps to classes in SKAcost. `Analysis Function` maps to methods, `System Parameter` maps to class attributes, `Number` maps to values that are stored in attributes. However, things are not as straightforward as they may seem on first sight: not every class is a `Component`, not every method is an `Analysis Function`, not every class attribute is a `System Parameter`, not every value stored in an attribute is a `Number`. Take for example the `__init__` methods of both `PersonAndBike` and `MassAndVelocity` in the example in figure 5.2: although these are methods, they do not contain system parameter calculations (they only call methods that contain these calculations), and therefore they do not map to `Analysis Function`. Not only class constructors are exceptions: methods that construct string representations of their objects, or methods that perform error checking are some examples. Likewise, a class attribute may not be directly involved in calculations, which means it doesn’t qualify as a `System Parameter`; a value may not be stored in an attribute, disqualifying it as a `Number`.

What is needed, is a set of rules to use to determine whether classes, methods, attributes and values are `Components`, `Analysis Functions`, `System Parameters` and `Numbers`, respectively:

1. A class is a `Component` when it contains at least one `Analysis Function` method
2. A method is an `Analysis Function` if it calculates at least one `System Parameter`
3. A class attribute is a `System Parameter` if the values (`Numbers`) that are stored in the corresponding object attributes...
   (a) are calculated by an `Analysis Function`, or
   (b) are provided as input parameters
4. Any value is a `Number` if it is stored in an attribute that is a `System Parameter`
There are some problems with this set of rules. First of all, it contains a cycle: if we want to determine whether a method is an Analysis Function, we need to know whether its return value is stored in an attribute that is a System Parameter; but to determine whether that is a System Parameter, we need to know whether it is an Analysis Function that calculates its value(s) in the first place. Secondly, the domain model implies that every System Parameter is calculated by an Analysis Function; in practice, SKAcost contains a lot of small "on-the-fly" calculations that are not contained in a method. The improved rule set is as follows:

1. A class is a Component when it contains at least one Analysis Function method.
2. A method is an Analysis Function if it is specified as such.
3. A class attribute is a System Parameter if if the values (Numbers) that are stored in the corresponding object attributes...
   (a) are calculated by an Analysis Function, or
   (b) are provided as input parameters, or
   (c) are the result of a calculation outside any Analysis Function which involves other System Parameters.
   To compensate for the lack of Analysis Function in this case, the "anonymous" calculation is modeled in the domain model with Anonymous Analysis Function (see section 5.5 for an overview of the domain model adapted for SKAcost).
4. Any value is a Number if it is stored in an attribute that is a System Parameter.

The new rule set requires the user of the new Python tool to specify all Analysis Functions she is interested in; Numbers (and subclasses), System Parameters and Components can be derived with this specification and a running program. The remaining classes in the domain model are Scenario and Namespace, which entail meta-data that needs to be explicitly specified.

5.3.2 Strategy

5.3.2.1 Static or run-time analysis

There are two ways to analyze a piece of program code. One is by parsing it and building a structure from which information can be derived; this is referred to as static analysis. In this scenario, only "compile-time" or static information can be extracted from the code; run-time elements such as objects and values of variables need to be derived, which can be a daunting task. An example of static program analysis is program slicing (see section 9.4 on page 98).

The second way of analyzing a program (run-time analysis), is by simply running it while monitoring what it does; parsing and building structures is left to the interpreter or compiler for the particular programming language. The tricky part here is that it depends on the programming environment how easy it is to monitor what an executing piece of code exactly does.

Python, the language in which SKAcost is written, is an interpreted language. It has a highly dynamic nature, making it difficult to do static analysis on: for example, attributes are typically not declared in a class (like in Java or C++) but are attached during the lifetime of an object, effectively hiding attributes for static analysis. The dynamic nature of Python is also reflected in the fact that it provides great run-time introspection features, and that it allows to modify just about anything that is not a built-in Python construct in a running program: for example, it is possible to remove and attach methods of a class at runtime.

It may not come as a surprise that run-time analysis was the chosen strategy to capture analysis model data from SKAcost.

5.3.2.2 Pycallgraph

A first consideration was to build a call graph of all method calls in a run of SKAcost, and to derive analysis model data from that call graph. A call graph is a graph in which each node represents a
method, and the directed arcs between nodes represent calls from within the originating method to the destination method.

An example of a call graph generator for Python is *pycallgraph* [3]. It uses built-in Python tracing features to build a graph structure of function and method calls; this structure can then be visualized through GraphViz *dot*. Figure 5.3 shows the output of *pycallgraph* for the example analysis model *PersonAndBike* of figure 5.2 on page 35.

![Call Graph Example](http://pycallgraph.sourceforge.com)

**Figure 5.3:** Pycallgraph result for the calculation for PersonAndBike of figure Figure 5.2 on page 35. Parameters to PersonAndBike were personMass = 80, bikeMass = 20, velocity = 20.

From figure 5.3, the shortcomings of *pycallgraph* for our purpose become clear. The graph shows how often a method or function has been called and by which other functions / methods; but it doesn’t show in which order the calls were made, it doesn’t show which arguments were passed in each call, and it doesn’t show what was returned.

A difference between call graphs and a data structure in terms of our domain model is that the former shows dependencies between methods, but not dependencies between calculated values. Even if a call graph were to provide the lacking data mentioned earlier (order of calls, arguments, return values), it would not show these dependencies; and it would not provide us with an easy way of finding out in which attributes values are stored. This information might be derivable from a modified version of *pycallgraph* that shows more detail; but leaving the call graph out altogether might be faster and give better results.

### 5.3.2.3 Tracing calls

Even though *pycallgraph* doesn’t solve all of our problems, the technique it uses to build call graphs is still useful. *Pycallgraph* uses Python’s built-in function call tracing. The standard *sys.settrace* and *sys.setprofile* functions allow to specify a callback function that is called at each method call that occurs (and at some other events). At each call, the callback function is provided with many details, including arguments, return values, line numbers, etc. A simple trace of *PersonAndBike* that can be produced with the help of *sys.settrace* is in figure 5.4 on the facing page.

Using this call-back mechanism, it should certainly be possible to derive some of the analysis model data from a run of SKAcost. The following explains how this is done, referring to the rules stated in section 5.3.1.

The easiest job is to find *Component* classes: they are the classes that contain at least one *Analysis Function*. With all *Analysis Functions* specified, the *Components* are implicitly specified too.
**CHAPTER 5. CAPTURING ANALYSIS MODEL DATA**  

5.3. SKACOST

```python
call PersonAndBike.__init__ (velocity = 20, bikeMass = 20, personMass = 80)
execute line 9 (PersonAndBike.__init__)
call MassAndVelocity.__init__ (velocity = 20, mass = 80)
execute line 24 (MassAndVelocity.__init__)
execute line 25 (MassAndVelocity.__init__)
execute line 26 (MassAndVelocity.__init__)
call MassAndVelocity.calcPartKE ()
execute line 29 (MassAndVelocity.calcPartKE)
execute line 30 (MassAndVelocity.calcPartKE)
return from MassAndVelocity.calcPartKE: None
execute line 10 (PersonAndBike.__init__)
call MassAndVelocity.__init__ (velocity = 20, mass = 20)
execute line 24 (MassAndVelocity.__init__)
execute line 25 (MassAndVelocity.__init__)
execute line 26 (MassAndVelocity.__init__)
call MassAndVelocity.calcPartKE ()
execute line 29 (MassAndVelocity.calcPartKE)
execute line 30 (MassAndVelocity.calcPartKE)
return from MassAndVelocity.calcPartKE: None
execute line 11 (PersonAndBike.__init__)
call PersonAndBike.calcSystemKE (bikeKE = 1234.56790123, personKE = 308.641975309)
execute line 14 (PersonAndBike.calcSystemKE)
return from PersonAndBike.calcSystemKE: 1543.20987654
return from PersonAndBike.__init__: None
```

Figure 5.4: Example of a Python trace for the `PersonAndBike` example, with personMass = 80, bikeMass = 20, velocity = 20. Each line is produced by one call to the call-back function provided to `sys.settrace`.

<table>
<thead>
<tr>
<th>Python trace</th>
<th>Domain model</th>
</tr>
</thead>
<tbody>
<tr>
<td>call PersonAndBike.calcKE (bikeKE = 16000, personKE = 4000)</td>
<td>Analysis Function (PersonAndBike.calcKE) takes as input</td>
</tr>
<tr>
<td>execute line 13 (PersonAndBike.calcKE)</td>
<td>Number (#1) depends on</td>
</tr>
<tr>
<td>return from PersonAndBike.calcKE: 20000</td>
<td>Business Function (PersonAndBike.calcKE) calculated by</td>
</tr>
<tr>
<td>...and it returns a Number (#3) that has value 20000</td>
<td>Number (#3) depends on</td>
</tr>
<tr>
<td>...with as arguments one Number (#1) that has value 16000 and one Number (#2) that has value 4000...</td>
<td>Number (#2) takes as input</td>
</tr>
</tbody>
</table>

Figure 5.5: Example of how relations between `Number` and `Analysis Function` can be determined. On the left is a part of the Python trace in figure 5.4 (lines 21-23), on the right is the representation that can be derived from it.
5.3. SKACOST  

CHAPTER 5. CAPTURING ANALYSIS MODEL DATA

Figure 5.6: Example of how a System Parameter and its relations to Numbers and an Analysis Function are established and represented. The line of Python code comes from figure 5.3, line10.

We can detect Analysis Functions (which are specified by the user, rule 2) when they are called; i.e. we match specified Analysis Functions to names of methods that are called. Their arguments and return values are (potential) Numbers. An Analysis Function has takes as input relations to its argument Numbers, and a Number that is a return value has a calculated by relation to the Analysis Function. A Number that is a return value has depends on relations to all Numbers that are arguments. See figure 5.5 for an example.

Note that the relations between Numbers and Analysis Functions and between Numbers and Numbers do not exist in the domain model in figure 5.1 on page 34. However, they are needed to find the relations between Analysis Functions and System Parameters. Therefore, they do exist in the updated domain model in figure 5.13 on page 51 (see also section 5.5 on page 50).

To find System Parameters, we need to determine the is of type relations that are in the domain model between Number and System Parameter. System Parameter in the domain model maps to class attributes in the analysis model, so the attributes that Numbers are stored in must be detected to determine their "is of type" System Parameters.

Luckily, attribute assignment is implemented with method calls; which are calls that we can trace (even though they are hidden from the trace in figure 5.4 on the preceding page). An attribute that is assigned a Number is a System Parameter; the Number has an is of type relation with it. A System Parameter gets all the relations with Analysis Functions from the Numbers that have the is of type relation with it (rules 3(a) and 3(b)). A Number that is not stored in an attribute of an object is discarded: it is not typed by a System Parameter and therefore cannot be an actual Number (rule 4). See figure 5.6 for an example.

Lastly, if the attribute corresponding to a System Parameter is part of a class that is a Component, the System Parameter is part of that component. Thus, we can determine four key classes of the domain model in the analysis model, and the relations between them.

There is one downside to this approach using built-in Python tracing: with every function or method call, the call-back is also called at least twice (once when the function or method is called, and once when it returns). This means that the amount of function calls in the traced program more than triples compared to when it is run without tracing; execution speed deteriorates dramatically when tracing is enabled.
5.3.2.4 Wrapping methods

As mentioned earlier, Python allows methods to be replaced at run-time. This gives the opportunity to insert points where call-backs are called (let's call "calling a call-back" "firing events"): we simply detach a method we are interested in from an object or class, wrap it into a "wrapper method" that fires events, and reattach the wrapper method to the object or class it came from. Not all classes and objects allow this to be done: for example, built-in classes like `float` and `int` are not modifiable. However, `Components` defined in the analysis model are not Python built-in classes; therefore, we can do with those classes and their instances whatever we like.

This approach results in a much faster and cleaner tool than the one with built-in Python tracing. Instead of passively waiting for the Python tracer to fire an abundance of events of which only a tiny fraction is actually valuable, we can perform a little surgery on the Python analysis model at run-time so that it provides us with precisely those events that are needed. Furthermore, we have full control of the information passed with each event.

5.3.2.5 Wrapping objects

One downside to the whole approach of only tracing calls on methods of `Components` is that we can only capture relations if input values are passed as arguments to an `Analysis Function` and result values are returned. This is not always the case:

- Sometimes a calculation is entirely outside any `Analysis Function` (rule 3(c)). See figure 5.7 on the following page for an example.

- Often, an `Analysis Function` uses a value in its calculation that was not passed as an argument, but is referenced from within its body. This value is not detected as a `Number` to which the `Analysis Function` has a `takes as input` relation, while it really should be. An example of such a function is the `calcKE` method of `MassAndVelocity` in figure 5.2 on page 35: it requires no arguments, but it does reference the `self.mass` and `self.velocity` attributes from its body.

- Similarly, when an `Analysis Function` does not return a value, but instead directly assigns a result value to an attribute, this value is not detected as being a `Number` with a calculated by relation to the `Analysis Function`. Again, the `calcKE` method of `MassAndVelocity` serves as an example: it does not return the result of the calculation, but directly assigns it to `self.KE`.

In other words, tracing method calls is not fine-grained enough to capture all relations. To be able to also capture these cases, we need to trace (mathematical) operations on values (potential `Numbers`).

All operations (mathematical, logical, ...) on values are implemented with special methods in Python. For example, the addition operator (`+`) is implemented with the special `__add__` method; an expression like `a + b`, where `a` and `b` being integers, is implemented with the `int.__add__` method and is executed as `a.__add__(b)` (`a + b` and `a.__add__(b)` are equivalent statements in Python). Obviously, these methods can be traced similarly to how calls to methods of `Component` classes are traced in the previous sections; see figure 5.8 on the following page for how relations between `Numbers` and an `Anonymous Analysis Function` are established in the case of rule 3(c) from section 5.3.1.

There is one culprit: built-in types such as `int` and `float` cannot be modified the way user-defined classes (such as `Components`) can be modified. For example, an `int` object does not allow its methods to be replaced. This is a problem because unlike `Component` classes, a value object in a calculation is often of a built-in type.

The solution is to use inheritance: it is possible to define a subclass of a built-in type, and to override its methods. The overriding methods fire the events needed to trace operations. Of course, the analysis model does not use these subclasses to instantiate values for its calculations; which means we need to wrap every value we want to trace into an instance of one of our subclasses whenever we encounter one. The values we want to track are those we were interested in in the first place: they are the values that are used in arguments to `Analysis Functions`, and the values that are returned from `Analysis Functions`.

With this mechanism, the following information can be captured:
5.3. SKACOST  
CHAPTER 5. CAPTURING ANALYSIS MODEL DATA

```python
class PersonAndBike:
    """Calculates the kinetic energy of a person and a bike travelling at a certain velocity"""
    def __init__(self, personMass, bikeMass, velocity):
        person = MassAndVelocity(personMass, velocity)
        bike = MassAndVelocity(bikeMass, velocity)
        self.KE = bike.KE + person.KE
```

Figure 5.7: A version of the `PersonAndBike` class from figure 5.2 that does its calculation entirely outside any method. The operation that was previously in the `PersonAndBike.calcKE` method is now directly performed in the constructor (the `PersonAndBike.__init__` method) on line 10.

![Diagram](image.png)

**Python**

```python
self.KE = bike.KE + person.KE
```

**Domain model**

- `self.KE = bike.KE.__add__(person.KE)`
- The addition is performed by calling the `__add__` method on `bike.KE`
- The resulting Number is stored in `self.KE`
- The Number stored in `bike.KE` and the Number stored in `person.KE` are the operands

![Diagram](image.png)

Figure 5.8: Example of how a `System Parameter` and its relations to `Numbers` and an `Anonymous Analysis Function` are established and represented in the case where a calculation is not performed inside a method that is an `Analysis Function`. The line of Python code comes from figure 5.7, line 10.
- Dependencies between values (potential Numbers). A value (Number) depends on another value (Number) if the latter was an operand in an operation that resulted in the former (see figure 5.8 on the preceding page).

- The Analysis Function that was being called when the operation that "created" the value was executed. The value is then a potential Number with a calculated by relation to the Analysis Function. It will become a definitive Number once it has been assigned to an attribute that is a System Parameter.

- The Analysis Function that is being called when a value is accessed (i.e., used in a calculation). The value is a potential Number to which the Analysis Function has a takes as input relation.

The last two entail information that can already be captured by the "wrapping methods" mechanism. However, the information captured here is more accurate: even if values are not passed as arguments or returned as results by an Analysis Function, they still get the correct relations with that Analysis Function.

5.3.2.6 Conclusion

The previous two sections (5.3.2.4 and 5.3.2.5) provide the means of creating events in the analysis model at which information is available; with this information, a structure according to our domain model can be built. At an event, a call-back is called that updates the structure with the information passed with the call-back. Six events are available:

1. **Analysis Function call**: The event that occurs just before an Analysis Function method is called.

2. **Analysis Function return**: The event that occurs right after an Analysis Function method has finished executing, but before it actually returns.

3. **Number assignment**: The event that occurs just before a Number is assigned to an attribute.

4. **Number operator call**: The event that occurs just before an operation on a Number is performed.

5. **Number operator return**: The event that occurs right after an operation on a Number has been performed, but just before the operation returns.

6. **Number access**: The event that occurs when a value is "accessed"; this is actually an event based on the last two, and occurs once for each operand just before the actual operation is carried out.

The mechanism needed for (1), (2) and (3), "wrapping methods", is provided by section 5.3.2.4; (4), (5) and (6) are done by "wrapping objects", from section 5.3.2.5. The next section delves into implementation details of how these events are added to the analysis model at run-time, and how they are used to capture the data.

5.3.3 Design and implementation

The pyQAM tool consists of a collection of Python modules and a user interface (pyQAMui) that helps the user to use the modules in the right way. Two of the modules perform the transformations on a Python analysis model that were discussed in the previous sections; they are called `pyqam.trace.ootrace` and `pyqam.trace.valuetrace`. The `pyqam.trace.ootrace` module performs the wrapping of Component methods discussed in 5.3.2.4. `pyqam.trace.valuetrace` performs the wrapping of value objects (potential Numbers) of 5.3.2.5. Together, these components allow call-backs to be specified for each of the six events from the last section.

A third module, `pyqam.alyze`, contains the call-backs which are called by `pyqam.trace.ootrace` and `pyqam.trace.valuetrace` at the six events. These events will therefore be depicted as calls to the `pyqam.alyze` module.
5.3.3.1 Wrapping methods on *Components* with *pyqam.trace.ootrace*

Three of the six events are provided by the *pyqam.trace.ootrace* module: the first two, that fire just before an *Analysis Function* is called (*Analysis Function call*) and just before an *Analysis Function* returns (*Analysis Function return*); and the third, which is fired when a value is assigned to an attribute (*Number assignment*). All of these are implemented by replacing methods on *Component* classes in the analysis model; because these method replacements occur on the class level, they have to be carried out only once for all objects of a class.

![Diagram](image)

(a) Analysis function call

![Diagram](image)

(b) Analysis function call, wrapped by *pyqam.trace.ootrace*

Figure 5.9: Tracing analysis function calls

**Analysis Function call and return events** Figure 5.9 shows how an "original" *Analysis Function* method (5.9a) is replaced with a method which fires the *Analysis Function call* and *Analysis Function return* events, in addition to the original method’s calculations (5.9b). Every method that is specified as an *Analysis Function* is replaced in this way with a wrapped version.

The operations performed at each numbered stage in figure 5.9b are:

1. **Put the analysis function being called on a stack**
   
The name of the *Analysis Function* is put on a global stack that is accessible for subsequent
call-back calls.

2. **Calculate system parameter values**
   This is the original method’s calculation. In this context, nothing special happens to it; however, if some of the values involved in this calculation are wrapped by `pyqam.trace.valuetrace`, callbacks that are not in figure 5.9b are called. These call-backs are described in section 5.3.3.2.

3. **Wrap value object of the return value**
   In most cases, the return value is already a value that is wrapped by `pyqam.value.valuetrace`. To ensure that the value is traced, we wrap it here anyway. The `pyqam.value.valuetrace` detects whether the value has already been wrapped and leaves it alone if that is the case.

4. **Attach information to the return value, remove analysis function from the stack**
   The information available here is that the wrapped value that is to be returned, is a value that is calculated by the *Analysis Function* on top of the stack. Therefore, we associate the wrapped value with a *Number* that has a *calculated* by relation to the *Analysis Function*. This *Number* is a "potential" one; it becomes a definitive *Number* once the associated value has been assigned to an attribute.

![Diagram](image.png)

**Figure 5.10:** Tracing system parameter assignments

**Number assignment event**  Module `pyqam.trace.ootrace` also wraps another kind of method a *Component*: the method that takes care of assigning values to attributes. This is the *__setattr__* method. Given an object `aObject` with an attribute named `bAttr`, the assignment `aObject.bAttr = 10` is executed as `aObject.__setattr__("bAttr", 10)`.
Figure 5.10a shows how a value is stored in the object dictionary, the structure attached to an object that stores attribute values. This method is replaced with one that fires the Number assignment event (figure 5.10b). The operations performed at each numbered stage in this figure are:

1. **Wrap the value object that is being assigned to the attribute**
   The `pyqam.trace.valuetrace` module wraps the value that will be assigned to the attribute. If this value has already been wrapped on a previous occasion, it is left alone.

2. **Attach information to the wrapped value**
   Using information that is attached to the wrapped value (i.e., a Number that is associated with this value, and its dependencies), we determine whether this attribute qualifies as a System Parameter (using the criteria in rule 3 of section 5.3.1). If it does, or if the attribute was already marked as being a System Parameter, the (potential) Number that is associated with the value is a definitive Number that is typed by the System Parameter.

3. **Store the wrapped value into the object dictionary**
   This is the original operation carried out by the `__setattr__` method.

### 5.3.3.2 Wrapping value objects with `pyqam.trace.valuetrace`

The purpose of wrapping value objects is to be able to trace how they are combined into new values through (mathematical) operations, and to be able to attach arbitrary information to them (such as the Number they are associated with).

In contrast to wrapping Component methods, wrapping values occurs on objects; every new object of a class needs to be wrapped in order to be traceable. That is why at almost every event that was described previously, values needed to be wrapped again.

So, what does the "wrapper" that a value object is wrapped in, look like? It is a new instance of a dynamically created subclass of the class of the original value. This wrapper delegates all method calls to the same methods on the wrapped value, in the meantime firing the Number operator call, Number operator return and Number operator access events. Because the wrapper is an instance of a subclass of the wrapped value’s class, its behavior with respect to the analysis model remains the same: type checks, mathematical operations, string conversions are performed as if the original value is used. But because the wrapper is also of a user-defined class, we can add arbitrary attributes to it, which can be used to attach information that is collected about a value to the value itself.

Although this strategy is primarily focused on operator methods (i.e., methods that implement operators, also referred to as operator overloads in other languages), the `pyqam.trace.valuetrace` module doesn’t make a distinction between operator methods and other kinds of methods. The reasoning here is that with any method call on a value object with arguments and return value(s), dependencies of the return value on the argument values and the object itself exist.

### Number operator call, return and access events

Figure 5.11a shows how a normal operation is executed; figure 5.11b shows how a wrapped value performs the same operation with events. The following operations happen at the numbered stages:

1. **Wrap the argument values**
   Although the value object that the method is called on is already a wrapped object, the arguments are not necessarily wrapped. For example: given an object `w` that is a wrapped value and an object `u` that is a non-wrapped value; the operation `w + u` would then be performed by executing `w.__add__(u)`. In this case, `u` is an argument-to-be-wrapped.

2. **Add information to accessed values**
   This happens during the Number access event. This event is fired once for each of the operands (this is not clearly depicted in figure 5.11b); the operands are the object the method is being called on, and the arguments to the method. Each operand has been wrapped in the previous step; in this step, we add information to the wrapped operand. The value is associated with a (potential) Number; the Analysis Function that has been put on top of the stack in step 1 of section 5.3.2.4 gets a takes as input relation to this Number.
3. **Do the original operation**
   The original operation is performed, resulting in a return value.

4. **Wrap the return value**
   The return value is wrapped.

5. **Establish dependency of the return value on the arguments**
   The newly wrapped number gets associated to a new *Number*. This *Number* gets *depends on* relations with the *Numbers* of the operands. Furthermore, the new *Number* gets a *calculated by* relation to the *Analysis Function* that is on top of the stack.
5.3. SKACOST

CHAPTER 5. CAPTURING ANALYSIS MODEL DATA

(a) Operation on a bare value

(b) The same operation on an object wrapped by `pyqam.trace.valuetrace`

Figure 5.11: Tracing operations on objects
5.4 SKAsim

SKAsim is intended to be a simulator; that is, it is an engine that can be invoked with a model definition. The model definition consists of a description in XML that states (among others) which components are part of the analysis model; each component is in turn described in a model definition. If a component needs more complex calculations than the aggregation of a number of other components, it may be described in Python. The mapping of components to classes was abandoned with SKAsim; instead, SKAsim defines a generic component that can be configured through XML files. The advantage of SKAsim over SKAcost is that SKAsim separates the model engine from its definition, (almost) removing the need for analysts to write Python code.

5.4.1 Capturing data from SKAsim

The new challenge with SKAsim is that the object-oriented mapping of Components to classes is abandoned; so is the mapping of System Parameters to object attributes. Therefore, the capturing algorithm that works so well for the SKAcost model cannot be reused for SKAsim.

However, not all is lost. Recall the six events, used to build up the structure in terms of the domain model, from section 5.3.2.6 on page 43:

1. **Analysis Function call**: The event that occurs just before an Analysis Function method is called.
2. **Analysis Function return**: The event that occurs right after an Analysis Function method has finished executing, but before it actually returns.

Figure 5.12: pyQAM’s modules with support for extensions for any analysis model design
3. **Number assignment**: The event that occurs just before a *Number* is assigned to an attribute.

4. **Number operator call**: The event that occurs just before an operation on a *Number* is performed.

5. **Number operator return**: The event that occurs right after an operation on a *Number* has been performed, but just before the operation returns.

6. **Number access**: The event that occurs when a value is accessed.

These events were created by modifying the Python program at run time; the first three by wrapping methods, the last three by wrapping objects. Considering that SKAsim, even though it has a design completely different from SKAcost’s, also is a program that manipulates values in Python variables, the object wrapping technique described in sections 5.3.2.5 and 5.3.3.2 will work just as well for SKAsim. This means the last three events in the list above can be re-used for SKAsim. However, the first three events of the list cannot be reproduced in SKAsim through the method wrapping technique of sections 5.3.2.4 and 5.3.3.1; this technique relies on the design used in SKAcost.

This does not mean that there is no other way to produce these events for SKAsim; or for an analysis model in Python with yet another design. In fact, to enable pyQAM to extract analysis model data from any analysis model design in Python, it is enough to define a way for it to generate the first three events:

1. **Analysis Function call**: The event that occurs just before an *Analysis Function* is called.

2. **Analysis Function return**: The event that occurs right after an *Analysis Function* has finished executing, but before it actually returns.

3. **Number assignment**: The event that occurs just before a *Number* is assigned.

To facilitate the definition of these events for analysis model designs in Python, pyQAM’s analysis engine (see figure 5.12 on the previous page) is set up to be extensible with these definitions. The red boxes in this figure are the extensible subsystem. In this system, the `pyqam.hook.adapter` module provides an interface to which event definitions can be hooked up; note the definitions that are already present, `pyqam.hook.oo` and `pyqam.hook.skacost`. The `pyqam.hook.oo` module contains the event definitions for object-oriented analysis models that map *Components* to classes and *System Parameters* to object attributes; `pyqam.hook.skacost` is an extension to this definition that contains specific adjustments for the SKAcost analysis model.

Next to `pyqam.hook.oo` and `pyqam.hook.skacost`, there is a red box labeled `pyqam.hook.skasim`, with one uses relation to the `pyqam.hook.adapter` module. To be able to extract analysis model data from SKAsim, this module needs to contain functionality to decorate SKAsim with definitions for the three events that were mentioned. Unfortunately, this has not been done because there was not enough time remaining to do so.

### 5.5 Domain models

The domain model used by the *Excel Knowledge Extractor* to represent analysis model data is discussed in section 3.3.1.1; part of it is depicted in figure 5.1 on page 34. This figure only contains the parts that are useful for quantitative modeling.

The quantitative part of the domain model defined by [2] was set up to support analysis models in general, not just analysis models in Excel. That is why it is capable of representing more than what is possible in Excel. However, the only analysis models that were represented using this domain model, were made in Excel; it has not been tested to work with analysis models in other forms, and it does miss some expressiveness when it comes to Python analysis models.

Therefore, some extensions and changes were made to the domain model to support Python analysis models. The quantitative part of the domain model for Python analysis models is depicted in figure 5.13 on the next page; a mapping of the entities in figure 5.13 onto Python elements is in table 5.1 on page 52. In the following, the differences between the used domain models, the reasons for these differences and their implications are discussed.
Figure 5.13: Domain model for Python analysis models
5.5. DOMAIN MODELS

### 5.5.1 Analysis Function, System Parameter, Number

An *Analysis Function* may be reusable in a generic analysis model; when the *Analysis Function* is calculated twice, both calculations yield a *Number*. However, both of these *Numbers* are a result of the same calculation (possibly with different parameters); therefore they are the same *type* of value, which is indicated by the *is of type* relation with *System Parameter*. An *Analysis Function* always results in a *Number* that is *typed by* the same *System Parameter*.

**Excel** In the Excel case, formulas stored in cells are *Analysis Functions* in the domain model. Values calculated by *Analysis Functions* are *Numbers*; so are the values that are input to *Analysis Functions*. Each *Number* has an *is of type* relation with a *System Parameter*, which represents a cell in Excel.

In an Excel analysis model, it is not possible to use a formula in one cell more than once with different parameters. Therefore, there is a one-to-one relation between *Number* and *System Parameter*, and not a many-to-one relation like in the domain model. Because a formula in a cell can only produce one result, the relations of *Number* and *System Parameter* to *Analysis Model* are also one-to-one. Effectively, every cell with a formula corresponds to one *Analysis Function*, one *Number* and one *System Parameter*, while the domain model allows multiple *Numbers* per *System Parameter*. A result of this is that although the analysis model may use the same calculation for different numbers, this is not apparent in the extracted structure.

**Python** Capturing *Analysis Functions*, *System Parameters* and *Numbers* has extensively been described in sections 5.3.2 and 5.3.3. The main difference between the relations between these and those in the Excel case, is that an *Analysis Function* can be re-used with different parameters. This means that now, we do have a many-to-one relation between *Number* and *System Parameter* in the domain model.

Moreover, an *Analysis Function* may even calculate multiple *System Parameters*. Suppose an *Analysis Function* does not return the result of the calculation it contains, but directly assigns it to an attribute that is a *System Parameter* (see rule 3(c) in section 5.3.1 and section 5.3.2.5): the author of the analysis model might decide to put more than one calculation into one *Analysis Function*, assigning each result to a different *System Parameter*. One could argue that this is bad...

---

<table>
<thead>
<tr>
<th>Domain Model</th>
<th>Python Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Function</td>
<td>Method on a <em>Component class</em></td>
</tr>
<tr>
<td>System Parameter</td>
<td>Attribute on an instance of a <em>Component</em> class</td>
</tr>
<tr>
<td>Component</td>
<td>Class that models a part (component) of the system.</td>
</tr>
<tr>
<td>Number</td>
<td>Value in an attribute that is a <em>System Parameter</em></td>
</tr>
<tr>
<td>Component Instance</td>
<td>An instance of a <em>Component</em> class. A <em>Component Instance</em> groups together <em>Numbers</em> that belong to one instance of a <em>Component</em>.</td>
</tr>
<tr>
<td>Value</td>
<td><em>Number</em> that is input to the analysis model: it is not calculated. All <em>Numbers</em> that have no <em>depends on</em> relations to other <em>Numbers</em> are <em>Values</em>.</td>
</tr>
<tr>
<td>Analysis Output</td>
<td>All <em>Numbers</em> that are not <em>Values</em> are <em>Analysis Outputs</em></td>
</tr>
<tr>
<td>Analysis Result</td>
<td>Some of the <em>Analysis Outputs</em> are <em>Analysis Results</em>. There is no way of determining automatically which <em>Analysis Outputs</em> qualify to be <em>Analysis Results</em>; some of them may be the <em>Analysis Outputs</em> that are not input to any <em>Analysis Function</em>. However, the majority of <em>Analysis Results</em> must be manually labeled as such by a domain expert.</td>
</tr>
<tr>
<td>Scenario</td>
<td>A set of input values for an analysis model</td>
</tr>
<tr>
<td>Artifact</td>
<td>Python source file</td>
</tr>
</tbody>
</table>

Table 5.1: Domain model to Python mapping for SKAcost
programming practice; but it should nevertheless be represented in the captured analysis model structure. Therefore, an Analysis Function can calculate multiple System Parameters; each of these System Parameters can type multiple Numbers.

Another difference with the Excel Knowledge Extractor is, that Analysis Functions need to be specified in the Python tool, while the Excel tool can detect them. This is due to the fact that part of a Python analysis model is code to support the analysis model. Excel itself takes care of these support tasks; a Python program needs to provide them in the same code as the analysis model itself.

5.5.2 Value, Analysis Output, Analysis Result

Excel The Excel Knowledge Extractor designates Numbers that are input to an Analysis Function to be Values. Outputs of Analysis Functions that are not used as input by other Analysis Functions are Analysis Results. The remainder of the Numbers (i.e., the "intermediate" results) are Analysis Outputs.

Python Although superficially, the Python tool does make a distinction between Values and other Numbers. A value that has not been calculated by an Analysis Function is a Value; other values are Analysis Outputs.

Analysis Results are left to the expert user to determine. Although it is possible to mark the "end" outputs (i.e., Analysis Outputs that are not used as input to any Analysis Function) as Analysis Results, in practice not all "end" outputs are considered Analysis Results by domain experts.

5.5.3 Component, Component Instance

Excel The Excel Knowledge Extractor does not group System Parameters into Components. An Excel element that could map to Component, is the worksheet: all System Parameters that are on one worksheet would then belong to the Component the worksheet corresponds to. In practice, however, this may not be very helpful: because Excel allows just about anything to be put into a worksheet, a worksheet could easily contain data that does not represent a Component. This is probably the reason why Components aren’t defined.

Python In contrast to the Excel case, in the SKAcost case we do have a rigid rule about what a Component is: an Analysis Function is always member of a class that is a Component. Therefore, the Python tool can group System Parameters into Components.

While trying to visualize number tables (see the next chapter), the need for a Component Instance entity in the domain model became apparent. A Component Instance groups all Numbers that belong to one instance of a Component class; Number has the same is of type relation to Component Instance as System Parameter has to Component.

5.5.4 Scenario

Excel A Scenario groups together the set of input values for an analysis model. As mentioned before, Excel does not allow the same calculation to be re-used with different parameters; therefore, an Excel analysis model is associated to only one Scenario.

The Scenario’s name is specified in annotations to the analysis model; hence its yellow color in the domain model on page 34. This Scenario has a sets relation to all Numbers that are not calculated by the analysis model (input values to the analysis model).

Python As in the Excel case, a Scenario groups together a set of Values that are input to the analysis model. In the Python case, a Scenario can be seen as one invocation of the analysis model with a set of values; each time analysis model data is captured with a different set of input values, this corresponds to a different Scenario.

The Scenario is another element that needs to be explicitly labeled by the user of the tool. All the data captured in one run of the tool is associated with one Scenario.
5.5.5 Artifact

**Excel** The domain model states that an *Analysis Model* is contained in only one *Artifact*. In case of the SKADS analysis model in Excel, this is true: the *Analysis Model* is contained in one Excel file. This is not true in general; in fact, this is even specific to the SKADS analysis model: even Excel allows calculations to be spread over multiple files, therefore an analysis model in Excel does not have to be contained in only one *Artifact*.

**Python** Python programs are typically spread over different source files. Also, it is possible that one source file is part of multiple analysis models. Therefore, the relation between *Analysis Model* and *Artifact* in the new domain model is a many-to-many relation.
Chapter 6

Visualizing Analysis Models

6.1 Introduction

![Image of Excel spreadsheet](image1)

(a) PersonAndBike in Excel

![Image of visualization](image2)

(b) Visualization of PersonAndBike

Figure 6.1: The example analysis model PersonAndBike in Excel and the visualization that the Excel Knowledge Extractor creates

One way to have analysts understand an analysis model more easily and quickly is to provide a visualization of the analysis model in such a way that it becomes easier to understand for the analyst. For this, the visualization needs to be intuitive; that is, it must fit naturally into aspects of analysis on an analysis model (see section 3.3.3). It also needs to leave out or de-emphasize information that is less important, to enable the analyst to quickly spot good starting points for analysis and to prevent information overflow.
To satisfy the intuitiveness requirement, the aspects of analysis discussed in section 3.3.3 can be taken as a basis for visualization. **Dependency analysis** and **significance analysis** (section 3.3.4.1) provide the structure of visualization and a means to partition the visualizations into manageable pieces. They also give a way to de-emphasize less important parts of an analysis model in favor of the more important ones.

The Knowledge Architect Excel Client already visualizes analysis model data in a similar fashion. An example of this visualization is in figure 6.1. The table in figure 6.1a shows the inputs and the outputs of an Excel analysis model that calculates the kinetic energy of a person and a bike. Figure 6.1b shows the dependency tree for this calculation, which is generated by the Knowledge Architect Excel Client. This visualization is specifically tailored for analysis models in Excel; not only is it built as a plug-in to the Excel program, it also doesn’t visualize **Analysis Functions** from the domain model in section 3.3.1.1.

This chapter describes a number of visualizations, the first of which resembles the visualization the Excel Client features (see section 6.3). All visualizations take into account a number of common principles; to ensure consistency across the visualizations, to make sure they are format (Excel, Python, ...) independent, and because they are just good principles for a visualization to live by. The next section (6.2) lists these common principles and the reasoning behind them. The subsequent sections describe the visualization of an analysis model as a graph (6.3), as lists of **System Parameters** (6.4) and as a nested table of **Numbers** (6.5).

### 6.2 Common principles

Three common principles were used in the design of the visualizations.

1. The visualizations described in this chapter are not specifically tailored to a format (e.g. Python, Excel). Rather, they work on data extracted from an analysis model, expressed in terms of the domain model of section 3.3.1.1. This means that in order to visualize them, data extraction has to be performed on an analysis model first. The advantage of this approach is, that the visualization can be used on any analysis model for which an extraction algorithm has been implemented that can express an analysis model in the domain model. Therefore, the only assumption made by a visualization with respect to the analysis model must be that it is expressed in terms of the domain model.

2. The domain model contains a **Component** class; the instances of this class define a logical subdivision of the analysis model. The visualizations are partitioned according to this subdivision in order to keep the presented information manageable to human beings.

3. In contrast to the extraction of analysis model data, its visualization focuses more on understandability than correctness. For example, sorting on elements in the visualization is an order that is probably correct: **Knowledge Entities** that are considered more significant are given a more prominent place in the visualization, even though we cannot be entirely sure of the order of significance of **Knowledge Entities**.

### 6.3 Graph visualization

#### 6.3.1 Dependency graph of **System Parameters**

As explained in section 3.3.3.1, one view on an analysis model is that of the dependences between values. A **System Parameter** represents values with the same semantic meaning; an assumption is that each value with the same semantic meaning is calculated in the same way, i.e., values with the same semantic meaning have the same reverse dependencies. The choice was made, to have a visualization contain as little repetitions of the same structures as possible, to visualize the dependences of these semantic meanings, not of the values themselves.

A value depends on another value if the former is the result of a calculation that uses the latter as input. Reformulating this in terms of the domain model, a **System Parameter** $a$ depends on another **System Parameter** $b$ if the **Analysis Function** calculating $a$ has $b$ as one of its inputs. In
a graph, one can represent this by a directed edge (arrow) from a node labeled $a$ to another node $b$; the edge has semantic meaning depends on. Of course, System Parameter $b$ may in turn have depends on relations to other System Parameters; $a$ may have depends on relations to yet some other System Parameters; and so forth. The graph thus constructed is the dependency graph for $a$.

The graph of all direct and indirect reverse dependencies that are involved in the calculation of a System Parameter $a$ has the following properties:

- A System Parameter cannot depend on itself; therefore, there are no cycles in the graph.
- Only System Parameters that are involved in the calculation of $a$ are in the graph; therefore, the graph is connected.
- There is one node (the root node) that has only outgoing edges. This is the node that represents system parameter $a$.
- A node with only incoming connections (a leaf node) is a System Parameter that represents a constant value that is input to the calculation: this is an an assumption for the calculation of $a$.
- All other nodes are System Parameters that represent intermediate results for $a$.

The graph contains no cycles, is connected and has one node with only outgoing edges; which makes it a directed tree. An example of such a dependency tree is in figure 6.2.

Generally, analysis models calculate more than only one result. For each of the results, a dependency tree of its corresponding System Parameter can be made. This produces a collection of directed trees; one for each System Parameter that doesn’t have any dependencies. These trees have certain parts in common; this occurs in places where multiple results depend on one intermediate result.

Combining the directed trees into one graph means mapping their common parts onto each other; see figure 6.3 for how this is done. This produces a graph that has the following properties:
6.3. GRAPH VISUALIZATION

6.3. VISUALIZING ANALYSIS MODELS

- There are no cycles in the graph.
- Only System Parameters that are involved in the calculation of other System Parameters and System Parameters that are results of calculations involving other System Parameters are in the graph; therefore, the graph is connected.
- There is at least one node that has only outgoing edges. These are the nodes that represent System Parameters that are not input to any Analysis Function.
- A node with only incoming connections is a System Parameter that represents a constant value that is input to the calculation: this is an assumption of the analysis model.
- All other nodes are System Parameters that are intermediate results of the analysis model.

A graph with these properties is a directed acyclic graph (DAG).

6.3.2 Adding Analysis Functions

Figure 6.4: The dependency graph of System Parameters extended with Analysis Functions

The dependency graph obtained in the previous section can give an analyst the insight she needs into the analysis model; that is, she can find out on which other System Parameters a result depends. However, she cannot see how and where these dependencies are defined: for this, the formulas or (in terms of the domain model) Analysis Functions in which a System Parameter is calculated must get a place in the visualization.

Analysis Functions are added to the graph as nodes; these nodes have a shape different from System Parameter nodes to avoid confusion. Because every System Parameter that is calculated has to be a result of an Analysis Function (per the domain model), all dependences in the dependency graph of only System Parameters need to be re-routed through Analysis Function nodes. If in the System Parameter only graph, there is a directed edge from a node a to a node b, then it is replaced in the graph extended with Analysis Functions in the following way:
If a System Parameter $a$ is calculated by an Analysis Function $\text{calc}_a$, there is a directed edge from $a$ to $\text{calc}_a$; this edge corresponds to the calculated by relation between System Parameter and Analysis Function in the domain model.

If an Analysis Function $\text{calc}_a$ uses a System Parameter $b$ to calculate $a$ (i.e., $a$ depends on $b$), there is a directed edge from $a$ to $b$; this edge corresponds to the takes as input relation between Analysis Function and System Parameter in the domain model.

For a graphic depiction of this process, see figure 6.4.

### 6.3.2.1 Analysis Function Visualization Challenges

Ideally, a System Parameter is calculated by only one Analysis Function, and an Analysis Function calculates exactly one System Parameter. Unfortunately, this is not a fact that can always be taken for granted. The strategy of capturing domain model semantics from an analysis model inevitably makes assumptions about the structure of the analysis model and the patterns used. But because analysis models aren’t created by machines but by human beings, they can deviate from these structural rules and patterns. Therefore it is possible that:

- Multiple System Parameters are calculated by one Analysis Function, or
- One System Parameter is calculated by multiple Analysis Functions

The strategy of capturing domain model semantics from an analysis model tries to prevent these cases; and when they do appear, it is likely that they indicate a bad practice or an error. However, when they do appear, they have to be visualized.

![Diagram of Analysis Function Visualization Challenges](image)

Figure 6.5: Multiple System Parameters calculated by one Analysis Function. Given is a case where $a$ and $b$ are System Parameters that are both calculated by Analysis Function $\text{anafunc}$. Figure (i) shows the dependences between System Parameters; (ii) shows how these dependences are hidden when the Analysis Function comes into the picture. Figures (iii) and (iv) show solutions to this problem: (iii) replicates the Analysis Function, (iv) uses alternative edge colors / weights to indicate which System Parameters depend on each other. Solution (iii) is still confusing because it does not map intuitively to the analysis model; (iv) still hides part of the dependences (dependence of $b$ on $d$ cannot be seen in this picture).
Multiple System Parameters calculated by one Analysis Function  In the context of only one analysis model, multiple System Parameters calculated by one Analysis Function means that either of two things happen:

- Although two System Parameters have different semantic meaning, they can be calculated by the same formula or algorithm. This happens mostly when a general utility formula/algorithm has the role of Analysis Function; e.g. an Analysis Function that performs a fast Fourier transform might be used to calculate multiple System Parameters with different semantic meanings.

- The Analysis Function in question contains an algorithm that results in more than one System Parameter. For example, in an algorithm that calculates the average of a set of values, the sum of these values can also be acquired with little effort. Instead of calculating this sum again in a separate Analysis Function, the analysis model designer may have decided to have this Analysis Function calculate both the sum and average values; which means it calculates two System Parameters.

These cases can be confusing in the visualization with System Parameters and Analysis Functions (illustrated in figure 6.5). One solution is to modify the capturing strategy so that it doesn’t. For example, a strategy could be to represent one bit of calculation with multiple Analysis Functions, one for each System Parameter it calculates. However, as the rules the capturing strategy applies to the analysis model become more complex, it becomes harder for the analyst to see the correspondence between the visualized structure and the analysis model in its original form; possibly making it easier to find dependencies, but harder to understand the model. Therefore, a choice has been made to leave these cases intact, and visualizing them as they are: that is, Analysis Functions can calculate multiple System Parameters.

If this means that the visualization is not understandable, it is left to the analyst or the analysis model writer to restructure the analysis model itself, by:

- Relocating, renaming or wrapping "general utility" functions so they are not considered Analysis Functions anymore by the capturing algorithm.

- Rewriting algorithms in Analysis Functions that calculate more than one System Parameter. This means an Analysis Function needs to be split up into multiple Analysis Functions.

One System Parameter calculated by multiple Analysis Functions  Recall that a System Parameter does not represent one value, rather a collection of values with the same semantic meaning. It is possible that in one analysis model, different values that are typed by the same System Parameter, are calculated by different Analysis Functions. If this happens, either the data capturing strategy is wrong, or the analysis model is sloppy.

The same choice is made as with the case of multiple System Parameters being calculated by the one Analysis Function: the structure is visualized nonetheless, and it is left to the model analyst or the model writer to modify the analysis model so that it doesn’t contain these quirks.

6.3.3 Adding Numbers

In most cases, a visualization with only Analysis Functions and System Parameters contains enough information for the analyst to perform analysis on. In some cases, it doesn’t; especially in those cases where the analysis model contains the bad-practice constructs or errors discussed in the last section.

In cases such as these, the actual values, represented by Number in the domain model, can be added to the graph in order to reveal finer-grained dependencies. A Number is represented by a node with a shape different from System Parameter and Analysis Function nodes; the following edges connect it to the graph of System Parameters and Analysis Functions:

- A Number’s is of type relation to a System Parameter is represented by a directed edge from the Number’s node to the System Parameter’s node.
CHAPTER 6. VISUALIZING ANALYSIS MODELS  6.4. SYSTEM PARAMETER LISTS

- A Number’s calculated by relation to an Analysis Function is represented by a directed edge from the Number’s node to the Analysis Function’s node.

Furthermore, there are edges between Number nodes:

- If a Number depends on another Number, there is a directed edge from the former to the latter with semantic meaning depends on.

6.3.4 Adding Components and Component Instances

A graph of System Parameters, Analysis Functions and Numbers can become quite large. To enhance the understandability of the graph, we partition it into logical components. Luckily, the domain model defines the Component and Component Instance classes which does just that: partition the analysis model into logical parts.

Applying the partitioning of the graph according to Components is straight-forward:

- If a System Parameter has a part of relation to a Component, the node of that System Parameter is displayed in a box carrying the name of that Component.

- If a Number has a part of relation to a Component Instance, the node of that Number is displayed in a box carrying the name of the Component Instance.

- If a System Parameter has a calculated by relation to an Analysis Function, the Analysis Function is displayed in the same Component box as the System Parameter.

The result is a graph in which groups of nodes are grouped together in boxes that carry the names of the Components they represent. This enables the analyst to think in abstractions: she can disregard any information in Components that are of less interest, focusing on the structure within one Component. Furthermore, it becomes possible to reason about the analysis model’s structure on a higher level: dependencies between Components, instead of between mere System Parameters or Numbers, can now be analyzed more easily.

6.4 System Parameter Lists

When doing dependency analysis, the goal is not always to gain a better understanding of the analysis model as a whole; sometimes, it is just to determine on which other values a value depends, or which other value depend on a value. Of course, the graph views presented before can be used for this: following edges in a graph is almost always easier than skipping through the analysis model in its original form. However, edges can become quite long in a big analysis model; and if the analyst is only viewing part of the graph, edges to other parts may be missing in the visualization.

Another downside to the graph visualization(s) is that it doesn’t give an ordering on significance of System Parameters; although nodes that are end results and nodes that are assumptions can be easily spotted and used as starting points to analysis. Again, as the analysis model is larger, the number of end results and assumptions is higher, decreasing the value of the graph visualization to find starting points.

The view that solves these issues, is that of the hierarchical list view of System Parameters, or list view for short. The list view lists all System Parameters, grouped by their Components, in order of probable significance. As explained in section 3.3.4.1, System Parameters have two roles based on which significance can be determined: the role of assumption and the role of result. This poses the problem of which role to take into account when determining significance. Practice learns that either of the two roles is more important in different scenarios: the assumption role determines a System Parameter’s significance if the analyst decides to start dependency analysis starting at the analysis model’s assumptions (forward dependency analysis, a bottom-up approach to the graph of the previous section); and its result role determines significance if analysis is started at the analysis model’s results (reverse dependency analysis, top-down). Therefore there are two list visualizations: one that treats System Parameters as assumptions, and one that treats them as results.
(a) Graph of System Parameters and Analysis Functions and their relations for PersonAndBike

(b) Graph of Analysis Functions, System Parameters, Numbers and the relations between them for PersonAndBike.

Figure 6.6: Two graph visualizations
6.4.1 Reverse dependency analysis (top-down)

1. Analysis Model: PersonAndBike example

1.1. Scenario: example.pyqam

1.1.1. Component: PersonAndBike

- PersonAndBike.KE
  - 1543.29987654

- MassAndVelocity.KE
  - 308.541976309
  - 1234.56790123

  - MassAndVelocity.mass
    - 20.0
    - 80.0

  - MassAndVelocity.velocity
    - 20.0
    - 20.0

Figure 6.7: Top-down System Parameter List

This visualization is also referred to as System Parameter List in later chapters and an example is depicted in figure 6.7. The top-down hierarchical system parameter list starts by listing all System Parameters that are the result of a calculation. Then, each of these System Parameters can be expanded to display a nested list of System Parameters, such that each member of a sub-list is dependent on its parent. The System Parameters in sub-lists may in turn depend on other System Parameters; these, too, can be expanded to show the System Parameters they depend on. When fully expanded, the System Parameters on the deepest levels of the list (i.e., System Parameter entries that cannot be expanded any more) represent assumptions to the analysis model.

On each level of the hierarchy, the System Parameters are sorted. The ordering is obtained by calculating a rating for each System Parameter; this rating is described in section 3.3.4.1 on page 16 (on significance of values in an analysis model), under Influence on other values.

6.4.2 Forward dependency analysis (bottom-up)

This visualization is also referred to as Assumption List in later chapters and is depicted in figure 6.8 on the next page. The bottom-up hierarchical System Parameter list works the other way round: this one starts with a list of System Parameters that are assumptions. Each of these can be expanded to display a nested list of System Parameters, such that each member of a sub-list depends on its parent. When fully expanded, the System Parameters on the deepest levels of the list (which cannot be expanded any more) represent the System Parameters on which no other System Parameters depend.

On each level of the hierarchy, the System Parameters are sorted. The ordering is obtained by calculating a rating for each System Parameter; this rating is described in section 3.3.4.1 on page 16 (on significance of values in an analysis model), under Depth of the reverse dependency tree.

6.5 Table of Numbers

An entirely different view on the analysis model, also derivable from the dependency graph and inspired by the way the SKA Excel analysis model tends to be structured, is the Table of Numbers. In this view, all values in the analysis model (represented by Numbers in the domain model) are displayed in tables in a way that is meant to be intuitive to someone who has worked with or
6.5. TABLE OF NUMBERS

CHAPTER 6. VISUALIZING ANALYSIS MODELS

1. Analysis Model: PersonAndBike example

1.1. Scenario: example.pyqam

1.1.1. Component: MassAndVelocity

- MassAndVelocity.mass
  - Value: 20.0
- MassAndVelocity.KE
  - Value: 308.641975309
  - Formula: Value: 1284.56790123
- PersonAndBike.KE
  - Value: 1543.20987554

- MassAndVelocity.velocity
  - Value: 20.0
- MassAndVelocity.KE
  - Value: 308.641975309
  - Formula: Value: 1284.56790123
- PersonAndBike.KE
  - Value: 1543.20987554

Figure 6.8: Bottom-up Assumption List

authored an analysis model with the Excel program. The reason to do so, is that this facilitates comparing analysis models in Excel to analysis models in other forms.

To clarify how this way of visualizing an analysis model is inspired on the SKA Excel analysis model, a brief example is discussed in the following section. In the subsequent sections, the "Table of Numbers" visualization is explained step by step.

6.5.1 Tables in Excel

Excel, being a spreadsheet program, is well suited to display data in tables along with calculated results. The analysis model for SKA in Excel uses a table lay-out of which an example is depicted in figure 6.9.

The table of importance in this figure is the one labeled "Total SKA"; the result it presents is "Grand Total". The value for Grand Total is a mathematical combination (actually, the sum) of the values that are contained in the cells above the cell holding the Grand Total. Each of these values is in turn the result of a calculation: each of the "Total per block" values is a mathematical combination (multiplication) of the values in the two cells left of them.

Following this pattern, any calculation in two steps can be put into a table. The first step is laid out horizontally, with the inputs to the calculation next to each other and its result on the extreme right. The second step is a calculation with as inputs the results of the former; the result of the second calculation is put underneath the column of results of the first step.

In a lot of cases, the "first steps" needed to calculate the inputs to the second step involve the same formula, with the same types of inputs: for example, each of the calculations in the horizontal direction in figure 6.9 has a "cost each" and a "quantity" input. It is good practice that when two "horizontal" calculations use inputs with the same semantic meaning ("cost each", "quantity", etc), the inputs with the same semantic meaning are lined up in the same column.

Unfortunately, the calculations for values in an analysis model involve more than two steps. Because Excel is not (yet) capable of extending the table into three or more dimensions, if the
Figure 6.9: Analysis Model in Excel: calculation in tables. Three levels in the dependency tree for the Grand Total are indicated with red, blue and green arrows.
calculation is in more than two steps, all steps but the last two have to be depicted in auxiliary tables. In figure 6.9 on the previous page this is depicted by the green arrows that leave the "main" table: the values in the "cost each" column are actual values that were the results of calculations in tables elsewhere. Note that figure 6.9 is a simplification of the real thing: the actual analysis model in Excel contains hundreds of tables.

This way of representing calculations in the Excel analysis model was an inspiration for the visualization "Table of Numbers". The visualization described in the following sections is not exactly the same as the Excel counterpart. This is due to the fact that the human authors of the Excel model didn't blindly follow lay-out rules, but used their intuition to display the calculations as understandably as possible. Another slight difference is that "auxiliary tables" such as the ones that were needed in figure 6.9, are in effect nested tables in the visualization.

In the following, the visualization of numbers in tables will be explained in three parts. First, the table of numbers for one "step" will be treated: this is a table lay-out that shows the inputs and the result of one formula (Analysis Function) in a table. Secondly the table in two steps is explained; here, the inputs for the formula calculating the end result for the table are intermediate results of other formulas; inputs, intermediate results and result are all in one table. Finally, this idea can be extended to multiple steps by rendering a nested table for each input value in the table.

6.5.2 Table of Numbers: one step

Section 6.2 stated that the only assumption a visualization may make is that the analysis model is expressed in terms of the domain model. Therefore, the representation in Excel needs to be mapped to domain model semantics; this section and the following sections define this mapping.

We start with a calculation in one step, i.e., a "column" or a "row" of input values with at the end a result value in the Excel representation. Let \( f \) be a formula that is an Analysis Function and \( z_1, y_1, y_2, \ldots, y_n \) be System Parameters such that:
CHAPTER 6. VISUALIZING ANALYSIS MODELS  6.5. TABLE OF NUMBERS

\[ z_1 = f(y_1, y_2, \cdots, y_n) \]

And let \( \text{val}(y_n) \) denote a \textit{Number} that is typed by \textit{System Parameter} \( y_n \). Figure 6.10a shows a graph representation of this one-step calculation. \( z_1 \) depends on \( y_1, y_2, \ldots, y_n \) through \textit{Analysis Function} \( f \). The same calculation is shown in tabular representations in figures 6.10b and 6.10c; in the former, input values are lined up in a column with the calculation result beneath; in the latter, input values are in a row with the calculation result as the last value.

### 6.5.3 Table of Numbers: two steps

![Graph representation](image)

<table>
<thead>
<tr>
<th></th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( \cdots )</th>
<th>( x_m )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi_1 )</td>
<td>\text{val}(x_{1,1})</td>
<td>\text{val}(x_{2,1})</td>
<td>\cdots</td>
<td>\text{val}(x_{m,1})</td>
<td>\text{val}(y_1)</td>
</tr>
<tr>
<td>( \phi_2 )</td>
<td>\text{val}(x_{1,2})</td>
<td>\text{val}(x_{2,2})</td>
<td>\cdots</td>
<td>\text{val}(x_{m,2})</td>
<td>\text{val}(y_2)</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\cdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( \phi_n )</td>
<td>\text{val}(x_{1,n})</td>
<td>\text{val}(x_{2,n})</td>
<td>\cdots</td>
<td>\text{val}(x_{m,n})</td>
<td>\text{val}(y_n)</td>
</tr>
<tr>
<td>( z_1 )</td>
<td>\text{val}(z_1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Visualized vertically

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( \cdots )</th>
<th>( x_m )</th>
<th>( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \phi_1 )</td>
<td>\text{val}(x_{1,1})</td>
<td>\text{val}(x_{1,2})</td>
<td>\cdots</td>
<td>\text{val}(x_{1,n})</td>
</tr>
<tr>
<td>( \phi_2 )</td>
<td>\text{val}(x_{2,1})</td>
<td>\text{val}(x_{2,2})</td>
<td>\cdots</td>
<td>\text{val}(x_{2,n})</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\cdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( \phi_n )</td>
<td>\text{val}(x_{m,1})</td>
<td>\text{val}(x_{m,2})</td>
<td>\cdots</td>
<td>\text{val}(x_{m,n})</td>
</tr>
<tr>
<td>( z_1 )</td>
<td></td>
<td>\text{val}(z_1)</td>
<td>\text{val}(y_1)</td>
<td>\text{val}(y_2)</td>
</tr>
</tbody>
</table>

(c) Visualized horizontally

Figure 6.11: Table of a calculation in two steps

The calculation in two steps proceeds similarly to the way tables in Excel are composed for calculations in two steps. Suppose the input parameters \( y_1, y_2, \ldots, y_n \) to the formula \( f \) that
calculates $z_1$ in are in turn calculated by Analysis Functions $\phi_1, \phi_2, \ldots, \phi_n$:

$$y_k = \phi_k(x_1, x_2, \ldots, x_m) \quad 1 \leq k \leq n$$

In this case, $\text{val}(x_1, n)$ denotes the $n^{th}$ Number that is typed by System Parameter $x_1$. Figure 6.11a shows a graph representation of this two-step calculation. $z_1$ depends on $y_1, y_2, \ldots, y_n$ through Analysis Function $f$; and on $x_1, x_2, \ldots, x_m$ through $\phi_1, \ldots, \phi_m$. The same two-step calculation is shown in tabular representations in figures 6.11b and 6.11c.

### 6.5.4 Table of Numbers: multiple steps

The last step, extending the visualization to support calculations with an arbitrary number of steps, is a bit different from the Excel practice. Remember that in Excel, when a calculation is done in more than two steps, it is spread out over multiple tables; each table contains a part of the calculation and the tables refer to each other. These references are a feature of Excel that can be confusing: they are not easily traced, especially if the referred cell is in another worksheet.

It was decided to remove the need to reference auxiliary tables by nesting the auxiliary tables inside the table where the referenced value is used. The nested table can be collapsed to show only the result of a table; or it can be expanded to show how a certain value was calculated. A calculation with arbitrarily many steps can be represented by nesting tables into each other until each step has been covered. Figure 6.12 shows how this nesting of tables works for calculations of up to four steps. Figure 6.13 shows the actual visualization that was implemented, performed on the PersonAndBike example analysis model from section 5.3 on page 35.
CHAPTER 6. VISUALIZING ANALYSIS MODELS

6.5. TABLE OF NUMBERS

Figure 6.12: Mapping a dependency graph of Numbers in a calculation to a (nested) table.
In the middle, dependency graphs of Numbers in an arbitrary calculation are depicted. Number nodes with the same color are typed by the same System Parameter.
On the left, the graphs of System Parameters and Analysis Functions are shown.
On the right, is the "Table of Numbers" representation.
1. **Analysis Model: PersonAndBike example**

1.1. **Scenario: example.pyqam**

1.1.1. **Component: PersonAndBike**

1.1.1.1. **KE**

<table>
<thead>
<tr>
<th></th>
<th>mass</th>
<th>velocity</th>
<th>calcSystemKE</th>
</tr>
</thead>
<tbody>
<tr>
<td>calcPartKE</td>
<td>80.0</td>
<td>20.0</td>
<td>1234.56790123</td>
</tr>
<tr>
<td>calcPartKE</td>
<td>20.0</td>
<td>20.0</td>
<td>308.641975309</td>
</tr>
<tr>
<td>KE</td>
<td></td>
<td></td>
<td>1543.20987654</td>
</tr>
</tbody>
</table>

Figure 6.13: Number Table: Table of Numbers and how they relate to each other through Analysis Functions in the PersonAndBike example analysis model.
Chapter 7

Comparing Analysis Models

7.1 Introduction

To compare two analysis models, one first has to have a sufficient level of understanding of the individual analysis models. That is why chapter 5 (Capturing Analysis Model Data) and 6 (Visualizing Analysis Models) focused on only one model: if an analyst is able to quickly learn everything about individual analysis models, she is also able to perform a comparison between two analysis models faster. The data capturing and visualization techniques for individual analysis models serve as a basis for the visualization of differences between analysis models. It is assumed in this chapter that for each of the analysis models that are to be compared, data is available that expresses the analysis model in terms of the domain model of section 3.3.1.1 on page 11; to acquire this data, a capturing program is needed such as the one discussed in chapter 5.

The first sections in this chapter discuss how captured data from two analysis models can be transformed into one body of data that can be represented in one visualization. Firstly, this involves mapping the domain models used by the different analysis models to each other. Because the capturing methods are assumed to capture data in terms of the same domain model, this problem is already largely solved. However, even though represented with the same domain, data captured from an analysis model in one format might have an entirely different structure than an analysis model in another (cf. section 5.5). The implications of this are discussed in section 7.2.

When domain models have been mapped, the next step is to map entities with the same semantic meaning to each other, described in section 7.3. Finally, visualization is treated in 7.4.

7.2 Mapping domain models

The title of this section is somewhat silly: as captured data from each of the analysis models to compare is expressed in the same domain model, mapping the domain models in which they are expressed would involve mapping one domain model to itself. However, considering that two analysis models may be built in different formats, e.g. in Python and in Excel, it is possible – even probable – that System Parameters captured from one analysis model are different beasts than System Parameters captured from the other analysis model. The same holds for Analysis Functions, Components and Component Instances.

An example is the difference between the System Parameters that are captured by the Excel Knowledge Extractor from an Excel analysis model, and the System Parameters captured by pyQAM from a Python analysis model. Recalling section 5.5, each System Parameter in the Excel case corresponds to exactly one Number and exactly one Analysis Function, while in the Python case, a System Parameter can type multiple Numbers, and multiple Numbers can be calculated by one Analysis Function. It is even possible, although probably a mistake in the analysis model, that in a Python analysis model, multiple System Parameters are calculated by one Analysis Function, or multiple Analysis Functions calculate Numbers typed by the same System Parameter.

The Knowledge Entity that does always represent the same thing, is a Number. Number leaves little room for interpretation in different formats: in the end, every analysis model has to calculate
numbers and take numbers as input; all these numbers are \textit{Numbers} in the domain model. Therefore, a \textit{Number} in one analysis model corresponds to exactly one \textit{Number} in the other analysis model, or to none at all.

Depending on the properties of the analysis models and the capturing methods, the same may apply for \textit{System Parameters}. However, it is also possible for a \textit{System Parameter} in one analysis model to correspond to multiple \textit{System Parameters} in another model. This is true for the Python - Excel case: one \textit{System Parameter} from the Python model can correspond to one or more \textit{System Parameters} in the Excel model, or to none at all. For \textit{Analysis Functions}, the same applies: one \textit{Analysis Function} in the Python model can correspond to one or more \textit{Analysis Functions} in the Excel analysis model, or to none at all.

A different correspondence exists for \textit{Components} and \textit{Component Instances}. These \textit{Knowledge Entities} represent a subdivision of an analysis model into logical parts. As there are no clear-cut rules that force an analysis model to always be divided into the same logical parts, a \textit{Component} in one model can map to one or more \textit{Components} in the other; but it can also map to a part of the other analysis model that belongs to two or more \textit{Components} in that model. The same holds for \textit{Component Instances}.

In order to reduce the room for errors made by the analyst, the way the domain models of the captured analysis models map to each other needs to be established first (e.g. by creating a table like table 7.1). This information can be used to inform the analyst of which mappings are possible, prior to her creating these mappings. It can also be used to constrain the input into a user interface in which these mappings can be defined.

<table>
<thead>
<tr>
<th>Knowledge Entity</th>
<th>Multiplicity (Python)</th>
<th>Multiplicity (Excel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Parameter</td>
<td>0..1</td>
<td>0..*</td>
</tr>
<tr>
<td>Analysis Function</td>
<td>0..1</td>
<td>0..*</td>
</tr>
<tr>
<td>Component</td>
<td>0..*</td>
<td>0..*</td>
</tr>
<tr>
<td>Component Instance</td>
<td>0..*</td>
<td>0..*</td>
</tr>
</tbody>
</table>

Table 7.1: Mapping of \textit{Knowledge Entities} between the data extracted by \textit{pyQAM} and the data extracted by the \textit{Excel Knowledge Extractor}

### 7.3 Mapping semantics

The next step is to map \textit{Numbers}, \textit{System Parameters}, \textit{Analysis Functions}, \textit{Components} and \textit{Component Instances} from one analysis model to their counterparts in the other analysis model. This is a matter of intuition: the analyst maps elements to each other that she thinks have the same meaning. However, it is hard to do this correctly, especially with \textit{System Parameters} and \textit{Analysis Functions}. A \textit{System Parameter} may \textit{seem} to represent the same thing as one or more \textit{System Parameters} in another model – for example, because the counterparts have similar names – but they can have entirely different roles; the same is true for \textit{Analysis Functions}. Therefore, it may not be wise to map \textit{System Parameters} and \textit{Analysis Functions} directly.

Instead, the analyst should only map \textit{Numbers}. The similarity of \textit{Numbers} is easier to determine; most of the time, they have similar values and similar names. When \textit{Numbers} have been mapped, \textit{System Parameters} and \textit{Analysis Functions} can be derived. How this can be done, is discussed below.

The mapping that was left out of the picture is that of \textit{Components} and \textit{Component Instances}. These entities define subdivisions of their respective analysis models. Because of that, they don’t really need to be mapped to each other; if two analysis models have been successfully mapped, they can co-exist as two different ways of dividing the mapped model into parts.

#### 7.3.1 Deriving mappings for \textit{System Parameters}

When \textit{Numbers} have been mapped, it is possible to derive with a certain degree of certainty which \textit{System Parameters} map to each other. Remember (section 3.3.1.1) that \textit{Numbers} have \textit{is of type}
relations to their respective System Parameters. When a Number $n_1$ from one analysis model has been mapped to a Number $n'_1$ in the other analysis model, $n_1$ having an \emph{is of type} relation to System Parameter $s_1$, and $n'_1$ to $s'_1$, it is reasonable to assume that $s_1$ and $s'_1$ have a certain degree of similarity. However, they do not have to be the same: the mapping of System Parameters between the analysis model can be one-to-many, in which case the System Parameter mapping based solely on $n_1$ and $n'_1$ is incomplete.

Take for example the two analysis models (A and B) in figure 7.1. Analysis model A is the PersonAndBike analysis model from chapter 5; it calculates the kinetic energy of a person and a bike traveling at a certain velocity. Analysis model B is a variation of the same model; but here, the calculations of kinetic energies of person and bike are done in separate Components (MassAndVelBike and MassAndVelPerson), in contrast to analysis model A, where these calculations are done in one Component (MassAndVelocity). If a Number that is typed by the System Parameter MassAndVelocity.mass in analysis model A is mapped to a Number that is typed by the System Parameter MassAndVelPerson.mass in analysis model B, the System Parameter mapping MassAndVelocity.mass $\mapsto$ MassAndVelPerson.mass based on this mapping between Numbers is obviously incomplete: MassAndVelocity.mass maps to both MassAndVelocity.mass and MassAndVelPerson.mass.

7.3.2 Deriving mappings for Analysis Functions

It is particularly difficult to decide whether Analysis Functions from both analysis models map to each other. Here, too, we have the problem that this mapping can be one-to-many: the example from the last section (figure 7.1) reveals that MassAndVelocity.calcPartKE in analysis model A maps to both MassAndVelPerson.calcPersonKE and MassAndVelBike.calcBikeKE in analysis model B.

Besides that, calculations in both analysis models may have different structures. If a Number $n_1$ of one analysis model is the result of a calculation with as input Numbers $n_2$ and $n_3$, and $n_1$, $n_2$ and $n_3$ map to Numbers $n'_1$, $n'_2$ and $n'_3$ in the other analysis model, it is entirely possible that in the other analysis model $n'_1$ and $n'_3$ are used to calculate $n'_2$. This is the case, for example, when one analysis model contains the calculation $n_1 = n_2 + n_3$ and another contains the calculation $n'_2 = n'_1 - n'_3$. Generalizing, one could say that Analysis Functions from different analysis models can be each other’s inverse, or at least partially.
7.4 Visualization

When a mapping has been defined, the two analysis models can be visualized. Two categories of visualization can be distinguished: visualizing two analysis models side by side, and visualizing two analysis models in the same figure (the so-called "diff" view).

7.4.1 Side by side view

This kind of view shows the two analysis models next to each other; this view is much like visualizing the two analysis models separately and then putting them next to each other. When a mapping is known between two analysis models, this information can be used to give the visualizations similar lay-outs: elements that have been indicated by the analyst to map to each other, are given the same place in the visualizations of their analysis models.

7.4.1.1 Side by side dependency graphs

In figure 7.2, the Analysis Functions and System Parameters that map to each other are laid out in dependency graphs in such a way that the two dependency graphs have a similar shape, and that it is easy to point out which parts are present in one of the analysis models and not in the other.

![Figure 7.2: Graph side-by-side view](image)

7.4.1.2 Side by side system parameter lists

The ordering on the system parameter list in figure 7.3 is such that the System Parameters that are in both analysis models have the same order in both visualizations.

7.4.2 Combined view ("diff")

The combined view takes the side-by-side view one step further: the two visualizations are combined into one. Examples of how this can be done are in figures 7.4 and 7.5 for the dependency graph visualization, and in 7.6 for the System Parameter list visualization.
CHAPTER 7. COMPARING ANALYSIS MODELS

7.4. VISUALIZATION

7.4.2.1 Combined Dependency Graph

Essential for the dependency graph visualizations is that common parts and parts that are unique to either of the analysis model are color-coded. The color coding is such that it is easy to see which parts are common, and which parts are only in either of the analysis models but not in the other.

This color-coding can be taken a step further: common System Parameters can be colored with a more or less intensive color, indicating how similar these System Parameters are in both models. This similarity can be derived from the similarity of the associated Numbers and how well the relations with the rest of the model coincide for the counterparts of the System Parameter. In figure 7.4, this is exemplified by common System Parameters that are a darker or lighter shade of grey.

The difference between figure 7.4 and 7.5 is that the former shows Analysis Functions in both analysis models that map to each other, and the latter shows them in a merged fashion. Related work in the field of software reverse engineering proposes more "diff" views for graphs; see section 9.4.
7.4.2.2 Combined System Parameter List

*System Parameters* in the list visualizations of both analysis models can be lined up next to each other so that mapped *System Parameters* are next to each other (figure 7.6). In this visualization, it is easy to spot which parts are common for both analysis models, and which parts belong to only one analysis model. For example, in the listing in figure 7.6, if a *System Parameter* in the blue analysis model on the left has a red counterpart, it is displayed on the same line; if not, that line is empty on the red side.

```
<table>
<thead>
<tr>
<th>sp1-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp1-3</td>
</tr>
<tr>
<td>sp1-4</td>
</tr>
<tr>
<td>sp1-7</td>
</tr>
<tr>
<td>sp1-8</td>
</tr>
<tr>
<td>sp1-9</td>
</tr>
<tr>
<td>sp1-2</td>
</tr>
<tr>
<td>sp1-4</td>
</tr>
<tr>
<td>sp1-7</td>
</tr>
<tr>
<td>sp1-8</td>
</tr>
<tr>
<td>sp1-9</td>
</tr>
<tr>
<td>sp1-5</td>
</tr>
<tr>
<td>sp1-10</td>
</tr>
<tr>
<td>sp1-11</td>
</tr>
<tr>
<td>sp1-6</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Figure 7.6: List: combined view
Chapter 8

Validation

8.1 Introduction

Chapter 4 presented the requirements, use cases and design of a software solution, pyQAM, that helps an analyst to understand an analysis model in Python. PyQAM provides a way of capturing data from a Python analysis model as described in chapter 5. This captured data can be used to visualize the analysis model (chapter 6), and it can be compared to captured data from another analysis model; the comparison of this data and its visualization are discussed in chapter 7.

To summarize, these chapters give an answer to the last research question of chapter 2:

*How can we make finding differences between different analyses of the same system easier, less error-prone and quicker?*

This chapter validates how good an answer pyQAM is to this question by testing:

- How well pyQAM implements the use cases and requirements formulated in chapter 4.
- How well pyQAM performs in practice, i.e., whether it makes finding differences easier, less error-prone and faster. A controlled experiment has been conducted to validate this.

8.2 Use cases and requirements

Tables 8.1 and 8.2 list the functional and non-functional requirements from chapter 4, along with an indication of the status of their implementation in pyQAM. A similar table for the use cases from the same chapter are in table 8.3.

Requirements R01 through R05 follow from use case UC1.3. All of these but the last have been marked as complete in the pyQAM solution; therefore, UC1.3 has also been marked as Complete, as the lack of R05 is only a minor drawback.

Use case UC1.2 stands at the base of requirements R07 through R10; these requirements have to do with the annotation capabilities of pyQAM. Only the annotation capabilities needed in order to implement R11 were implemented, along with annotations needed for the storage of multiple analysis models in one repository. Therefore, UC1.2 is marked as Partially implemented.

Requirements having to do with the creation or modification of analysis models belong to use case UC1.1; they are R06, R11 and R12. R06 is trivial: the analysis model’s Components can serve as abstractions, making R06 a part of the analysis model’s design rather than a feature of pyQAM. R11 and R12 are implemented in the pyQAM user interface (pyQAMui). UC1.1 has been marked as Complete.

The last of the use cases, UC2.1, is about comparing analysis models. Ideas for how this might be done have been proposed in chapter 7, but not (yet) implemented in pyQAM and therefore not in the Knowledge Architect. However, comparing analysis models to each other by making use of the visualizations of chapter 6 certainly helps when comparing two analysis models to each other (see section 8.3). Therefore, UC2.1 has been marked as Partially implemented.
# 8.2. Use Cases and Requirements

## 8. Validation

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Dep.</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>Represent the analysis model in a visually attractive manner</td>
<td>R11</td>
<td>Complete</td>
</tr>
<tr>
<td>R02</td>
<td>Enable the user to fold/unfold abstractions in the visualization</td>
<td></td>
<td>Complete</td>
</tr>
<tr>
<td>R03</td>
<td>Clearly indicate descendants and precedents of a selected system parameter</td>
<td>R01</td>
<td>Complete</td>
</tr>
<tr>
<td>R04</td>
<td>Provide a way to perform parameter analysis by hand</td>
<td></td>
<td>Complete</td>
</tr>
<tr>
<td>R05</td>
<td>Deduce semantically enhanced formulas from the model’s formulas</td>
<td>R08, R09</td>
<td>Not implemented</td>
</tr>
<tr>
<td>R06</td>
<td>Provide a way to group formulas in abstractions</td>
<td></td>
<td>Complete</td>
</tr>
<tr>
<td>R07</td>
<td>Provide a way to store the point of origin of an input: who defined it and why?</td>
<td></td>
<td>Partially implemented</td>
</tr>
<tr>
<td>R08</td>
<td>Provide a way to annotate formulas with unit information</td>
<td></td>
<td>Not implemented</td>
</tr>
<tr>
<td>R09</td>
<td>Provide a way to annotate formulas with semantic information</td>
<td></td>
<td>Complete</td>
</tr>
<tr>
<td>R10</td>
<td>Provide a way to annotate abstractions with rationale</td>
<td>R06</td>
<td>Not implemented</td>
</tr>
<tr>
<td>R11</td>
<td>Extract formulas from a model</td>
<td></td>
<td>Complete</td>
</tr>
<tr>
<td>R12</td>
<td>Build a dependency graph from formulas</td>
<td>R10</td>
<td>Complete</td>
</tr>
<tr>
<td>R13</td>
<td>Visualize the differences between two analysis models</td>
<td>R11</td>
<td>Partially implemented</td>
</tr>
</tbody>
</table>

Table 8.1: Functional requirements

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirement</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF01</td>
<td>Lightweight</td>
<td>Complete</td>
</tr>
<tr>
<td>NF02</td>
<td>Easily deployable and maintainable</td>
<td>Complete</td>
</tr>
<tr>
<td>NF03</td>
<td>Easy to use and understand</td>
<td>Complete</td>
</tr>
<tr>
<td>NF05</td>
<td>Robust (not crash)</td>
<td>Complete</td>
</tr>
<tr>
<td>NF07</td>
<td>Extensible</td>
<td>Complete</td>
</tr>
<tr>
<td>NF08</td>
<td>Stand-alone</td>
<td>Complete</td>
</tr>
<tr>
<td>NF09</td>
<td>Part of Knowledge Architect</td>
<td>Complete</td>
</tr>
</tbody>
</table>

Table 8.2: Non-functional requirements

<table>
<thead>
<tr>
<th>ID</th>
<th>Use Case</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC1.1</td>
<td>Create / modify analysis model</td>
<td>Complete</td>
</tr>
<tr>
<td>UC1.2</td>
<td>Annotate analysis model</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>UC1.3</td>
<td>Analyze analysis model</td>
<td>Complete</td>
</tr>
<tr>
<td>UC2.1</td>
<td>Compare analysis models</td>
<td>Partially implemented</td>
</tr>
</tbody>
</table>

Table 8.3: Use cases
8.3 Experiment

The second part of the validation of the pyQAM solution presented in this thesis, is testing whether it makes finding differences easier, less error-prone, and faster. For this purpose, an experiment has been conducted in which three ASTRON engineers and five GRIFFIN researchers participated. The experiment consists of a number of assignments that consist of questions about one analysis model and its differences to another analysis model; each participant first answers a set of questions without, and then a set of questions with the help of pyQAM visualizations.

Using pyQAM in practice involves invoking it with a Python analysis model and a set of input parameters (the scenario being analyzed), which produces a number of visualizations. In turn, these visualizations are used to navigate through the extracted data. The focus of this experiment is on whether the visualizations help the analyst to understand an analysis model and find differences between analysis models. However, the invocation part of the pyQAM routine is not entirely trivial, and may have an influence on how well someone can work with pyQAM. Configuring and invoking pyQAM is a skill that can be mastered with some practice, and is an entirely separate matter from actually understanding an analysis model and finding differences between analysis models. Therefore, the participants of this experiment didn’t have to actually invoke pyQAM to produce the visualizations they used to answer some of the questions; the visualizations were ready-made.

A prerequisite for finding differences between analysis models is understanding the individual analysis models: when an analyst understands two models, she will be more capable of finding differences between them. Therefore, the experiment is set up to find an answer to the following research questions:

1. On understanding analysis models:
   (a) Does using pyQAM visualizations enable the user to understand an analysis model quicker than without using it?
   (b) Does using pyQAM visualizations enable the user to understand an analysis model better than without using it?
   (c) Does using pyQAM visualizations make understanding an analysis model easier?
2. On finding differences between analysis models:
   (a) Does using pyQAM visualizations enable the user to find differences between analysis models quicker than without using it?
   (b) Does using pyQAM visualizations enable the user to find differences between analysis models better than without using it?
   (c) Does using pyQAM visualizations make finding differences between analysis models easier?

An answer to these questions is an indicator for how fit the pyQAM visualizations are for their purpose.

8.3.1 Experiment Setup

8.3.1.1 Participants

Because of a lack of resources and time, only eight subjects participated in the experiment. Three of them are ASTRON employees, and of them two regularly work with analysis models similar to SKAcost; however, SKAcost is pretty much unexplored territory to all of them. The five other participants are from the Griffin group at the University of Groningen, and three of them have experience with the domain model used to represent SKAcost in the visualizations of pyQAM.

Four of the eight subjects answered question set 1, the four others answered question set 2. Questions in one set that were answered without use of visualizations were answered in the other set with visualizations available, in order to test for average performance differences on each of the questions with and without visualizations. For an overview of participants and the question sets they answered, and whether they have experience in fields that might influence the experiment’s results, see table 8.4 on the following page.
### Table 8.4: Participants to the experiment

<table>
<thead>
<tr>
<th>Subject</th>
<th>Question Set</th>
<th>Occupation</th>
<th>Eclipses</th>
<th>Python</th>
<th>SKAcost</th>
<th>in General</th>
<th>Analyses Models</th>
<th>Domain Model</th>
<th>Subject Question Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Griffin Researcher</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Griffin Researcher</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Griffin Researcher</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Griffin Researcher</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Griffin Researcher</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Griffin Researcher</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Griffin Researcher</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>ASTRON Engineer</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ASTRON Engineer</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>ASTRON Engineer</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ASTRON Engineer</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>ASTRON Engineer</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Griffin Researcher</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Griffin Researcher</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Griffin Researcher</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Griffin Researcher</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

8.3. EXPERIMENT
8.3.1.2 Questions

![Figure 8.1: Overview of the question sets used in the experiment and how they correspond to each other. Part A in question set 1 contains the exact same question as part A in question set 2; the same is true for the B parts of both question sets. Part A is performed without tool in question set 1 and with tool in question set 2. Par B is performed without tool in question set 2 and with tool in question set 1.](image)

The two sets of questions concern the properties of one analysis model, and its differences to another analysis model. The first analysis model is the SKAcost analysis model, calculated for the scenario called "SKADS". The tooling was not yet capable of visualizing Excel and Python models in the same way; therefore the "other" analysis model is also the SKAcost model, but calculated in the scenario called "WebDefault".

The question sets each consist of two parts; referred to as parts A and B. Question set 1 starts with part A (to be answered without visualizations), and then continues with part B (to be answered with visualizations available). Question set 2 has the parts in a different order; questions that are answered with visualizations in question set 1 are answered without visualizations in question set 2 and vice versa (see figure 8.1).

Furthermore, parts A and B are as similar as possible. They consist of the same questions, but about different parts of the analysis model(s). The questions are:

1. Property Discovery
   1.1. Find at most five system parameters that represent the most important outcomes for component [Component]. Give them in order of importance, most important first.
   1.2. Find at most five system parameters that represent important assumptions for component [Component]. Give them in order of importance, most important first.

2. Dependency Discovery
   2.1. Find all assumptions that influence system parameter [SystemParameter]
   2.2. Find all results that are influenced by system parameter [SystemParameter]

3. Improvement Analysis
   3.1. Try to find errors, inconsistencies or possible improvements in component [Component]
4. Difference Analysis

4.1. Find the system parameter that is present in both WebDefault and SKADS scenarios, on which the most important difference is in component [Component].

4.2. What dependant system parameters of the found system parameters are the most important causes for this difference? Also give the values that are associated with these system parameters.

The placeholders [Component] and [SystemParameter] show where the differences between parts A and B are. Table 8.5 shows the values for [Component] and [SystemParameter] used in each question set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Q. Set</th>
<th>With / without tool</th>
<th>[Component]</th>
<th>[SystemParameter]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>without</td>
<td>AAProcessingFirst</td>
<td>AAProcessingFirst.cost2DFFT2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>with</td>
<td>AAProcessingStation</td>
<td>AAProcessingStation.dataRatePerDualPolStationBeam</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>without</td>
<td>AAProcessingStation</td>
<td>AAProcessingStation.dataRatePerDualPolStationBeam</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>with</td>
<td>AAProcessingFirst</td>
<td>AAProcessingFirst.cost2DFFT2</td>
</tr>
</tbody>
</table>

Table 8.5: Values for [Component] and [SystemParameter] in each of the question sets

The answers that are given by participants to the experiment questions above should provide an answer to the research questions stated at the beginning of the Experiment section (that is, section 8.3). The questions on property discovery (1.*), dependency discovery (2.*) and improvement analysis (3.*) were devised to get an answer to research questions 1(a), 1(b) and 1(c); the questions on difference analysis (4.*) to get an answer for research questions 2(a), 2(b) and 2(c).

8.3.1.3 Hypotheses

The results of the experiment should be tested formally, as far as the nature of the results allows this to be done. This section formulates null hypotheses and corresponding alternative hypotheses. For the test results, see section 8.3.2 on page 85.

Research questions 1(c) and 2(c) are of a qualitative nature; that is, they can only be answered by asking people whether it was easier to do these tasks with pyQAM visualizations than without them (i.e., "easiness" cannot be directly measured). Therefore, no null and alternative hypotheses are formulated for answers to these; instead, they are answered directly by the participants.

Hypotheses about the answers to questions 1(a) and 2(a) are treated in Timeliness below; hypotheses on answers to 1(b) and 2(b) are treated under Level of understanding.

Timeliness Experiment research questions 1(a) and 2(a) are about the timeliness of the answers given to questions in the experiment. The time that a participant needs to answer each question is recorded; the goal is to test whether the average time needed with tool is smaller than the average time needed without tool. Thus, the null hypotheses is:

- \( H_{A0} \): average time with visualizations = average time without visualizations

The corresponding alternative hypotheses then are:

- \( H_{A1} \): average time with visualizations < average time without visualizations
- \( H_{A2} \): average time with visualizations > average time without visualizations

Because participants didn’t get a time limit to answer question 3.1 (about improvements to the analysis model at hand), and this question leaves much room for interpretation by participants, this question is left out of consideration when testing the \( H_{A0} \) hypothesis. For an overview of which answers are taken into account when testing hypotheses, see table 8.6 on the facing page.
CHAPTER 8. VALIDATION 8.3. EXPERIMENT

Quality of answers Whether a participant is able to do a better job with visualizations than without them, is a question that can be inverted into whether participants make less errors with visualizations than without them:

- \( H_{b0} \): number of errors with visualizations = number of errors without visualizations

The alternative hypotheses then are:

- \( H_{b1} \): number of errors with visualizations < number of errors without visualizations
- \( H_{b2} \): number of errors with visualizations > number of errors without visualizations

Looking at the type of questions prepared for the experiment, testing these alternative hypotheses about the number of errors cannot be done directly on some of them. Answers to the experiment questions on property discovery (1.*) and difference analysis (4.*) do not necessarily have a correct answer; the answer to these questions depends on the participant’s domain of expertise, at what stage of development the analysis model is, perhaps even the participant’s personal taste (cf. section 3.3.4 on significance of values in analysis models). Therefore, testing of hypothesis \( H_{b0} \) and it’s alternatives cannot take into account answers to questions on property discovery (1.*); and on difference analysis (4.*).

Interpreting answers to question 3.1 (about improvements to the analysis model) is even more evasive than is the case with the property discovery and difference analysis questions: a participant can come up with anything. Answers to this question will be interpreted qualitatively. For an overview of which answers are taken into account when testing hypotheses, see table 8.6.

<table>
<thead>
<tr>
<th>Question</th>
<th>( H_{a0} )</th>
<th>( H_{b0} )</th>
<th>( H_{b1}' )</th>
<th>( H_{b2}' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.6: Answers to questions that are taken into account when testing hypotheses

Reformulating null hypothesis \( H_{b0} \) into \( H_{b1}' \) and \( H_{b2}' \) Even though testing alternative hypotheses to \( H_{b0} \) cannot take into account answers to experiment questions 1.1, 1.2, 4.1 and 4.2, it is still possible to formulate similar hypotheses that can be tested.

As can be seen in table 8.4 on page 80, subject 3 is an ASTRON engineer with experience in the field of analysis models and the SKAcost model. One possibility is to consider this subject’s answers as the "truth", so that answers that differ from this subject’s answers are erroneous. The first null hypotheses to replace \( H_{b0} \), \( H_{1b0} \) and \( H_{2b0} \) are:

- \( H_{b1}' \): number of errors w.r.t. subject 3’s answers with visualizations = number of errors w.r.t. subject 3’s answers without visualizations

Another possibility is to consider no one’s answers to necessarily be true. When no solid truth can be used to determine whether answers are correct or not, the only thing that remains is to determine how much all of the participants agree on the answer to a question:

- \( H_{b2}' \): agreement among all subjects with visualizations = agreement among all subjects without visualizations

Where agreement stands for the Fleiss’ kappa coefficient, a measure of inter-rater agreement for categorical items such as the answers to the experiment questions.

The corresponding alternative hypotheses are:
8.3. EXPERIMENT

\[ H_{A1}^1: \text{number of errors w.r.t. subject 3's answers with visualizations} < \text{number of errors w.r.t. subject 3's answers without visualizations} \]

\[ H_{A2}^1: \text{number of errors w.r.t. subject 3's answers with visualizations} > \text{number of errors w.r.t. subject 3's answers without visualizations} \]

\[ H_{A1}^2: \text{agreement among all subjects with visualizations} > \text{agreement among all subjects without visualizations} \]

\[ H_{A2}^2: \text{agreement among all subjects with visualizations} < \text{agreement among all subjects without visualizations} \]

8.3.1.4 Conducting the experiment

Prior to the experiment, the participant is interviewed to determine knowledge and experience that influence the speed and accuracy of answering the questions. After that, a training session is held, consisting of answering a set of practice questions that are the same as the actual experiment questions, but about a different part of the analysis model. The training session lasts until the participant understands each of the questions in the practice set and knows how to find an answer to them.

Subsequently, a participant starts on the first part of the question set (without help from pyQAM), recording:

- The answer(s) given
- The time needed to come up with the answer
- Remarks the participant may have about the question that influence the timeliness and accuracy of the answer.

With the second part of the question set, it is allowed to use pyQAM visualizations to find answers to questions. The participant records:

- The method(s) used to find the answer. This is a list of:
  - M: Manually
  - G: Graph of System Parameters and Analysis Functions
  - S: System Parameter List
  - A: Assumption List
  - N: Number Table

After the questions in the question set have been answered, the participant answers some feedback questions about the experiment and the pyQAM tool.

8.3.1.5 Threats to validity

There are a number of circumstances that may influence the validity of the results. Below is a short summary of the biggest threats to validity, and in some cases what has been done to minimize these threats.

Number of Participants Eight participants took part in the experiment, of which three ASTRON engineers and five GRIFFIN researchers. This is not a very large number, and it influences how accurate the results are.
Learning effect  The participants are tested twice, once with and once without tool, on how quickly and how accurately they can harvest information from an analysis model. One threat to validity of the results is that they may learn a great deal about the analysis model in the first half of the experiment, which enables them to answer questions faster and more accurately in the second half.

To prevent this from happening, the analysis model to be analyzed has a degree of complexity that cannot be mastered in less than one day. Furthermore, questions of the first half of the experiment are on a different part of the analysis model than those of the second half.

Not enough training time  Participants get only about an hour’s time to learn how to use the visualizations to solve their problems. Therefore, the experiment can only measure whether the visualizations help instantly, and not whether they help once an analyst has grown more accustomed to them after using them for a few weeks.

Order of testing  In all cases, the participants answer the first half of a question set without, and the second half with visualizations. Answering one half of the question set takes about 90 minutes; a participant might be a bit discouraged to start the second half, which poses the threat that answers to the second half of the experiment are conceived with less effort than answers in the first half of the experiment. This may translate in less time spent per question, or answers that are not well thought through.

8.3.2 Quantitative results

8.3.2.1 Timeliness

<table>
<thead>
<tr>
<th>Question Set</th>
<th>Subject</th>
<th>1. Without Tool</th>
<th>2. With Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1.1 1.2 2.1 2.2</td>
<td>4.1 4.2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>530 361 565 605</td>
<td>241 132</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>198 374 324 250</td>
<td>296</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>600 226 257 231</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>356 252 442 264</td>
<td>776 600</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>660 210 426 443</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>355 342 59  625</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>370 546 225 212</td>
<td>572 220</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>590 600 182 174</td>
<td>600</td>
</tr>
</tbody>
</table>

Figure 8.2: Times with and without tool

The time in seconds needed by each participant for each question is lined out in table 8.2. A dash ("-") indicates that the participant didn’t give any answer to a question. Hypothesis $H_{a0}$ can be tested by applying Student’s $t$ test on this data. We accept a hypothesis if it this test confirms it with uncertainty $\alpha < 0.05$.

In fact, the nature of our data is such that this test can be applied in two ways. Firstly, we can see the two sets of times, the set of times for answers without tool (the "without" set) and the set of times for answers with tool (the "with" set), as two independent samples. These samples have unequal variance and unequal sample size (due to different numbers of unanswered questions).

It is also possible, because each of the question sets consists of two parts that are exactly the same but are answered in different orders, to calculate average times for answers with visualizations and without visualizations per question. The question can be seen as a patient undergoing two treatments: the treatment without visualizations and the treatment with visualizations. The average times per question are displayed in figure 8.3a on the following page. These average times can be used in Student’s $t$ test for paired samples.

Results of Student’s $t$ test with independent samples  Figure 8.4 on the next page shows a table with three rows; the first contains the results for testing $H_{a0}$ when taking into account all answers (1.1, 1.2, 2.1, 2.2, 4.1, 4.2); the second contains results for testing $H_{a0}$ on answers...
8.3. EXPERIMENT

CHAPTER 8. VALIDATION

<table>
<thead>
<tr>
<th>question</th>
<th>average time with tool</th>
<th>average time without tool</th>
<th>difference (with – without)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1.1</td>
<td>471</td>
<td>421</td>
<td>50</td>
<td>10.62%</td>
</tr>
<tr>
<td>A-1.2</td>
<td>310</td>
<td>303</td>
<td>7</td>
<td>2.26%</td>
</tr>
<tr>
<td>A-2.1</td>
<td>261</td>
<td>397</td>
<td>-136</td>
<td>-52.11%</td>
</tr>
<tr>
<td>A-2.2</td>
<td>291</td>
<td>338</td>
<td>-47</td>
<td>-16.15%</td>
</tr>
<tr>
<td>A-4.1</td>
<td>357</td>
<td>509</td>
<td>-152</td>
<td>-42.58%</td>
</tr>
<tr>
<td>A-4.2</td>
<td>182</td>
<td>315</td>
<td>-133</td>
<td>-73.08%</td>
</tr>
<tr>
<td>B-1.1</td>
<td>343</td>
<td>540</td>
<td>-197</td>
<td>-57.43%</td>
</tr>
<tr>
<td>B-1.2</td>
<td>351</td>
<td>428</td>
<td>-77</td>
<td>-21.94%</td>
</tr>
<tr>
<td>B-2.1</td>
<td>370</td>
<td>294</td>
<td>76</td>
<td>20.54%</td>
</tr>
<tr>
<td>B-2.2</td>
<td>202</td>
<td>222</td>
<td>-20</td>
<td>-9.90%</td>
</tr>
<tr>
<td>B-4.1</td>
<td>340</td>
<td>599</td>
<td>-259</td>
<td>-76.18%</td>
</tr>
<tr>
<td>B-4.2</td>
<td>247</td>
<td>201</td>
<td>46</td>
<td>18.62%</td>
</tr>
</tbody>
</table>

(a) Average times with and without tool, paired by question

Figure 8.3: Average time difference, paired by question

<table>
<thead>
<tr>
<th>question</th>
<th>without tool</th>
<th>with tool</th>
<th>degrees of freedom</th>
<th>T</th>
<th>p-value (one-tailed)</th>
<th>α</th>
<th>certainty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>42 373.83</td>
<td>46 311.37</td>
<td>82.68</td>
<td>-1.7255</td>
<td>1.6636</td>
<td>0.050</td>
<td>0.9559</td>
</tr>
<tr>
<td>On understanding (1.1 – 2.2)</td>
<td>31 362.19</td>
<td>30 327.37</td>
<td>58.92</td>
<td>-0.8486</td>
<td>1.6716</td>
<td>0.050</td>
<td>0.8002</td>
</tr>
<tr>
<td>On Comparing (4.1, 4.2)</td>
<td>11 406.64</td>
<td>16 281.38</td>
<td>16.58</td>
<td>-1.5795</td>
<td>1.7459</td>
<td>0.050</td>
<td>0.9331</td>
</tr>
</tbody>
</table>

Figure 8.4: Results for Student’s t test independent samples. Timings with and without tool are treated as independent samples with unequal variance and unequal size.
to questions on understanding the analysis model (1.1, 1.2, 2.1, 2.2); the third is on answers to questions on finding differences between, or comparing, analysis models (4.1, 4.2).

Taking all time measurements without tool as the first series of measurements and all time measurements with tool as the second series yields a test statistic $T = -1.7255$ with a $t_{83}$ distribution. For $\alpha = 0.05$, the one-sided p-value for $t_{83}$ is 1.6636. Testing $H_{a0}$ against $H_{aA1}$, $H_{a0}$ is rejected in favor of $H_{aA1}$ because $T = -1.7255 < -1.6636$. Therefore, taking into account all timings, we can say with a sufficient level of confidence that the visualizations do help participants to work more quickly.

However, the two other test runs are not as conclusive as the first one. Taking into account only the timings on questions on understanding the analysis model (1.1 to 2.2), we get a test statistic $T = -0.8486$, with a $t_{59}$ distribution. For $\alpha = 0.05$, the one-sided p-value for $t_{59}$ is 1.6716. Therefore, neither $H_{aA1}$ nor $H_{aA2}$ can be accepted, because $-1.6716 < T < 1.6716$. Thus, if only timings on questions on understanding the analysis model are taken into account, whether the visualizations help participants to find answers more rapidly cannot be answered with enough certainty. This is most likely due to the small sample size.

Taking into account only the timings on questions on comparing analysis models (4.1 and 4.2), we get a test statistic $T = -1.5795$, with a $t_{17}$ distribution. For $\alpha = 0.05$, the one-sided p-value for $t_{17}$ is 1.7459. Therefore, neither $H_{aA1}$ nor $H_{aA2}$ can be accepted, because $-1.7459 < T < 1.7459$. Thus, if only timings on questions on comparing analysis models are taken into account, whether the visualizations help participants to find answers more rapidly cannot be answered with enough certainty; again, the small sample size is likely the cause.

<table>
<thead>
<tr>
<th></th>
<th>mean improvement (seconds)</th>
<th>T</th>
<th>p-value (one-tailed)</th>
<th>$\alpha$</th>
<th>certainty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>70.17</td>
<td>-2.2794</td>
<td>1.7959</td>
<td>0.0500</td>
<td>0.9782</td>
</tr>
<tr>
<td>On understanding (1.1 – 2.2)</td>
<td>43.00</td>
<td>-1.3206</td>
<td>1.8946</td>
<td>0.0500</td>
<td>0.8959</td>
</tr>
<tr>
<td>On Comparing (4.1, 4.2)</td>
<td>124.50</td>
<td>-1.9687</td>
<td>2.3534</td>
<td>0.0500</td>
<td>0.9282</td>
</tr>
</tbody>
</table>

Figure 8.5: Results for Student’s t test on paired samples. Each pair consists of timings with and without tool averaged for each question.

**Results of Student’s t test with paired samples** As with the independent samples, figure 8.5 has three rows; one for overall results on $H_{a0}$, one for results concerning understanding the analysis model and one for results concerning comparing analysis models.

When looking at measurements for the whole sample, we get a test statistic $T = -2.2794$ with a $t_{59}$ distribution. For $\alpha = 0.05$, the one-sided p-value for $t_{59}$ is 1.7959. Testing $H_{a0}$ against $H_{aA1}$, $H_{a0}$ is rejected in favor of $H_{aA1}$ because $T = -2.2794 < -1.6716$. Therefore, taking into account all pairs, the visualizations help participants to work more quickly.

Again, the two other measurements are not as conclusive as the first one. Taking into account only the timings on questions on understanding the analysis model (1.1 to 2.2), we get a test statistic $T = -1.3206$, with a $t_{17}$ distribution. For $\alpha = 0.05$, the one-sided p-value for $t_{17}$ is 1.8946. Therefore, neither $H_{aA1}$ nor $H_{aA2}$ can be accepted, because $-1.8946 < T < 1.8946$. Thus, if only timings on questions on understanding the analysis model are taken into account, whether the visualizations help participants to find answers more rapidly cannot be answered with enough certainty, likely due to the small sample size.

Taking into account only the timings on questions on comparing analysis models (4.1 and 4.2), we get a test statistic $T = -1.9687$, with a $t_{3}$ distribution. For $\alpha = 0.05$, the one-sided p-value for $t_{3}$ is 2.3534. Therefore, neither $H_{aA1}$ nor $H_{aA2}$ can be accepted, because $-2.3534 < T < 2.3534$. Thus, if only timings on questions on comparing analysis models are taken into account, whether the visualizations help participants to find answers more rapidly cannot be answered with enough certainty, likely due to the small sample size.

8.3.2.2 Quality of answers
### 8.3. EXPERIMENT

#### CHAPTER 8. VALIDATION

**Figure 8.6: Number of errors in answers to questions 2.1 and 2.2**

<table>
<thead>
<tr>
<th>Question</th>
<th>Errors Without Tool</th>
<th>Errors With Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-2.1</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>A-2.2</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>B-2.1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>B-2.2</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>53</strong></td>
<td><strong>50</strong></td>
</tr>
</tbody>
</table>

Improvement with tool 5.66%

![Bar chart showing number of errors for questions A-2.1, A-2.2, B-2.1, B-2.2 for both conditions, with and without tool, illustrating the improvement with tool use.](image-url)
Correctness of answers  The answers to questions 2.1 and 2.2 were tested for correctness. The answers to these questions consist of lists of System Parameters; each occurrence of a System Parameter’s name in a participant’s answer to 2.1 or 2.2 that does not occur in the correct answer, is counted as an error. Furthermore, if the participant’s answer is a shorter list than the correct answer is, the difference in length is counted as an additional number of errors.

The results are depicted in figure 8.6 on the preceding page. Overall, the answers given with help of the pyQAM tool’s visualizations contained 5.66% less errors than the answers given without them. How significant this difference is, cannot be tested through Student’s t test as was done with the timings, because the Shapiro-Wilk normality test that was performed on the data in figure 8.6 gave a negative result.

<table>
<thead>
<tr>
<th>Question</th>
<th>Errors Without Tool</th>
<th>Errors With Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1.1</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>A-1.2</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>A-4.1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A-4.2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>B-1.1</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>B-1.2</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>B-4.1</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>B-4.2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Totals</td>
<td>74</td>
<td>55</td>
</tr>
<tr>
<td>Improvement with tool</td>
<td>25,68%</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.7: Number of "errors" in answers to questions 1.1, 1.2, 4.1, 4.2. The "errors" are actually differences to subject 3’s answers.

Agreement with subject 3  On questions 1.1, 1.2, 4.1 and 4.2, it is not possible to define "correct" answers because they may well be non-existent. Therefore, the level of agreement with
one of the subjects is taken as a measure of the quality of the participants’ answers; that is, the answers of one of the subjects are defined to be "correct", and the other subjects’ answers are tested against these. Subject 3 is one of the ASTRON engineers, with experience in both the analysis model’s knowledge domain, authoring analysis models and reviewing them; subject 3’s answers are therefore taken to be "correct".

The results are depicted in figure 8.7 on the preceding page. Overall, the answers given with help of the pyQAM tool’s visualizations contained 25.68% less "errors" than the answers given without them. How significant this difference is, cannot be tested through Student’s t test as was done with the timings, because again, the Shapiro-Wilk normality test that was performed on the data in figure 8.7 gave a negative result.

<table>
<thead>
<tr>
<th>Question</th>
<th>Without Tool</th>
<th>With Tool</th>
<th>More / Less Agreement With Tool?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\kappa$</td>
<td>agreement</td>
<td>$\kappa$</td>
</tr>
<tr>
<td>A-1.1</td>
<td>0.28</td>
<td>Fair</td>
<td>0.21</td>
</tr>
<tr>
<td>A-2.2</td>
<td>0.34</td>
<td>Fair</td>
<td>0.21</td>
</tr>
<tr>
<td>A-4.1</td>
<td>0.01</td>
<td>Poor</td>
<td>-0.01</td>
</tr>
<tr>
<td>A-4.2</td>
<td>0.12</td>
<td>Slight</td>
<td>0.13</td>
</tr>
<tr>
<td>B-1.1</td>
<td>0.10</td>
<td>Slight</td>
<td>0.60</td>
</tr>
<tr>
<td>B-1.2</td>
<td>0.06</td>
<td>Slight</td>
<td>0.13</td>
</tr>
<tr>
<td>B-4.1</td>
<td>0.17</td>
<td>Slight</td>
<td>0.16</td>
</tr>
<tr>
<td>B-4.2</td>
<td>-0.01</td>
<td>Poor</td>
<td>-0.02</td>
</tr>
</tbody>
</table>

Figure 8.8: Fleiss’ kappa evaluated for answers to questions 1.1, 1.2, 4.1, 4.2

Agreement among participants  How much participants agree with each other is also a measure of the quality of the answers they give. If all participants give different answers, the quality of their answers is likely to be poor; if they all give the same answers, the quality is likely to be better.

Measuring how much participants agree with each other is done through Fleiss’ kappa, a statistical measure for the reliability of agreement between a number of raters that assign categorical ratings to a number of subjects. In this case, the raters are the participants to the experiment. The subjects are formed by the System Parameters that are potentially part of the answer to a question (because questions 1.1, 1.2, 4.1 and 4.2 all require lists of System Parameters as answers). The categories a rater can put a subject into are simply yes and no; a rater puts a subject (System Parameter) in the yes category for a question if that System Parameter is part of her answer; she (implicitly) puts it in the no category otherwise.

The results of this exercise are in figure 8.8. Two lists of questions with their corresponding $\kappa$ values and agreement ratings are displayed in this table; the left list is of questions answered without tool and the right list is of questions answered with tool. The lists are lined up to each other: two question results on the same row of the table represent results for the same question, the left with tool and the right without tool. If the difference between the two $\kappa$ values is greater than 0.20, we consider this a significant difference; whether this is the case, and whether there is more or less agreement when the tool is used, is shown in the last column of the table. As can be seen in figure 8.8, on most of the questions, there is not more agreement among participants that were allowed to use the visualizations than among participants that were not allowed to use them. Overall, the difference in agreement between participants not using the visualizations and the participants that do use them, is not significant.

8.3.2.3 Methods used

The participants were asked to record which methods they used to find an answer to the experiment questions. The methods available were:

- M: Manually
- G: Graph of System Parameters and Analysis Functions
• S: System Parameter List
• A: Assumption List
• N: Number Table

Figure 8.9 contains a table that shows for each question how many participants used every method. Below the table is a chart, showing how many times each of the methods was used in total.

Fortunately, the "Manual" method has the lowest score: apparently, the visualizations have enough appeal to the analyst to be chosen over the "good old" approach. The figure also shows that the G(raph) ans S(system Parameter list) methods are the most popular ones. The N(umber table) and A(ssumption list) visualizations are third and fourth, respectively. For the A method, this is probably because it is only suitable for a very specific kind of question: tracing assumptions up to the results they influence is a less common task than the other way round. The N method involves interpreting Number Tables; these are quite complicated beasts for which an analyst may need more practice to be able to use them effectively.

<table>
<thead>
<tr>
<th>Question</th>
<th>M</th>
<th>G</th>
<th>S</th>
<th>A</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2.1</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2.2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3.1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4.1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4.2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>5</td>
<td>30</td>
<td>17</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 8.9: Methods used in the second part of the experiment. M = Manual (i.e., without tool), G = Graph of System Parameters and Analysis Functions, S = System Parameter list, A = Assumption list, N = Number Table.

8.3.2.4 Conclusion

The following conclusions can be drawn from the quantitative results of the controlled experiment:

• It can be stated with fair certainty that overall, the tool enables users to do analysis on analysis model(s) quicker. However, the sample size is too small to determine whether this improvement is mainly in analysis of individual models, or in comparison of models.
• It is not certain that the tool causes users to make less errors, although the results suggest a slight improvement.

• The visualizations most used were the System Parameter - Analysis Function Graph and the System Parameter List. The Number Table may be used more when users grow accustomed to it, due to its complexity. Improvement of the visualizations lies mainly in their interactiveness: it is difficult to navigate through them, and search functionality is minimal.

• Overall, the tool is received quite positively.

8.3.3 Qualitative results

The qualitative results consist of the answers given in the interviews at the end of each experiment session and the results of question 3.1 that are not considered in the tested hypotheses (see section 8.3.1.3 on page 82).

Interview at the end of each experiment session  The interview at the end of each experiment session consisted of the following questions:

• What is your opinion about the experiment / the questions posed (difficulty, amount of time, relevance)?
  Participants generally find the questions hard, but doable; the amount of time they are given (10 minutes per question) is enough most of the times. All of the participants indicate that to optimally do the experiment, they need more training time. Questions are deemed relevant; some participants indicate that the question about improvements (3.1) is less so.

• How did you like the pyQAM tool?
  The tool (or rather, its visualizations) is generally liked well; participants find it helpful and easy to use. However, the analyzed analysis model is so big that one can easily lose track, even with a visualization.

• Which visualization is the most effective for you? Do you suggest any improvements?
  The most popular visualization among the participants is the graph of System Parameters and Analysis Functions. This is also the one that is used most. However, this visualization has one major drawback: it does not provide a way to reduce its size, making it easy to lose track. One participant suggests to implement a way to highlight dependency chains in the graph, so connected assumptions and results can be easily found.
  On the second place are the Number Table and the System Parameter List visualizations. Participants indicate that the former might be of more use when they get more time to learn how to use it.
  Overall, the visualizations can be improved with a search function: a function that allows the user to enter a phrase, after which the visualization pans to the search result. This reduces the time needed to scroll through the visualizations, looking for a particular System Parameter or Analysis Function.

• If you were to be assigned the task of analyzing a Python analysis model such as SKAcost, would you use this tool?
  Fortunately, the answer to this question was a unanimous yes.

Results to question 3.1  Table 8.7 on the next page shows the answers given to question 3.1, with and without tool. In some cases, errors in the analysis model are indicated; but most comments are on the overall coding practices of the analysis model, and on the design. There are no fundamental differences between the answers given without, and the answers given with the help of the visualizations; except for the fact that the answers given with tool generally entail specific errors, while the answers given without tool are more about coding style and design of the analysis model.
### 8.4 Conclusion

This chapter showed that two of the four use cases to be supported by the pyQAM tool have been implemented, the two others have been partially implemented.

Validation of practical soundness has been performed through an experiment, of which the results are:

- The visualizations provided by the pyQAM tool enable an analyst to perform analysis more quickly; whether this is true for specifically analysis to understand an analysis model or specifically for analysis to compare analysis models, could not be determined with enough certainty,

- It could not be determined whether the quality of the answers found with help from pyQAM’s visualizations is better than without it.

In general, the participants to the experiment like the visualizations, and indicate that they would use them if they were given the task of analyzing and/or comparing an analysis model such as the one that was used in the experiment.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Without tool</th>
<th>With tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Make assumption of using biplexed pipelined FFT flexible</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Improve the error handling mechanism</td>
<td>Improve the error handling mechanism</td>
</tr>
<tr>
<td>3</td>
<td>specFOVArray assumption is not used in this component (AAProcessingFirst)</td>
<td>Many results and parameters also depend on (other) component &quot;AAProcessingFirst&quot;</td>
</tr>
<tr>
<td>4</td>
<td>Code is not well commented, <strong>init</strong> function may slow object creation/initialization due to its large size</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Remove hard-coded numbers</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>The system parameter &quot;tileBeamFOVmin&quot; is not used by any analysis function</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Too many parameters, both locally and from higher blocks. Object-oriented design is not sound – too many dependencies are hard-coded. Cost &amp; performance parameters don’t seem to be in sync.</td>
<td>costProcessing1PerTile has too many inputs. Modularization might improve things.</td>
</tr>
</tbody>
</table>

**Table 8.7: Results to question 3.1**
8.4. CONCLUSION

CHAPTER 8. VALIDATION
Chapter 9

Related Work

9.1 Introduction

The pyQAM tool that was developed and presented in the previous chapters is part of the Knowledge Architect. The Knowledge Architect is a suite of tools that is intended to help system designers, analysts and maintainers by capturing and sharing architectural knowledge. PyQAM is a tool that captures architectural knowledge from an analysis model; several visualizations have been created to enable the user to browse through the captured knowledge. Although these visualizations have been packaged with the pyQAM tool, they can be re-used with minimal effort by other capturing tools that are part of the Knowledge Architect, such as the Excel Knowledge Extractor created by Tjaard de Vries.

This chapter gives an overview of related work from several fields, which includes the Knowledge Architect as a whole and its field of Architectural Knowledge. Analysis models like the ones captured by pyQAM, provide a way of evaluating a system design with respect to certain properties or quality attributes, which is subject of the field of Architecture Evaluation Methods. For pyQAM, capturing architectural knowledge involves analyzing dependencies in a Python program; similar work is being done in the field of Reverse Engineering. Visualization of the captured model data, and comparison thereof, are subject of the Model based software development.

9.2 Architectural Knowledge

The pyQAM tool was designed as part of the Knowledge Architect, which is a suite of tools with which requirements, design decisions and their rationale, and other Architectural Knowledge can be captured from artifacts created in the process of the development of a system. The captured knowledge is stored into a repository and is browsable by the system architecture’s stakeholders.

The notion of Architectural Knowledge represents a shift from the traditional approach towards software architecture. Traditionally, the focus in software architecting was on the result it produces: a description of system to build, along with figures of the components and connectors it conceptually consists of. According to Bosch [1], this focus leads to knowledge vaporization, i.e., "virtually all knowledge and information concerning the results of domain analysis, architectural styles used in the system, selected design of the system are embedded and implicitly present in the resulting software architecture". As a consequence, changing an existing system, even with a solid description of its architecture, is difficult: the implicit or tacit knowledge present in it is difficult to recover.

The solution suggested by [1] is a new approach: focus on a software architecture as a set of design decisions rather than a collection of components and connectors. These design decisions should contain the "tacit knowledge" in the traditional software architecture description.

Architectural Knowledge is defined by Kruchten et al. [4] as

Architectural Knowledge = Design Decisions + Design

In this formula, the (architectural) design stands for the description of a software system, with diagrams and different views of the system [5, 6]; this design is augmented with design decisions.
An (Architectural) Design Decision (ADD) describes not only a decision that is made, but also its rationale, the requirements it fulfills, the considered alternatives to the decision, additional requirements that follow from the decision and prescriptions for subsequent design decisions [7].

Architectural Design Decisions (ADDs) ADDs can be defined as [8, 7]:

A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.

An ontology that further describes what ADDs are is provided by Kruchten et al. [4]. This article identifies four kinds of ADDs:

1. Existence decisions (ontocrises): decisions that state that some element or artifact will positively show up.
2. Bans or non-existence decisions (anticrises): decisions that state that some element or artifact will not show up in the design or implementation.
3. Property decisions (diacrises): decisions that state an "enduring, overarching trait or quality of the system".
4. Executive decisions (pericrises): decisions that do not relate directly to the design elements or their qualities, but are driven more by the business environment.

Each of these design decisions has a set of essential attributes:

- Epitome (the decision itself): a short textual statement of the design decision
- Rationale: a textual explanation of why this decision was made
- Scope: decisions may have limited scope, in time, in the organization or in the design and implementation
- Author, Time-Stamp, History
- State: the state of the design decision, e.g. "approved", "obsolete", or "rejected"

Between design decisions, Kruchten et al. identify 11 relations: Constrains, Forbids, Enables, Subsumes, Conflicts With, Overrides, Comprises, Is An Alternative to, Is Bound To, Is Related To, and Depends On.

Architectural Knowledge Management Tools Van der Ven et al. [9] and De Boer et al. [10, 11] envision storing AK in a Knowledge Grid, first proposed by [12]. A Knowledge Grid is an intelligent and sustainable interconnection environment that enables people and machines to effectively capture, publish, share and manage knowledge resources. A Knowledge Grid could support sharing and managing AK and ADDs. To this end, [9] devised a use case model (based on the use case model proposed by [4]) to which AK tooling should comply. This use case model was used to validate the Knowledge Architect in the context of its Excel Client tool [2].

To be able to store AK into a Knowledge Grid, a core model to represent AK with is needed. Kruchten’s ontology [4] (discussed above) and a conceptual model from IEEE standard 1471 [13] are conceptual examples of such a core model. De Boer et al. [10, 11] propose a model to which both the ontology of [4] and the model of [13] can be mapped. Another model is proposed by [14], to be used in PAKME, a web-based architecture knowledge management tool. The domain model for the Knowledge Architect, described by [2] and partially reiterated in section 3.3.1.1 of this thesis, is a core model similar to the one presented in [10, 11], be it a bit more elaborate.

Software tools to manage and share AK are commonly referred to as Knowledge Management Tools. A number of Knowledge Management Tools are compared by [15], of which the Knowledge Architect can be seen as an implementation of a Knowledge Grid.
Architect is one (although it is called by its previous name, "Matrix"). The others are the Architecture Design Decision Support System (ADDSS) [16], Archium [17, 18, 7], Architecture Rationale and Element Linkage (AREL) [19] and Process-based Architecture Knowledge Management Environment (PAKME) [20]. Another one is the Automatic Architecture Knowledge Extraction Tool (AAKET) [21].

Relation with pyQAM  The pyQAM tool itself has little or nothing to do with design decisions, around which the aforementioned Knowledge Management Tools revolve. In fact, the Knowledge Entity Design Decision is left out of consideration in the description of the domain model for pyQAM (see sections 3.3.1.1, 5.5). This is because pyQAM itself doesn’t capture design decisions: it captures the results of analysis models, and how these results have been calculated. Results of analysis models are used to make design decisions, they are (part of) the rationale part of design decisions. Therefore, the results of analysis models and how these results have been calculated qualify as Architectural Knowledge, and this information should be captured along with the design decisions made based on it.

9.3 Architecture Evaluation Methods

Architecture evaluation methods are ways in which architectures can be assessed with respect to certain quality attributes. Analysis models can be thought of as architectural evaluation tools: they calculate properties, or quantitative values for quality attributes, given a system design and a set of predetermined properties of the system. In their survey on architecture evaluation methods, Mattson et al. [22] identify four types of architecture evaluation methods:

1. **Experience-based** evaluations are evaluations based on the previous experience and domain knowledge of developers or consultants.

2. **Simulation-based evaluations**: simulations or high level implementations are used to evaluate an architecture.

3. **Mathematical modeling** uses mathematical proofs and methods to evaluate an architecture.

4. **Scenario-based** methods evaluate an architecture by creating a usage scenario and then evaluating the architecture’s quality attributes with respect to that scenario.

An experience-based evaluation method is based on Attribute-Based Architectural Styles (ABASs,[23]). In this method, architectural styles are the basis for reasoning frameworks for architectural quality attributes. The reasoning framework for an architectural style can be qualitative or quantitative, and is based on models for specific quality attributes.

A notable example of scenario-based methods is the Software Architecture Analysis Method (SAAM, [24]). SAAM consists of a number of steps in which the architecture is documented thoroughly, and scenarios are developed that describe what the system should be capable of doing. The scenarios that represent an aspect to be evaluated are evaluated. The scenarios are then ordered with respect to their priority and their expected impact on the architecture. SAAM has been used and validated extensively. Dobrica and Nemirá [25] describe methods that are extensions to SAAM. The Architecture Tradeoff Analysis Method (ATAM, [26]) builds on the SAAM, but considers multiple quality attributes (SAAM considers only one at a time) and is a way to determine trade-offs between quality attributes.

Simulation-based and mathematical modeling methods are combined in *layered queuing network models* [22]. In this method, the architecture is transformed into a model that describes the interactions between components in the architecture and the processing times required for each interaction. Such a model can be solved analytically, but in practice they are solved through simulation.

The analysis models analyzed by pyQAM can be placed in the category of simulation-based and/or mathematical modeling category of evaluation methods.
9.4 Reverse Engineering

Without documentation, or little thereof, it is difficult to understand a piece of software. The SKAcost analysis model is an example: to truly understand what it does, it is necessary to dive into its source code. The activities that are involved in finding out what a software program does, with no specification or other documentation available, are the subject of the area of reverse engineering.

The part of pyQAM that captures Numbers, System Parameters and Analysis Functions from the analysis model can be seen as a reverse engineering tool. Without any documentation (except for some annotations in the model itself), the tool is capable of capturing knowledge from a software program; this knowledge helps, when displayed properly, the analyst to understand the software program. Similar tools have been developed and are in use in the field of reverse engineering.

Program Slicing and the Program Dependence Graph  In 1981, Weiser introduced program slicing [27, 28, 29]. Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics; the technique deletes those parts of a program which can be determined to have no effect on the semantics of interest. A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest in the program. This can be done in two ways: statically – producing a static program slice, the kind that Weiser introduced – or dynamically, yielding a dynamic program slice. The latter was introduced by Korel and Laski [30]; the difference with the static slice is that a static slice shows the slice for any run of a program (i.e., all conditional branches of the program are contained in the slice), while the dynamic slice only contains information for one execution of the program. Tip [31] gives an overview of program slicing techniques.

The distinction between static and dynamic slicing is the same as the one made in section 5.3.2.1 between static and run-time analysis. In fact, construction of a (dynamic) program slice deals with some of the same problems as the methods discussed in chapter 5. Both methods determine dependences between values in a program execution. The dependences between values found in chapter 5 only are data dependences, while a program slice also contains control dependences. On the other hand, a program slice doesn’t determine which values are semantically equivalent.

Another distinction that can be made, is that between backward slices and forward slices. A backward slice selects those statements of a program that influence a particular statement in the program; a forward slice selects those statements that are influenced by a particular statement. Interestingly, this corresponds to the two directions of analysis discussed in section 3.3.1.1. The data dependencies found by forward program slicing are those that are found by starting at a Number and determining which Numbers depend on it; backward slicing corresponds to starting at a Number and determining which Numbers it depends on.

Ottenstein and Ottenstein [32] came up with the program dependence graph (PDG) in order to transform Weiser’s program slicing algorithm into a graph reachability problem, making the algorithm easier to understand and more versatile. Each node in a PDG represents a statement or control predicate; edges correspond to data and control dependences. The System Parameter - Analysis Function - Number graph from section 6.3 contains all the data dependences also contained in a PDG, and adds semantic relations (e.g. the relation typed by between Number and System Parameter). However, there is one subtle but important difference: whereas the PDG contains data dependences between statements and control predicates, the System Parameter - Analysis Function - Number graph models dependences between variables and variable instances. A modification of the PDG, proposed by Jackson and Rollins [33], does (also) data dependences between variables, and is a closer match to our graph.

The application that Weiser had in mind for program slicing was mainly debugging: if a variable at some point in a program contains an erroneous value, the bug causing the error is most likely found in the (backward) slice with respect to that point. Today, program slicing finds its application in a multiple fields; it is used for parallelization, program differencing and integration, software maintenance, testing, reverse engineering and compiler tuning [30]. Particularly, program differencing [34] is of interest: this is the task of analyzing an old and a new version of a program in order to determine the set of program components of the new version that represent syntactic and semantic changes. Program differencing is similar to what we try to do with analysis models;
however, two analysis models to be compared ("differenced") may not be in the same programming language, are not even restricted to being implemented as a program at all. But, the graph of System Parameters, Analysis Functions and Numbers containing information that is also contained in a PDG, techniques of program differencing that work on a PDG might still be usable to compare analysis models.

A technique that satisfies this requirement is the one presented by Krinke in [35]. The technique identifies similar code by using a PDG; of course, identifying similar “code” also means that “code” that is not similar is also identified. The algorithm proposed by Krinke is a very generic one: it can be executed on any attributed directed graph (i.e., a directed graph that has attributes attached to its nodes and its edges) such as a PDG. It works by finding maximal similar subgraphs. Two graphs \( G \) and \( G' \) are similar if for every path starting at one node \( n \) in \( G \), there exists a path starting at one node \( n' \) in \( G' \) such that, when both paths are mapped against each other, all attributes of the nodes and edges of the path starting at \( n \) in \( G \) are identical to the attributes of the nodes and edges of the path starting at \( n' \) in \( G' \). To have the algorithm run in acceptable time, the considered paths are \( k \)-limited: only paths up to a certain length are considered. Furthermore, only subsets of \( G \) and \( G' \) should be considered; the algorithm has running time quadratic with respect to the number of nodes considered. According to Krinke, this subset should be based on specific features of the vertices, which is highly application specific.

The method of identifying similar code may be applicable to the System Parameter - Analysis Function - Number graph. If so, it can be implemented as a semi-automatic way of identifying parts that are similar in two analysis models. Semi-automatic, because the algorithm only runs in acceptable time if a small number of nodes is selected to find maximal similar subgraphs. In practice, this would mean that an analyst selects pairs of corresponding nodes (presumably System Parameters) in the two analysis models to compare, and then executes the algorithm to see how large the maximal similar subgraphs surrounding these nodes are.

**Portable Bookshelf and Grok** [36] introduced the the Portable Bookshelf (PBS) toolkit, an implementation of the Portable Bookshelf paradigm for the "presentation and navigation of information representing large software systems" [37].

A "Software Bookshelf" for a large software system provides a web-accessible structure for storing information (documents and source code) about a system; this structure is based on the hierarchical decomposition of the software system into subsystems [36]. This structure can be obtained by extracting so-called facts from source code. A fact is a relation between two entities: an example fact would be "functionA calls functionB", where "functionA" and "functionB" are the entities, and "calls" is the relation. Some facts cannot be extracted from source code: for example, a hierarchical decomposition (which can also be described with facts involving "part-of" relations) may not be obvious from source code and needs to be added by an expert.

When all facts are complete, these facts must be manipulated to yield a graph that can be visualized in an understandable way: in most cases, there is too much detail (too many facts) in the original set of facts to be grasped by human intelligence alone. Remember that the same problem exists in the case of the System Parameter - Analysis Function - Number graph extracted by pyQAM. As noted by the validation experiment’s participants (section 8.3.3), this graph is too large to be usable effectively: the graph is so large that a user easily loses track. One solution is to show only a high-level abstraction of the graph, hiding most details, enabling the user to zoom in to low-level details when needed. Putting this into terms of facts, the solution is to deduce and add high-level facts, discarding most of the low-level facts.

For this purpose, Grok, an executable language that “allows us to specify and reuse common architectural transformations” [38], is also part of the PBS toolkit. The Grok language has operations based on Tarski Relational Algebra [39], allowing the user to define transformations on a set of facts, which are effectively transformations on the graph of entities and relations. One can, for example, remove nodes that represent lower-level abstractions, rerouting the edges that connect these nodes to the rest of the graph to their parent ("containing") nodes. These "architectural transformations" provide a powerful way of transforming a collection of detailed facts into a collection of facts with less detail.
The Rigi Environment  A tool similar to the Portable Bookshelf is the Rigi environment [40, 41]. From the Rigi User's Manual [42]:

Rigi is a system for understanding large information spaces such as software programs, documentation, and the World Wide Web. This is done through a reverse engineering approach that models the system by extracting artifacts from the information space, organizing them into higher level abstractions, and presenting the model graphically.

Rigi can be divided into three parts; one for information representation, storage and exchange ("repository"), one for fact extraction and one for interactive graphical manipulation ("graph editor"). This is the same subdivision made in the design of the Knowledge Architect tool suite (cf. sections 1.2 and 4.4).

Rigi performs fact extraction automatically on the source code of a system to reverse engineer, and stores them in a repository. The graph editor is used to visualize and manipulate these facts. The same problem pointed out by participants in pyQAM’s validation experiment (section 8.3.3), the problem of visualizing a large graph, exists in Rigi’s visualizations. Rigi’s solution is to use so-called Simple Hierarchical Multi-Perspective (SHriMP) views; SHriMP being a "visualization technique that presents software structures using fisheye views of nested inclusion graphs" [43]. With these SHriMP views, it is possible to magnify part of a graph, while keeping the rest of the graph at it’s original size. Thus, the user can get a detailed view of a graph, and not lose track of where she is.

9.5 Model Based Development

Model based development is a field that looks into how models can be used in software development. The 2008 workshop on Comparison and Versioning of Software Models (CVSM’08), held at the International Conference on Software Engineering 2008 (ICSE’08), yielded a number of articles that are relevant to the comparative visualizations discussed in chapter 7.

Diff and Merge  Bendix and Emanuelsson [44] give an overview of work that has been done on versioning, differencing, comparing, merging and union in the domain of models: they are looking for a way to achieve better support for teamwork in model-driven development. The models they are considering are mainly UML models. The mentioned operations, however, can also benefit the comparative visualizations of pyQAM: as long as the methods proposed are generic enough to work on other graphs than UML models, they may be applicable.

A promising set of definitions and operations is given by Alanen and Porres [45]. In this work, they define a number of operations on models that together enable the implementation of "diff" and "merge" operations. They state a number of requirements for a model to be compatible to these operations:

A model consists of a set of linked elements. Each element in a model conforms to a type, called a metaclass, while each link between two model elements conforms to a meta-association. The meta-association is further divided into two association ends or metafeatures. [...] Attributes and associations in the metamodel have two other properties: multiplicity and order. The multiplicity describes a constraint on the number of values that can be stored in the attribute or association end. Additionally, these elements may be ordered. [...] The complete definition of the metaclasses and meta-associations allowed in a modeling language is called the metamodel. [...] While attributes are unidirectional features, it is of utmost importance to keep the bi-directionality of associations; if an element $A$ has an association to $B$, then $B$ must have an association back to $A$.

In addition, Alanen and Porres rely on each element having a unique identifier (UUID). The output produced by pyQAM should be adaptable to these requirements.

Altmanninger’s approach [46] takes into account both syntactical and semantical aspects of models. This allows for more precise conflict detection (in the case of merging models), and more robust detection of differences between models.
Visualization of Model Differences  Wenzel [47] leaves the actual differencing to a library, and uses polymetric views (introduced by Lanza and Ducasse [48]) to visualize the differences found. In a polymetric view, all entities of a model (nodes in a graph) are represented by rectangles; up to five metrics can be mapped onto the rectangles’ height, width, color, and horizontal / vertical positions. So-called "outliers", entities that have an abnormal value for one of the metrics (e.g. a huge number of lines, if the entities are code files) are easily spotted visually when visualized in a polymetric view. The polymetric views allow the user to view the system on a high level (i.e., details are hidden), and to zoom in to see more details when necessary.

To be able to visualize differences in a polymetric view, Wenzel defines a set of difference metrics. These consist of the number of nodes changed and the number of node attributes changed. These need to be corrected for their significance: a small number of very significant changes may be more important than a large number of insignificant changes; which changes are significant and which aren’t is domain specific.

These polymetric views that are produced reveal the differences between two models at a glance. A difference from Lanza’s polymetric views is that each "difference rectangle" also gets a colored border: yellow indicates updates, green stands for insertions, red for deletions, blue shows that the node has been moved, and magenta expresses reference changes. Nodes that belong to multiple difference types are framed in cyan. Unchanged nodes are left black.

A similar view could work for comparison of analysis model visualizations: as in the Wenzel case, there are two graphs of which the differences need to be visualized. Polymetric difference views are a visualization equivalent to the diff views that are commonly available to compare two versions of a text.
Chapter 10

Conclusions, Future Work

10.1 Conclusions

What kind of knowledge is present in analysis models, how to extract this knowledge and how to use this knowledge to compare analysis models to each other, is what this thesis is about. The answers and solutions found to these questions have been brought into practice with the development of pyQAM. PyQAM is a tool built for ASTRON to understand the SKAcost model more easily, faster and with less errors. It is meant to compare the SKAcost model to the "previous" Excel analysis model and the "next" SKAsim analysis model, although functionality to do so has not been fully implemented in pyQAM. The pyQAM tool has been designed both as a part of the Knowledge Architect suite of tools and as a stand-alone tool to be easily deployed on the workstations of ASTRON engineers. The three most prominent aspects of pyQAM that are described in this thesis are capturing of analysis model data, visualizing it, and comparing captured data from two (or more) analysis models to each other.

In the following, the research questions formulated at the beginning of this thesis are re-iterated and a short summary of the solutions found is provided with them.

What are the causes for different analyses of the same system?

This question was treated in section 3.2. The causes meant are causes that lie outside the analysis model: that is, they stem from the decisions made at the design time of the model, and circumstances that were in effect at the time. Four causes were identified:

1. The analysis models have different levels of detail
2. Different assumptions are made.
3. Results of one analysis model are assumptions of the other
4. Different formulas are used

How does one find key differences between different analyses of the same system?

Section 3.3 divides this question into five sub-questions:

1. What knowledge is needed to understand an individual analysis model?
2. What knowledge is needed to compare two analysis models that one understands?
3. What are the aspects of acquiring knowledge of an individual analysis model and of comparing two analysis models?
4. What are significant values in an analysis model?
5. What are the prerequisites for successfully comparing analysis models?
In response to (1), the parts from the domain model presented in [2] that have to do with quantitative modeling are reiterated in section 3.3.1.1; this domain model is improved on a number of points in section 5.5. With this domain model, the required knowledge to understand an analysis model can be formally represented.

Comparing two analysis models (question 2) that are represented in this domain model means mapping System Parameters from one analysis model to System Parameters in the other analysis model, and mapping Analysis Functions from one model to Analysis Functions in the other model (section 3.3.2). With this mapping in place, it is possible to determine the differences and similarities between the models: the entities that are not mapped (i.e., System Parameters and Analysis Functions in either model that don’t have a counterpart in the other model) are differences.

Having established what knowledge is needed to understand and compare, section 3.3.3 looks for aspects of acquiring this knowledge (question 3). Two main activities are identified: dependency analysis and mapping semantics.

Dependency analysis involves tracing dependences between values through calculations: if a calculation has a value $a$ as input and a value $b$ as output, value $b$ depends on $a$. Through dependency analysis, an analyst can match assumptions to the results they influence and vice versa; dependency analysis is also employed to gain more localized understanding of a part of an analysis model in which case parts of different dependency chains are analyzed. Dependency analysis can be performed in two directions: in a forward manner, starting with assumptions and tracing (reverse) dependences to the results they influence; or in a reverse manner, starting at a result and tracing dependences to the assumptions they are influenced by.

Mapping semantics, an analyst gathers the knowledge required to compare analysis models, as established with question 2: this activity involves mapping System Parameters and Analysis Functions of the models to compare. Not all System Parameters and Analysis Functions need to be mapped: if a sufficient proportion of the mapping has been made, the remaining part can be derived. Sections 7.2 and 7.3 elaborate on details of how such a mapping should be performed, and to what extent a partial mapping can be completed by deriving remaining parts.

An important issue is how to determine the significance of values in an analysis model (question 4). Section 3.3.4 determines that there are three kinds of values in an analysis model: assumptions, intermediate results and results. Significance of assumptions is determined by how much influence they have on results, and how significant the results they influence are. Intermediate results are by definition insignificant; however, it is not possible to determine with full certainty which results are "intermediate" results. It is also not possible to determine with certainty which results are more important than others. Which result is more important than another, and which results are "intermediate", is in a lot of cases a matter of opinion and circumstance. However, it is possible to attach a likely significance to each value in an analysis model; section 3.3.4.1 proposes a significance rating in order to do so.

Section 3.3.5 concludes the answer to this research question by summarizing the prerequisites for analyst to be able to successfully compare analysis models:

1. To have enough domain knowledge to identify and map System Parameters,
2. To have enough knowledge to identify and map Analysis Functions,
3. To be able to perform dependency analysis on the format of any of the analysis models
4. To be able to determine which values are significant in the analysis model and which aren’t.

A solution to the last research question, summarized below, minimizes or eliminates these prerequisites.

How can we make finding differences between different analyses of the same system easier, less error-prone and quicker?

This question is more accurately posed by re-formulating it into three new questions (section 3.4):

1. How can we minimize the number of prerequisites for finding these differences?
CHAPTER 10. CONCLUSIONS, FUTURE WORK

10.2 Future Work

Future work on the Knowledge Architect and pyQAM may go into many directions; a few of them that are the logical next steps are formulated here and elaborated on below. Firstly, following up on the results of the validation experiment that was performed, visualizations of captured data should be improved. This not only benefits the visualization capabilities of pyQAM, but of any quantitative modeling tool that is part of the Knowledge Architect, because visualizations are generated from data expressed in the domain model used by all Knowledge Architect tools. Secondly, the mapping methods and comparative visualizations that are described in chapter 7, but not implemented due to lack of time, are also a candidate. Lastly, to improve the effectiveness of visualizations, both of single models and of multiple models, future work can involve an extension of the domain model to incorporate more fine-grained information types such as the nature of calculations performed in Analysis Functions.

Improvement of existing visualizations The existing visualizations that are packaged as a part of pyQAM, and can be used with quantitative data from any other Knowledge Architect tool\(^1\), lack interactiveness according to the results of the validation experiment. A similar problem was reported by [2]: the Excel Client’s main drawback is that it doesn’t have a dynamic graph lay-out algorithm, causing a confusing repositioning of the graph each time it needs to be redrawn; in effect, diminishing the interactive usefulness of Excel Client’s graph visualization. A next step to improve both the Excel Client and pyQAM, is to design new visualizations that have the required interactiveness. Some desired properties of these new visualizations are:

- Collapse / expand parts of the view: the user should be able to collapse parts of a view into higher-level elements, focusing only on a relatively small number of details;

---

\(^1\)Currently, the only other tool that produces such quantitative data is the Excel Knowledge Extractor
10.2. FUTURE WORK

• Search functionality: the user should be presented with a possibility to enter search terms in order to find one entity in a large collection;

• Trace functionality: graph views should provide the user with a way to easily trace paths in the graph. For example, in a dependency graph, this could be implemented by providing the user with a function that highlights all results depending on an assumption;

• Dynamic and/or animated update: each time the user performs an action that requires the view to be change, the view should not be redrawn entirely but changed in such a way (e.g. by animation) that the user can see what happens while it happens.

Development of mapping tools and comparative visualizations Chapter 7 provides a starting point for the development of mapping functions and comparative visualizations. A mapping tool should enable the user to manually map elements of one analysis model to another, after which she executes an algorithm (by pressing a button) that calculates derived mappings. She should then be able to correct errors: whenever a mapping is derived by the tool, this mapping should be clearly indicated as derived; the user should be given the opportunity to undo mappings if they prove to be incorrect. Ideally, this mapping tool is accompanied by a visualization that shows of each of the models that are mapped to each other, which elements have been mapped and which haven’t. Again, interactivity influences greatly the ability of the user to work with this kind of tool.

Once analysis models have been mapped, several comparative visualizations can be applied such as the side-by-side and combined views discussed in section 7.4. Ideally, these visualizations have the same properties as described above for visualizations of single models.

Extension of the domain model The data extracted from an analysis model, represented in the domain model described in this thesis, captures the nature of the calculations made in that analysis model. When comparing two analysis models to each other that model the same system but have entirely different designs, the current domain model may not have enough detail to be able to represent these two analysis models in order to be mapped to each other. Furthermore, visualizations of single analysis models may benefit greatly if the domain model is augmented with finer-grained information.

The current domain model captures relations between Numbers, System Parameters and Analysis Functions, among others. However, what happens inside an Analysis Function is not captured. How calculations, mathematical operations on values, are spread over Analysis Functions is a decision of the analysis model’s author(s). In different analysis models by different authors at different times, the decisions made with respect to the subdivision of calculations into Analysis Functions may differ significantly, even though the sets of calculations performed by both analysis models are similar. A domain model that, in addition to System Parameters, Numbers, Analysis Functions and their interrelations, also models the mathematical operations performed on Numbers within Analysis Functions, does contain enough information to detect this kind of similarity. How this kind of similarity may be detected, is an interesting topic for future work.

Visualizations, both of single analysis models and of multiple (combined) analysis models, can also benefit from a domain model that is capable of representing these operations on Numbers. For example, with the current domain model, it is possible to determine that one value influences a number of other values. This influence is modeled with a dependence relation. However, how strong this influence is cannot be determined. With a domain model that also models operations on Numbers, it is possible to calculate this influence. The amount of influence of one value on another can be modeled as the impact of the dependence relation that exists between them. The impact of dependences may be defined in terms of the mathematical derivatives of the operations that cause the dependence relation. When each dependence relation has an associated impact, this can be used to improve visualizations of the analysis model: for example, System Parameters with a large cumulative impact on other System Parameters can be shown with more prominence than others. This may also improve the assessment of significance of System Parameters and Numbers: their respective significance ratings can be determined more precisely.
Bibliography


[25] L. Dobrica and E. Niemel


