University of Groningen

3rd SC@RUG 2006 proceedings

Smedinga, Rein; Avgeriou, Paris

[Link to publication in University of Groningen/UMCG research database](#)

SC@RUG 2006 proceedings

Rein Smedinga
Paris Avgeriou
editors

2006
Groningen

Contents

Contents

About SC@RUG

Introduction   SC@RUG (or student colloquium in full) is a course that master students in computing science follow in the first year of their master study at the University of Groningen.

In the academic year 2005-2006 SC@RUG was organized for the third time as a conference. Students wrote a paper, participated in the review process, gave a presentation and were session chairs during the conference.

The organizers Rein Smedinga and Paris Avgeriou would like to thank all colleagues, who cooperated in this SC@RUG by collecting sets of papers to be used by the students and by being expert reviewers during the review process. They would also like to thank Femke Kramer from the Faculty of Arts for her help in organizing this course.

In these proceedings all accepted papers are published.

Organizational matters   SC@RUG 2006 was organized as follows. Students were expected to work in teams, consisting of two persons. The student teams could choose between different sets of papers, that were made available through *Nestor*, the digital learning environment of the university. Each set of papers consisted of three papers about the same subject (within Computing Science). Soms sets of papers contained conflicting meanings. Students were instructed to write a survey paper about this subject including the different approaches in the given papers. The paper should compare the theory in each of the papers in the set and include own conclusions about the subject.
Some teams proposed their own subject.

After submission of the papers individual students were assigned one paper to review using a standard review form (see Appendix A of the first StudColl2004 proceedings). The colleagues who had provided the set of papers were also asked to fill in such a form. Thus, each paper was reviewed three times (twice by peer reviewers and once by the expert reviewer). Each review form was made available to the authors of the paper through *Nestor*.

All papers could be rewritten and resubmitted, independent of the conclusions from the review. After resubmission each reviewer was asked to re-review the same paper and to conclude whether the paper had improved. Re-reviewers could accept or reject a paper. All accepted papers can be found in these proceedings.

All students were asked to present their paper at the conference and act as a chair or as discussion leader during one of the other presentations. The audience graded both the presentation and the chairing or leading the discussion. Femke Kramer of the Faculty of Arts gave an introductory lecture about general aspects of presentation techniques to help the students with their presentation. She also did a workshop on writing scientific papers.

Students were graded both on all three aspects: the writing process, the review process and the presentation. Writing and rewriting counted for 50% (here we used the grades given by the reviewers and the re-reviewers), the review process itself for 15% and the presentation for 35% (including 5% for the grading of being a chair or discussion leader during the conference). For the grading of the presentations we used a selected number of judgements from the audience and calculated the average of these.

On January 23rd and 24th, the actual conference took place. Each paper was presented by both authors. Both days, we had ten presentations, each consisting of a total of 30 minutes for the presentation and 10 minutes for discussion. As mentioned before, each presenter also had to act as a chair or as discussion leader for another presentation during that day. The audience was asked to fill in a questionnaire and grade the presentations, the chairing and leading the discussion.

Thanks   We could not have achieved the ambitious goal of this course without the invaluable help of the following expert reviewers:
– Ronald van der Berg
– Sybren Deelstra
– Jan Jongejan
– Gerard Renardel
– Marco Sinnema
– Jan Salvador van der Ven
– Michael Wilkinson

Also, the organizers would like to thank the *School for Computing Science* for making it possible to publish these proceedings.

Paris Avgeriou,
Rein Smedinga

Three Methods for reducing clutter, a review

Hubert ten Hove, Marco van der Kooi

University of Groningen, Dept. of Computer Science,
Oude Boteringestraat 44, 9712 GL Groningen, The Netherlands
`h.ten.hove@student.rug.nl, m.r.van.der.kooi@student.rug.nl`

Abstract. Every computer user has encountered images or screens that were unreadable because of the fact that too much data was presented or because the data was too closely clustered. This phenomenon is called clutter. Scientists have addressed the problem of clutter and presented some methods to prevent and locate clutter. The Visual Information Density Adjuster (VIDA) uses layering to reduce the amount of information displayed and works with the constant information density principle. The Feature Congestion method is used to locate clutter within images and screens. It provides methods to solve cluttering, for example with the use of colours to bring the useful information to the front of the image. Other methods involve sampling, displacement and user perception of 2D scatter plots.

1 Introduction

Anyone who has been in front of a computer has seen screens or images that are so full of information that nothing is reasonably understandable anymore. This phenomenon is called clutter. There are several researchers that have studied the problem of clutter. Several proposals have been made to measure or reduce the amount of clutter. Rosenholtz, Li, Mansfield and Jin present a measuring technique for clutter, their feature congestion [1]. Woodruff and Stonebraker propose a program called VIDA [2] which is an extension of the Datasplash program [4]. Bertini and Santucci have a solution to improve the readability of 2D scatterplots [3].

How can one judge if an image is cluttered, research by Rosenholtz, Li, Mansfield and Jin [1] found that some people judge the amount of clutter by thinking of task and how difficult it is to perform that task on a given screen. If a task is difficult to perform with the image presented to the user, the perception of the image is cluttered. They also power that users judge the amount of clutter according to their experience and knowledge. So by increasing the difficulty to perform a task, clutter affects the work speed of the user. But not only speed is affected, if an image is too cluttered the user might derive wrong conclusions about the data presented in the image. In that case clutter also affects the results. This makes clutter a highly undesirable phenomenon that has to be solved.

Clutter is known in many forms. On websites we see cluttering when the sheer amount of options is at such a level that the user is not able to find what he is looking for. In diagrams like scatter plots, the dots or lines can be so close together that a good interpretation of the diagram is difficult.

Many things can cause clutter, for example the organization, structure of the symbols and figures within an image can be so distorted that the image seems cluttered. Reorganization of the data presented in the image can solve these problems. Furthermore the use of colour can decrease or increase the amount of clutter perceived by a user. Too much text in an image or screen also does not improve the readability.

The goal of this paper is to inform the reader about clutter and what problems arise with this phenomenon. Furthermore we review some methods that present a solution for clutter and recognize clutter.

In this paper we first describe the methods that have been researched for solving and measuring the problem of clutter and their results. The paper is concluded with a discussion in which we will discuss the methods and results and also look at the possibilities of combining those methods.

2 Three methods to measure or reduce clutter

2.1 Feature Congestion

Feature Congestion is proposed by Rosenholtz, Li, Mansfield and Jin [1] and is used to assess the amount of clutter a display or image contains. The method is based on their saliency model. The statistical saliency model gives a representation of the local distribution of features by their mean and covariance. This is equivalent to representing the distribution by a set of covariance ellipsoids in the appropriate feature space.
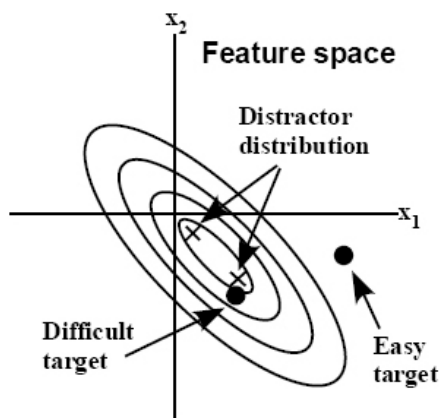
Fig. 1. The Statistical Saliency Model

Figure 1 shows a graphical depiction of this model. Starting with the innermost each ellipsoid is one more standard deviation away from the mean feature vector. According to long term research in this area those features are contrast, colour, orientation and motion [5]. Using this model a target with a feature vector on the n ellipsoid has a saliency n. When adding lots of new items to a display it becomes more and more difficult to add a new salient item, Rosenholtz, Li, Mansfield and Jin call this Feature Congestion. Clutter is measured by the local variability in certain key features. The easier it is to add a salient item to a display the less cluttered the display is.

They have implemented their model and performed a study in which they had a group of people who judged the amount of clutter in an image. They compared them with the results of their method. The results from the Feature Congestion research pointed out that their saliency model which they used to measure the amount of clutter on a screen / image performed well. After performing a average Spearman rank-order correlation and comparing the results for the users and the Feature Congestion method they concluded that their was a significant correlation of 0.83. This result was comparable with the measured correlation between the different observers, and leads to the conclusion that the model performs well.

2.2 VIDA

VIDA, short for Visual Information Density Adjuster [2], is a measure to create clutter free zoomable interfaces based on the DataSplash database visualization environment [4]. DataSplash has a system in which specific information from the database is represented in different layers. While zooming, layers will become visible according to the zoom ratio. At the overview level only the first layer might be visible, although at the detail level the top level might be invisible, but a few bottom layers might be shown. The layers can overlap each other, which can be controlled by the layer manager [2] that can adjust which layers are shown at a certain zoom level.

An example of a DataSplash visualisation can be a country map. The application user will only see a global representation of the country when viewing the entire map. Only the important cities and borders are shown. As soon as the user zooms in on a certain area of the map, the detail of the map increases. While the zoom ratio increases, smaller villages and roads appear.

The idea on which VIDA is based is called the Principle of Constant Information Density [2] guideline. According to this principle, the number of objects on a screen should continuously remain at the same level. In order to achieve such a constant information density old information has to be removed from the screen in case new information is added. In a zooming environment like DataSplash this means the detail of an object should increase when the user zooms into it. However to maintain a constant information density, the surroundings of the object should contain less information.

According to studies, the users of the original DataSplash application found it quite difficult to create visualizations that had the right amount of detail at any zoom ratio. The layer manager of DataSplash could not sufficiently indicate the amount of information density on a certain layer at a certain elevation. An automated system to adjust the amount of information was needed. To accommodate this need Woodruff and Stonebraker extended the DataSplash database visualization environment [4] in such a way that it would comply with the guidelines of Constant Information Density. Furthermore a feature was added to the layer manager so it can display the information density at certain zoom levels. This improved version of DataSplash with the Visual Information Density Adjuster can be seen in figure 2.
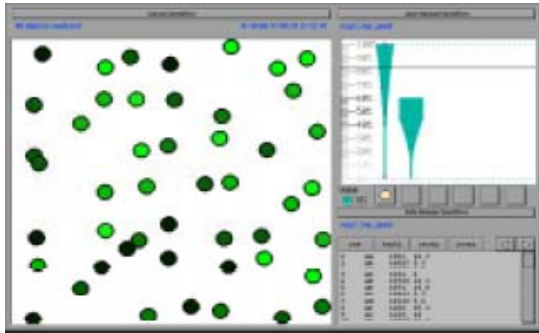
Fig. 2. A visualization of selected companies from the Fortune 500 and Global 500 lists.

"The following text is quoted from [2]. The width of each layer bar at a given elevation now represents the density of the layer at that elevation. The minimum and maximum density bounds are set to 10 and 100 objects, respectively. The colors of the tick marks on the left side of the layer manager indicate the density values at given elevations. Elevations 40%-60% are too dense, elevations 14%-38% and 62%-100% have appropriate density, and elevations 0%-12% are too sparse."

To prove the functionality of the Visual Information Density Adjuster, an informal study has been set up to test it. A small group of participants was asked to perform a set of tasks with and without a constant information density, using a prototype written in Java. The participants were given instructions how to use the prototype beforehand. The way the participants navigated through the applications was monitored.

The informal user study has shown that most participants could complete the tasks that were set easier with the application with a constant information density. Furthermore many users responded positively to the fact that it was easier to read than similar methods without a constant information density. The exact implementation of the navigation through the layers was criticized by the participants. This was done by a panning mechanism, whereas most users would have preferred a more straightforward solution like scrollbars. The tasks that were set out for the participants were also found confusing.

2.3 Improving 2D Scatter Plots

Another form of clutter is encountered in scatter plots. Bertini and Santucci have researched ways to improve scatter plots using sampling, displacement and user perception [3]. In a scatter plot some of the items can overlap resulting in a cluttered image. Bertini and Santucci divide the image into several sampling areas, within those areas two values are calculated, real density and represented density. A target density is calculated using the density data from all areas. When the represented density is lower than the target density they use pixel displacement to reveal overlapping items until target density is reached. This process will result in a less cluttered scatter plot which is a better representation of the data than the original image.

The difference between the various sampling methods as mentioned by Bertini and Santucci can be seen in figure 3. The leftmost scatter plot is the original plot without any improvements. The centre plot uses the old sampling method. The rightmost scatter plot uses the methods developed by Bertini and Santucci.
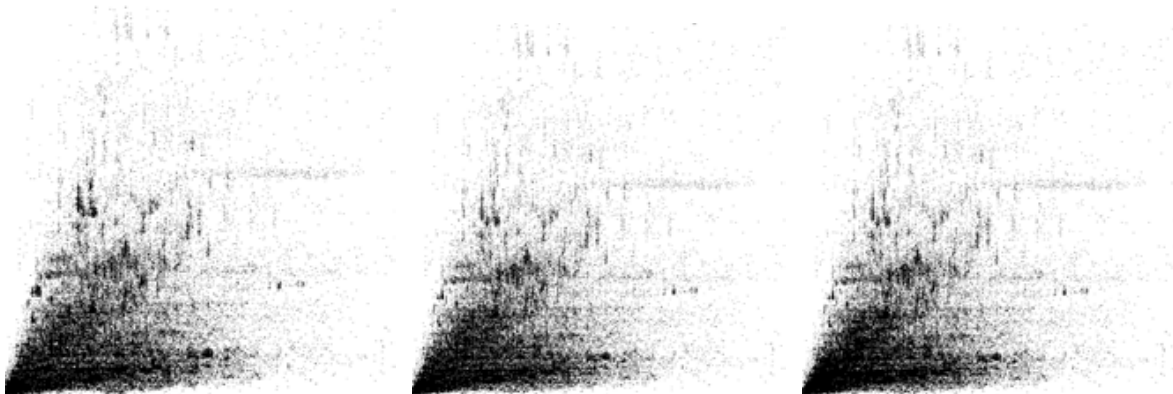
Fig. 3. From left to right: The original scatter plot; the old sampling method; the new method.

To obtain knowledge about the perception of people a user study was performed. The study focused on presenting the people with a uniform density screen which contained three more dense zones, they were asked to find those zones. These experiments were used to gain knowledge about the perception of people, so they could make the scatter plots display the results in such a way that the user would get the correct perception of the image.

The results of the user study performed by Bertini and Santucci were used to calculated what increment they would have to choose to guarantee that 70% of the end users would notify the density difference. So an area A must have at least a density increment of X to be perceived denser than area B. They also tested their method using a data set, the resulting images were showing more detail than the original image, thus proving that their method is improving 2D scatter plots.

3 Discussion

3.1 Feature Congestion

As seen in the results of the user study performed by Rosenholtz, Li, Mansfield and Jin [1], the Feature Congestion method is a good measure for clutter. By not telling the participants of the study a sound definition of clutter, the participant were not searching for a pattern which fit the definition but judged clutter to what according to them clutter was. In their paper they mention that knowledge and expertise influence the amount of clutter perceived. In the study a task was presented to the participants were no specific knowledge and expertise was required, to single out these influences.

Feature Congestion is designed to handle clutter in maps. Some of the methods used in Feature Congestion can also be used to aid designers in adding attention-grabbing elements to the display. The methods can point out a position in the image to add the element and what colour the element should be to grab the most attention. A study is needed to check for the possibilities in this field which may result in the integration of Feature Congestion methods into graphical software program

3.2 VIDA

Woodruff and Stonebraker who developed the Visual Information Density Adjuster conducted an informal study in which they let a few participants use their system to perform a set of tasks. The subject of these tasks was to find certain companies with a specific economic growth, using data from the Fortune 500 and Global 500. The survey they held after the study showed that the users found the method of zooming with a constant information density practical. However they found the tasks to be quite difficult and complex.

The goal of their research was to measure the time it took for the participants to complete the tasks and to monitor through which different layers of the application the user moved and how much time the users spent in each layer. However paragraph 6.8, User Response in [2] states the following:

"Several users stated that they found the task confusing. Our intent was that participants would infer a relationship between the variables presented and the variable they were supposed to predict (revenue growth). For example, they might infer that companies with fewer employees would be smaller and therefore more likely to experience rapid growth. We hoped that they would infer this relationship by exploring the relationships that were explicitly present in the graph. The participants who said they had been confused appeared from their comments to have performed this infer-

ential task, but were not certain they had been correct in doing so. In retrospect, the inferential nature of the task should have been made more clear."

The users in this survey mentioned afterwards that the tasks were confusing. An obvious result of this is that more time was needed to complete the tasks than would have when the users would not have been confused. If this is the case, it will negatively affect the result given in paragraph 6.6 of [2]. Therefore we think that using a predefined group of participants, given tasks that suit them more would give a better indication of the pros and cons of the Visual Information Density Adjuster. We suggest a new user study with a well defined search question so that the data collected from the monitoring process is more representative.

3.3 Improving 2D Scatter Plots

The article Improving 2D scatter plots effectiveness through sampling, displacement and user perception by Bertini and Santucci is a report on how to improve the information density at certain areas of a scatter plot. The methods created by Bertini and Santucci try to create the preferred information density for a given part of the scatter plot. By using sampling and displacement on just a small part of the plot, the information becomes clearer for that area, whereas the areas that do already have the preferred information density remain unchanged. The loss of precision that occurs when using these methods will not affect visualisations such as the scatter plot, but might not be desired for other types of visual representations. Therefore the methods proposed by Bertini and Santucci are very specific to scatter plots only.

3.4 Combining VIDA with Feature Congestion

VIDA works according the principle of constant information density. This means the screen displays a constant degree of clutter. The measuring technique in VIDA that asses the amount of clutter a display contains is not the most robust method. To improve the working of VIDA the saliency model of feature congestion must replace the current method for assessing the information density. Not only gives this model a measure for the information density but it also calculates the features which have to be added to and image if it is placed on the screen. With the model calculations can be made to display new data, that is presented in the next zoom layer, in the most effective way.

In VIDA all views have the same information density. Information density in an image gives an measure of clutter for that image. The saliency model of feature congestion measures the amount of clutter an image contains, so a constant measure of clutter should give the same result as a constant measure of information density. Therefore the algorithm for measuring constant information density in VIDA can be replaced with the saliency model.

By replacing the methods, the saliency model should improve the measure of clutter which VIDA uses. The improvement in method gives a more accurate measure of

clutter / information density. This will give a better result of constant information density which is the core of VIDA. And with the ability to calculate the most effective way to display new data according to its features, more information can be displayed with the same information density.

References

1. Rosenholtz, R. Li, Y. Mansfield, J. and Jin, Z., Feature Congestion: A Measure of Display Clutter, Cambridge, 2005.
2. Woodruff, A. and Stonebreaker, M., Visual Information Density Adjuster (VIDA), Univ. of California, Berkeley, 1998.
3. Bertini, E. and Santucci, G. Improving 2D scatterplots effectiveness through sampling, displacement, and user perception, Rome.
4. Aiken, A. Chen, J. Stonebraker, M. and Woodruff, A., Tioga-2: A Direct manipulation Database Visualization Environment, Proc. 12th Int'l Conf. on Data Engineering, New Orleans, Louisiana, Feb 1996, pp. 208-217.
5. Wolfe, J.M. Visual Search. in H. Pashler, ed., Attention. University College London Press, London, U.K., 1998.
6. Woodruff,A. Wisnovsky, C. and Taylor, C., Zooming and Tunneling in Tioga: Supporting Navigation in Multidimensional Space, Proc IEEE Symp. on Visual Languages, St. Louis, Missouri, Oct. 1994, pp. 191-193.
7. Woodruff, A. Su, A. and Stonebraker, M., Navigation and Coordination Primitives for Multidimensional Browsers, Visual Database Systems 3: Visual Information Management (Proc. 3rd IFIP 2.6 Working Conference on Visual Database Systems, Lausanne, Switzerland, March 1995) pp. 360-371, S. Spaccapietra and R. Jain (Eds) Chapman & Hall, 1995.

Perceptual Methods in Information Visualisation

Nick Kirtley s1471651
Robert Vrooland s1494821
Computer Science Department, Rijkuniversiteit Groningen

Abstract

Arranging information in a way that a human can understand is of great importance. Perceptual methods in information visualization helps a user to understand complex data by arranging it in a specific way.

The type of visualization depends on a number of factors like user type, amount of data and the complexity of the data. We will discuss several methods for the visualization of information. These methods are needed in order to maintain the optimal flow of information to any user. These methods can build a system such as a H.U.D. (Heads Up Display) or a system that can extract data from large databases with information of high dimensionality.

1. Introduction

In this digital age information is everywhere. Infact there is often so much information that it is difficult to understand what the information means. One of the easiest and most commonly used ways of interpreting and relaying information is through visual means. With the increase in the amount of information it is more difficult to decide how to visualize data. Realizing how to display information is extremely important because it can affect the way that the information is interpreted. Therefore the same data can be visually represented in different ways and can cause confusion between users that try to interpret that data.

This review paper focuses on serveral methods of displaying complex information in a way that important information is retained and can be interpreted by the user. The word 'important' can of course be interpreted in different ways to different people.

2. Human perception

From the many methods available to us for visualisation, it stands out that most of them are based on the priciples of human perception. Several problems caused by human perception when trying to interpret visual data are shown in this paper. To name a few examples. When trying to increase the intensity of a visual dimension, the magnitude of the same increase can be perseived differently, like in brightness or height.

When using color in a graph, using a continuous colormap could cause people to not clearly see the diffrence of data in a region. Different shades of yellow can be hard to tell apart.
Data can be visualised in several ways. Sometimes 2 or more methods are combined to give the user more information on the same graph. This can cause problems if the different layers of the graph interact with each other.

Recent studies and experiments have shown that when we look at something, the following classes are preattentively processed:

- Form
  - line orientation, length, width and colinearity
  - size
  - curvature
  - spatial grouping
  - added masks
  - numerosity (numeber of items)

- Color,hue and intensity

- Motion, flicker and direction

- Spatial position
  - 2D position
  - stereoscopic depth
  - convexity/concavity (shape form shading)

Using these characteristics of the human perception will allow us to create better visual representations of data. Building a rule-set on these characteristics will allow us to for example make it easier to understand the data, focus on important bits of data or being able to define a clear border between data.

3. Visualization of one surface overlying another.

It is often important to visualize multiple layers of information. This can be the case with 3D models where not only the outside (of the model) is of importance. For example the ocean can be modelled with various temperatures represented as different layers. In such a case it is important to be able to differentiate between the layers. Both surfaces must be textured and the top layer must be partially transparent in order for them to be differentiated. The choice of textures for both surfaces is not an easy one because it is dependant on a number of parameters. The solution is a three step process.

The first step involves developing a parameterization of the problem. This should result in a vector of parameters representing the visualization of the data. The number of parameters is likely to be very large because it is for visualization purposes and that tends to contain a large amount of information.

The vector is put in a genome so that step two can use a genetic algorithm where selection is based on user preferences. A genetic algorithm is used because of the high dimensionality of the parameter space and because it is able to rapidly explore promising regions. Each generation of results should be evaluated by a human so the selection process progresses in the 'right' direction.

Step three is needed to characterize results. This is based on what is important to the user. If a specific solution is needed then the genetic selection process can be stopped as soon as a satisfactory solution is found. If a more general solution is needed then a large set of solutions can be analyzed.

4. How humans interpret visualization and how to use this knowledge to enhance visualization

As stated before, human perception is important when creating visual representations of data. To get a closer look at this phenomenon we will look at two examples of systems. These visual systems build completly around the idea of human perception.

First theres the Stereoscopic Field Analyzer (SFA). This system allowes for effective conveying of information from large multidimentional datasets. This system can represent data in a 3D field filled with 2D glyphs. Every glyph in this field can have up to 9 variables. Each of these variables are directly connected to the variables that the human perception is good in identifying. These attributes range from location, size, shape and more.

The user this system can freely move around in the field looking at every possible angle. When looking at one of these fields everyone will quickly be able to read the data correctly, and be able to spot the most interesting regions.

This immeadiatly brings us to the second system. We will now look at a volume illustration techniques. This system is used to enhance excisting illustrations in order to clarify the data it represents. Before using a system like this we ask ourselves 3 questions:

1. What information should be displayed?
2. What display techniques should be used?
3. How should the display technique be implemented.

The 3 most common illustration techniques are: feature, depth/orientation and regional enhancements.

By using one of these techniques we are able to enhance an excisting illustration, and highlight the regeons of interest.

5. Building a perceptual visualisation architecture

Visualising large multidimensional datasets can be a hard task. It is often a case of having too much information so the hard task is determining what information is important to visualize.Size of a dataset is based on three characteristics:

- The number of elements in the dataset
- The number of attributes or dimensions
- The range of values

The first task of visualizing a large dataset is managing the data. This can be done using data mining classification algorithms and has the following results:

- Identifies dependencies
- Estimate missing or correct erroneous values
- Compress size and dimensionality of a dataset

After the data has been managed it is necessary to visualize the data. This consists of mapping data attributes to visual features such as colour, intensity and texture. When visualizing the information it is important to take into account how humans preattentively interpret information. This has the advantage that visual analysis is rapid and accurate, output is insensitive to display size and avoiding bad visual feature combinations.

Perceptual texture elements, or pexels, can be used to display 3D information and more importantly can be interpretted preattentively. A pexel is basically a rectangle at a certain position in a 3D map. The size of a pexel can vary by length indicating a certain value and the density of all the pexels has meaning as well. Regularity of pexels can also quickly be observed by people.

Colour can also be used to represent attributes in multidimensional datasets. It is important to be able to quickly differentiate between colours.

6. An architecture for perceptual rules into the visualisation proces using metadata

Representing data is difficult. Presenting data in a visual way is not the problem. Its how people interped the data from the visual representation. There is a basic rule that is the cause of this problem: ?Mathematically identical != Perceptually identical?.
The human perceptual and cognitive mechanisms are very important when trying to visualize data.
The same data, like the temerature of an area, can be represented in different ways. Ways like using grayscale, using contours or the height of the graph creating a landscape. All these methods can produce a diffrent perception of the same data. Several methods have been though up to combat this problem. One of them is the ?architecture for rule based visualisation?.
This method in short works on 2 principles:

1. casting the visualisation operations to higher levels of representation of the
data, called metadata.
2. use a given set of visualisation rules based on priciples of human vision,
cognition and color theory.

These higher levels of representation called metadata is composed of data such as:
image statistics, if data is discrete ot continuous of spacial coherence. These kinds of data can help when trying to create a structure for a visual operation. The meadata can also be used in the visualisation itself. For example, it can be better to use a standard deviation instead of the entire array for visual operations.

Now that we have established a structure for the visual data, we can start adding the data. We do this with the second rule. Using the prinicples of human vision already previously discussed, we start filling in the data.
We have choosen the proper format like using color, contours, heightmaps or any combination, and put it in the structure created by the metadata.

Now we have a visual representation where the dataset is faithfully represented.

Conclusion

An underlying theme found in all of the papers is the psychology of human visual perception. It is important to understand this phychology and use it so that information is perceived correctly. All of the papers use the theory of human perception as a basis for creating rule-based systems. There seems to be little or no diffrence in the way the human perception is used in all of the systems.

All of these systems are not the same though. They all use the same basis for perception but then have their own technical problem. Each of these systems will then solve the different problems in their own different way. So in a technical sence they do not overlap or contradict each other.

References

1. Rogowitz B.E. , Treinish L.A. , An Architecture for rule-based visualisation
2. Ebert D.S. , Extending visualisation to Perceptualization: The importandce of perception in effective communication of information
3. Wattenberg M, Fisher D, A model of multi-scale perceptual organisation in information graphics
4. House D, Ware C, A method for the perceptual optimization of complex visualisations
5. Healey C.G. , Building a perceptial visualisation architecture

Combining perception-based visualization techniques

Han Stiekema

Institute for mathematics and computer science (IWI),
Rijks*universiteit* Groningen, The Netherlands
H.Stiekema@student.rug.nl

Abstract -- In this article, four papers on perceptual visualization techniques will be reviewed. This paper will cover two things: first, the described techniques are briefly explained and their applicability is discussed. Second, a combination of the techniques is recommended that might result in an even better visualization of the data. An example from one of the papers will be extended to give an indication of where this combination of techniques could work.

1. Introduction

Making a good visualization of (scientific) data is a very difficult task, especially when this data consists of several dimensions. Visualizing for instance ten data attributes understandably in a 2D, or even a 3D image seems like an impossible task, especially when you also want to visualize the relationships between the data attributes. Many studies have been conducted to find out how to make a good visualization that clearly shows the data and allows interpretation of more complex structures. Some of these studies try to use characteristics of the human visual system. In these studies, characteristics of the human visual systems were investigated to see how they can be used to make visualization techniques more effective. Four papers ([1], [2], [3] and [4]) that discuss a visualization technique using the human visual system in some way will be discussed in this article and the applicability of the discussed techniques will be investigated.

Under the right circumstances, combining some of these techniques might result in even better results. Therefore, a real life example, taken from one of the articles, will be taken one step further to possibly give a better visualization of the data than already had been achieved.

2. Review of visualization techniques

The visualization techniques that will be discussed in this article have been taken from Rogowitz and Treinish [1], House and Ware [2], Wattenberg and Fisher [3] and Healey [4]. Only a brief description of the techniques will be given, for further details (on formulas, algorithms or statistical results of a study for example), see the original documents. The description of each technique will be followed by a short note on the applicability of the technique.

2.1 Rule based visualization

In many cases, it is possible to visualize the same data in different ways. This could lead to wrong interpretations if the data is represented in a less visualization-favorable way. Also, visualization of one data attribute might perceptually interact with the visualization of another attribute. For example, color and shape might interact, because of shading effects. (See also [5], where Kim et. al. discuss the influence of texture on the perception of shape).

To solve this kind of problems, Rogowitz and Treinish propose a rule-based approach to visualization in [1]. An important part of this approach is that metadata should be added that gives extra information about the data. This metadata could contain information about, for example, the interval that indicates possible values of a certain data attribute, or the dimension of the data. This metadata can either be derived from the data, or it can be obtained by interaction with the user.

This metadata can now be combined with perceptual knowledge to create rules that indicate how a certain data attribute can be visualized effectively. Ideally, the system should be able to determine what type of visualization to use with what type of data attribute. For instance, data about wind will usually contain some information

about direction and strength. If such information is available in the metadata, a visualization rule could say that this data should be visualized by using arrows.

This method could be effective in certain situations. However, it might be hard to determine the rules that this approach is based on. Especially, coming up with effective rules that do not interact can be quite difficult in multidimensional data. The method is somewhat abstract and could prove to be difficult to generalize. Of course, this does not imply that it is futile to use this type of visualization structure.

2.2 Overlaying surfaces, a genetic approach

House and Ware try to tackle visualizations of overlaying surfaces. In [2], they focus on the problem of showing two overlaying surfaces in such a way that a user can see both surfaces. To solve this problem, they use a genetic algorithm to reach visualizations that allow a user to clearly see the different surfaces that are being displayed. The described algorithm uses vectors containing texture parameters for both surfaces. These vectors are the genes that represent texture in the texture space (which is too large to try all possible textures). A generation contains several of these genes. These genes are initialized randomly. The algorithm generates new generations by performing two genetic steps (i.e., pair wise cross-over and mutation), and then asks the user to evaluate the display that is formed by each specific gene. After a while the algorithm has reached a somewhat uniform generation in which the display is optimized. The algorithm could be restarted with new initial gene values to get a different optimized display.
After running the genetic algorithm, a clustering algorithm is used to find groups of displays that use similar parameters (two solutions are clustered when their parameters have a Euclidian distance smaller than a certain threshold). According to House and Ware, this approach usually results in two clusters, one big cluster and a smaller one.

House and Ware have done a user study on this approach with 5 subjects, each completing 3 trials. Generally, after about 20 generations, reasonable solutions were achieved, where good solutions were common. Performing one trial took the subjects about two hours.

Even though this approach might result in acceptable results for this rather complicated problem, execution time is simply too slow and takes too much effort to be a real visualization technique. The described method might be considered an 'escape' method, used when other visualization techniques are unable to provide acceptable results. However, the visualizations that have been found could be examined to see why they are better than, for example, the random initial visualizations. In this respect, their research might be a starting point for a user study to find out what the important elements of a successful visualization are. House and Ware give some suggestions of what type of texture they believe is more successful.

2.3 Multi-scale visualization technique

In [3], Wattenberg and Fisher discuss a multi-scale model that is capable of finding structures in grayscale images. As an example, they use the so called "Dr. Seuss" image (see figure 1) that gives an indication of how information in an image is grouped.

Fig. 1. Four different scales of the Dr. Seuss image (image taken from [3])

Wattenberg and Fisher model this idea of scales mathematically using Gaussians. They describe an algorithm that uses this definition to link different scales together. This algorithm finds elements in the image that are grouped at a high scale. These high-scale linked groups indicate that the specific region belongs together in a way. In the Dr. Seuss image, these are, for example, the lines in the image. There are clear spaces between the characters and the words, but the algorithm finds that at a certain scale, the words belong together and form a line (which in turn forms the paragraph, together with the other lines).

Given this algorithm, we can analyze images to see whether parts that the algorithm links together actually belong together. If certain elements in an image that have no real relation to each other and should not be considered one element (for example, grid lines and a graph drawn in this grid), are linked by the algorithm, this gives an indication that the visual system has difficulties in separating the individual elements. In this example, the visual system would find it hard to distinguish between the graph and the grid lines if the graph and the grid lines would be linked together by the algorithm. The illustration below demonstrates this example.
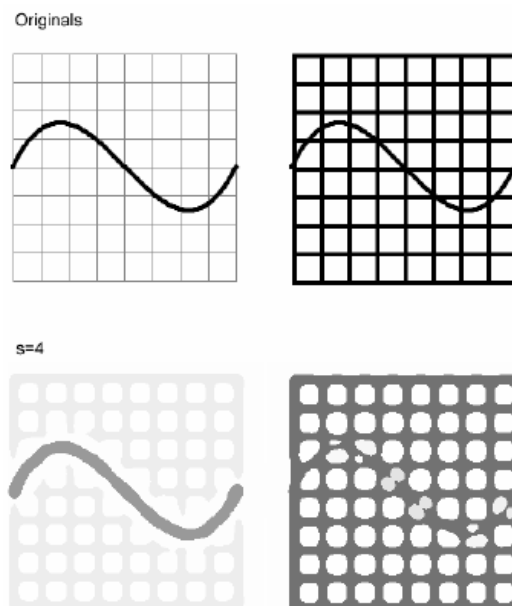
Fig. 2. Two versions of the same graph on grid with thin lines (left) and thick lines (right). The thin gridlines do not pose a problem, whereas on a higher scale, the thick grid lines merge with the graph. This indicates that the human vision system tends to see the graph and the thick grid as one object. (image taken from [3])

Because this algorithm gives a clear indication of where things might go wrong, we are able to improve images. In the grid line example, thinner or dotted grid lines might result in a better image.

Even though this method does not generate good-quality images, it could be very useful in evaluating the quality of images. A shortcoming of this method is that it only works on grayscale images, but the method might be extended for the use of color. Wattenberg and Fisher do point out that this method has not been validated, but it seems to be quite a promising one.

2.4 Perceptual visualization architecture

In [4], Healey describes a method for visualizing large, multidimensional datasets that uses characteristics of the low-level human vision system. The method he describes is a step-by-step approach that gives a visualization that shows the most important data in a way that it is easy to see the data.

The first step in this approach is to analyze the data with data mining techniques that are capable of giving a reasonable estimation of missing data, and that indicate which of the data attributes are the most significant. By only displaying the most significant data attributes, there is obviously a loss of data, but this loss is minimized; the least important things are ignored. To provide a reasonably understandable visualization, only a limited number of attributes can be visualized effectively. If necessary, more attributes can be included by making several images. This could for example work quite well if another important attribute is time. Each image could then show the data, that varies over time (a different image for each time step), which is quite intuitive in many cases.

The actual visualization of the attributes that are selected in the previous step is the result of a rather extensive study to find out how both texture and color can be used for data representation. First, a brief overview of the texture study.
Perceptual texture elements (pexels) are used that contain little 'towers'. Pexels can vary in the height of the towers (the height of a pexel), the amount of towers contained by a single pexel (density) and the arrangement of the towers (regularity).
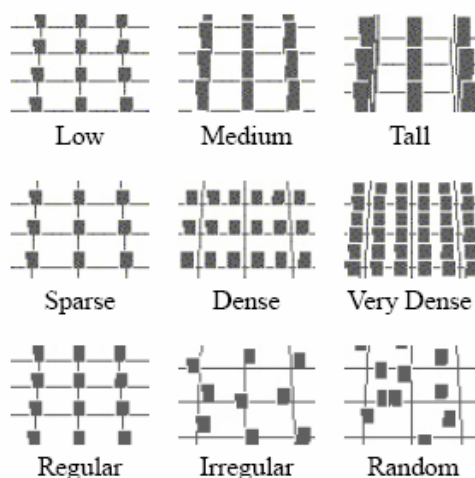
Fig. 3. Different types of pexels used in the experiments (image taken from [4])

The study described in [4] showed that height and density of the pexels are characteristics that are easily recognized by the human visual system, e.g., in a grid of low pexels, a tall pexel is easy to spot. Regularity is considered to be very difficult to judge, and is therefore a bad way to visualize an attribute. It should be remarked that height influences the ability of identifying density, taller pexels make it more difficult to spot denser pexels. This led to Healey's choice of using pexel height to represent the more significant data attribute and density for other attributes.

As mentioned, color was studied in addition to the texture. This study showed that for a good visualization the amount of colors that is used should be limited to about seven and that these colors should have a certain distance between them. It helps if each color is selected from a different category (for example red, blue and yellow). Colors that are a combination between for example green and yellow are sometimes hard to name, i.e.,

users tend to give both green-yellow combinations and pure green the name 'green'. Usually, users can see the difference, but the further the colors are apart, the easier it gets.

Healey has used this approach to visualize plankton densities quite effectively. A disadvantage of this technique is that some data is ignored, but in many cases, this is acceptable. Another disadvantage is that absolute values are very hard to distinguish. For example, you cannot tell exactly how dense a certain pexel is by just looking at the image, and therefore it is very difficult to get an idea of the value that the density represents. However, this method is very suitable when differences are more important than exact values (e.g., 'this pexel is twice as high, so the wind speed will be twice as high').

3. Visualization techniques combined

Not each of the techniques mentioned above is equally suitable for any given problem. Here, an outline will be given of how these techniques could possibly be combined to form a technique capable of better visualizations and the visualization of more complex data. Note that no studies on this combination have been conducted, and that this combination is purely an attempt based on what is described in the discussed articles.

As a basis, the plankton density example from Healey [4] will be used. The image below is the visualization that has been created by the method Healey has described. In this image, color indicates plankton density, height represents current strength and texture density represents sea surface temperature.
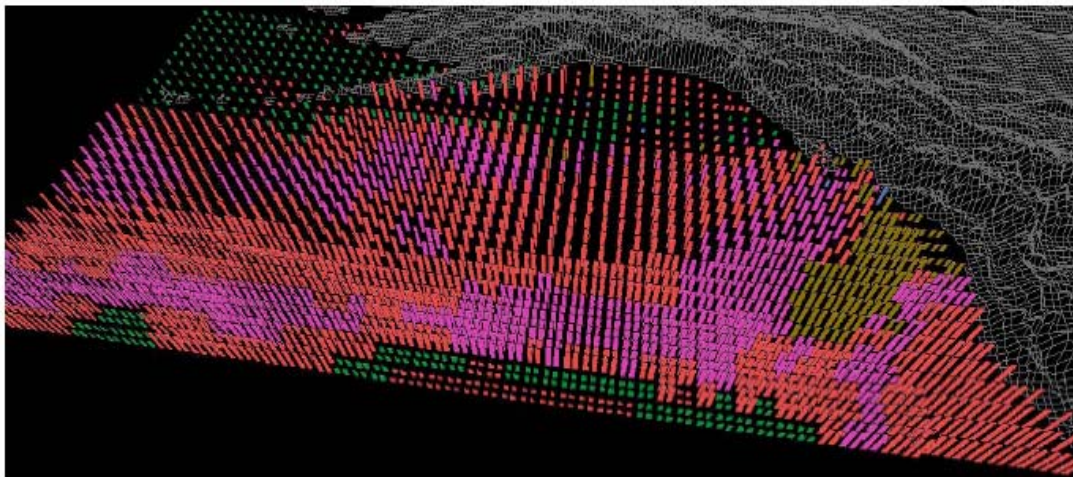
Fig. 4. Visualization of the oceanography datasets, color used to represent plankton density (blue, green, brown, red, and purple represent lowest to highest densities), height used to represent current strength, texture density used to represent sea surface temperature. All in June, 1956. (image taken from [4])

Implicitly, Healey has already applied the rule based approach from [1], by deciding to use pexel height to represent current strength and which color should represent what level of plankton density etc. These choices could be included in the architecture, by adding visualization rules to Healey's method. Rules can be used to decide which data attributes to represent, how to represent certain types of attributes and to decide which color should represent what kind of information (for example, use blue for low values and red for high values). Another rule might be to always use pexel height for the most significant data attribute and density for the least significant attribute.
This is just a simple example of a combination of techniques, which could make a method more general. Of course, more complex and more detailed rules could be used.

For really complex data that is very hard to represent nicely, the overlaying surfaces method from [2] could be used to add an extra 'layer' of information on top of the data that is already available. By making both layers visible with the technique from [2], it might be possible to display more dimensions than is possible with the original method. However, this kind of visualizations might become very hard to interpret and the extra layer should be expected to interfere with other information. This added layer of information is therefore not directly recommended, but perhaps a method for successfully overlaying two surfaces, while still being able to see all the information contained in both surfaces, will be developed from the method in [2]. Perhaps, in combination with the rule based approach, it can become useful.

The multi-scale approach from [3] is hard to fit in this example. First of all, because this method is grayscale based, but as mentioned in [3], the method might be extended for color. Second, this method is based on two dimensional images, and does not have real value in a three dimensional image. These are disadvantages inherent in the method, but that does not imply that this method is useless. In two dimensional cases, it could very well have evaluational value to make sure the visualization is interpreted in the right way by the human visual system.

4. Conclusion

In this article, four visualization techniques have been discussed. The discussed methods all have some way in which they could be applied, but in general, the rule based approach in [1] and the visualization architecture from [4] are most likely to give a way to create a good visualization. Both of these techniques can also be combined to get a more general technique.

The genetic overlaying surfaces technique mentioned in [2] might be useful to create clearly visible overlaying surfaces, but it is a somewhat slow method that requires a little too much effort to be a good automatic visualization method. On the other hand, it will probably give information to find out effective visualization textures for this problem, which could lead to a real automatic visualization tool. It might even be combined with the rule based approach, as described in [1].

The multi-scale approach from [3] probably has the highest value as an evaluation technique, to get an indication as to how the human visual system interprets structures in an image, rather than a visualization technique.

5. References

[1]. An architecture for Rule-Based Visualization. B.E. Rogowitz and L.A Treinish, IBM, 1993;
[2]. A model of multi-scale perceptual organization in information graphics. M. Wattenberg and D. Fisher, IBM and University of California, 2003;
[3]. A method for the perceptual optimization of complex visualizations. D. House and C. Ware, Texas A&M University and University of New Hampshire, 2002;
[4]. Building a perceptual visualisation architecture. C.G. Healey, North Carolina State University, 2000;
[5]. Showing shape with texture – two directions seem better than one, S. Kim et. al. Univ. of Minnesota, 2003.

Visual Textures for Displaying Multidimensional Datasets

Mai Ho and Gert-Jan de Vries

Abstract. A technique for displaying multidimensional datasets is the use of visual textures. Based on outcomes of psychophysical research and experiments, methods have been developed to generate visual textures. This paper discusses four of those methods: natural textures, using patterns found in nature; oriented sliver textures, which use oriented stripes and luminance; textures with paper strips, with varying height, density and regularity; and OSC textures (with varying orientation, scale and contrast) using Gabor functions.

1 Introduction

Information visualization is a field of science in which research is being done on how to display data in a form that allows users to rapidly and accurately explore a dataset. A typical graphic information display, such as a computer screen, provides two spatial and three color dimensions. For displaying data with a higher dimensionality, display channels like perspective, motion and visual texture can be used. Although visual texture uses the dimensions 'reserved' for displaying shape and color, human vision can still easily distinguish the general color and shape of an object from its texture, like for example a woolen jersey. Therefore the use of texture, in most cases, does not largely affect the perception of color and shape.

The data we consider consists of multidimensional data points spread along a grid on a surface. Therefore each grid area contains a vector of, say, $N > 1$ dimensions that represent the values of N attributes (for example measurements) connected to that location on the surface. Examples of scientific areas in which this kind of data can be found, are meteorology (temperature, wind speed, air pressure over Europe), physics (pressure, temperature along the surface of an object), chemistry (concentration of chemical elements on a surface), etc.

The key in displaying multidimensional data is to optimally use the available display dimensions to enable the viewer to observe patterns in the data, thereby not causing interference between different data dimensions. In this paper we give an overview of four techniques that use visual texture for this purpose.

2 Previous work

Texture has been studied extensively in the computer vision, computer graphics and cognitive psychology communities. Results from cognitive psychology have

shown that height, density and regularity are detected by the low-level human visual system [2]. The latter two have been identified in the computer vision literature as being important for performing texture segmentation and classification [3]. Healey and Enns [1] designed a perceptual technique for visualization of multidimensional datasets that uses all three dimensions.

A similar technique using oriented sliver textures was introduced by Weigle et al. [4]. This technique extends EXVIS [5] by allowing a viewer to estimate relative values within an individual field, while still producing the characteristic textures needed to highlight interactions between different fields.

The graphic design community has long held that perfectly regular synthetic textures on a flat plane are discomforting to the eye and annoying to look at. The use of natural textures, resembling those from photographic images [6], is discussed by Interrante [7].

A mathematical model based on the Gabor function has been studied by different people [8]. Ware and Knight [9] studied the suitability of a simplified class of Gabor functions as primitives in the generation of texture.

3 Existing methods

In the previous section different techniques were mentioned that use visual texture to visualize multidimensional datasets. Here we discuss four of those techniques.

3.1 Perceptual texture elements

Description This technique uses perceptual texture elements termed pexels to represent each data element. Attribute values encoded in the data element are used to vary the appearance of a corresponding pexel. The pexels are built by controlling three separate texture dimensions: height, density and regularity. They are three-dimensional and look like a collection of paper strips that "sit up" on the underlying surface. See Figure 1 for an example.

Height and density both have a natural ordering that can easily be used to order attribute values. Ordering regularity is more complex.

Experiments and results Controlled experiments have been conducted to measure the effectiveness of textures built out of pexels. The most important results:

- Taller pexels can be identified at preattentive exposure durations (within 150 msec.) with very high accuracy.
- Shorter, denser and sparser pexels are more difficult to identify than taller pexels, although good results are possible at 150 and 450 msec.
- Irregular pexels are difficult to identify and regular pexels cannot be accurately identified.
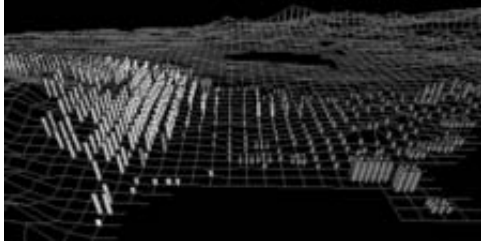
Fig. 1. A map of North America, pexels represent areas of high cultivation, height mapped to level of cultivation, density mapped to ground type, greyscale mapped to vegetation type.
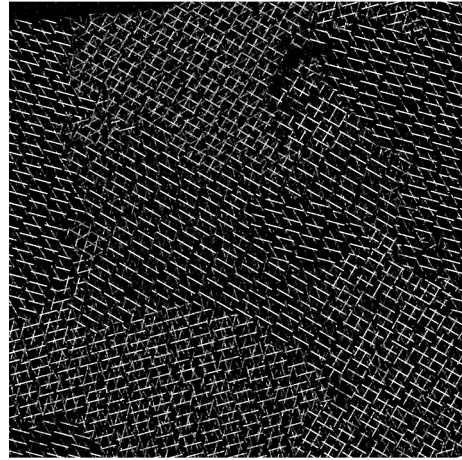
Fig. 2. A sliver texture displaying the presence of several chemical elements, like silicon (165°) and oxygen (120°), on a surface.

Since some type of regularity is often used as one of the primary texture dimensions in texture segmentation and classification algorithms, the poor experimental results for regularity were unexpected. Regularity targets can be made easier to identify by increasing the size of the target patch or by increasing every pexel's density. Both methods involve tradeoffs in terms of the kind of datasets we can visualize or in the number of attributes our pexels can encode. Therefore, attributes with low importance are normally being displayed using regularity.

Future Combination of features like hue, intensity, orientation, motion and isocontours with perceptual textures are suggested, to increase the number of data values that can be displayed simultaneously. Another possibility is to apply various orientations to a pexel.

3.2 Oriented sliver textures

Description This technique uses small line segments termed slivers to represent data. Chosen constant orientations (angles over a scale of 0°-180° are used, since the slivers are rotationally symmetric) are assigned to attributes A_i. Luminance changes with the value of attribute A_i where the minimum value of A_i is depicted as black, the maximum value of A_i is depicted as white. The values in between are distributed along the grey value scale using a perceptually based luminance scale to compensate for the visual system's response to luminance (which is approximately logarithmic).

The actual texture is created by placing the orientated slivers onto a grid. To remove possible visual patterns created by the regularity of the grid, each sliver is moved by a small random distance in a random direction. An example is given in Figure 2.

Experiments and Results Conducted experiments have led to the following results:

- Orientations of the sliver should differ at least 15° to be easily distinguished. This led to the insight that at most 12 (180° contains 12 angles of 15°) attributes can be displayed. Note that 0° and 180° coincide because of symmetry of the slivers.
- The cardinal directions 0° and 90° form a special case: they can be distinguished from slivers rotated at least 5° (instead of 15°). Further, they form a good background patch, but not a good foreground patch. Therefore these orientations should be treated with care and need to be assigned to attributes which are present in large quantities (so they can act as kind of a background). Keeping this in mind 14 different orientations can be chosen, for example [0 10 25 40 55 70 85 90 95 110 125 140 155 170].
- When orientation varies over 45°-90°, subjects tend to take slightly more time to give their answers than in the 0°-45° experiment. For static (or slowly changing) data the researchers concluded this result was due to differences between subjects. However, for the use of oriented sliver textures in real-time applications, more research has to be done to verify if this difference is generally the case.

Future There has been no research to the effect of placing multiple slivers on the same location. This is the case when regions with presence of different attributes overlap. Of course overlap of large numbers of orientated slivers limits the ability to observe the actual presence of attributes.

A large advantage of this method of texture creation is that it is relatively efficient in terms of computational power and can be executed in parallel (slivers with different orientations can be determined in parallel), which is certainly a pre for real-time applications.

This texture has the potential to be combined with other texture dimensions like the size or density. Furthermore future research will be done to use 3D orientation in textures.

3.3 Natural textures

Description This technique uses texture patterns found in nature as elemental primitives for data representation. To use this technique, a texture palette is needed. An example of a small potential texture palette is shown in Figure 3. It is important to have a perceptually linear texture space to quantify the perceptual distances between individual texture patterns.

Experiments and Results Interrante [7] only describes approaches to estimate the magnitude of the perceived distance due to the differences along each of the feature dimensions. No actual experiments have been conducted to measure how effectively information can be visualized using natural textures.
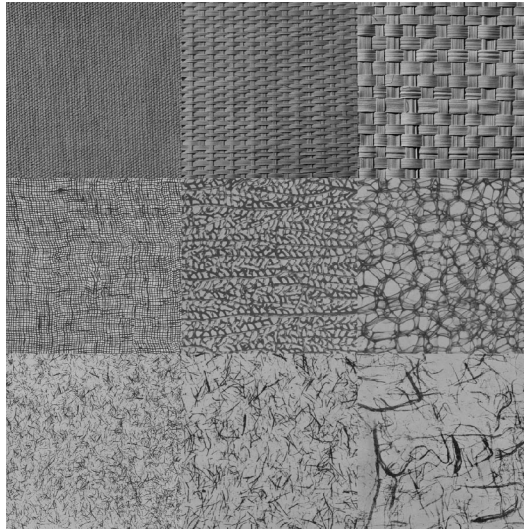
Fig. 3. A small potential texture palette. Scale increases along the horizontal axis and regularity increases along the vertical axis.

Future Some important issues that have been overlooked in the past are the following:

– What is the effect of uncontrolled-for influences of higher level processes, when identifying the features according to which people tend to classify texture patterns?
– The question of whether to control for rotation, scale, luminance and contrast variance among texture samples when seeking insight into the perceptual groupings of texture images.

A possibility is to consider textures that exhibit relief. This possibility introduces the issue of how to properly deal with the lighting consistency problems that will inevitably arise.

3.4 OSC (Gabor) textures

Description Research in electrophysiology and psychophysiology [10] indicates that the human brain contains large arrays of neurons, which filter for orientation and size and are sensitive to elongated shapes. Because of the large similarity between these elongated shapes and Gabor functions, Ware and Knight [9] pose the idea to use Gabor functions to create textures.

A Gabor function is a complex valued function, consisting of the product of a Gaussian with a complex exponential and, in 2D defined by

$$e^{-\frac{x^2+y^2}{2}} e^{if_0 y}$$

with f_0 specifying a frequency. Available for the use in textures are both the real part (cosine Gabor) and the imaginary part (sine Gabor). Illustrations of these Gabor functions are depicted in Figure 4.
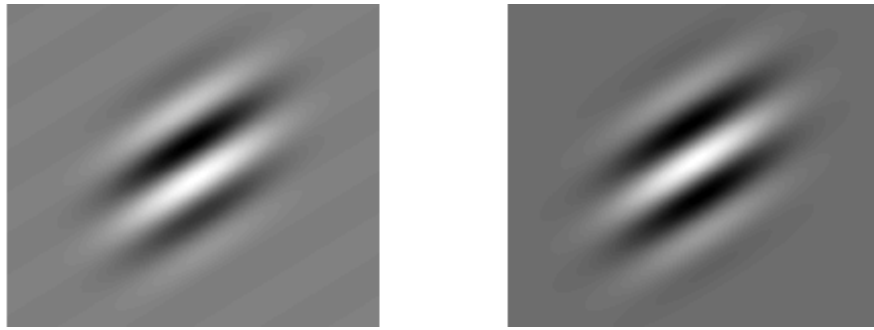
Fig. 4. A sine Gabor (left) and a cosine Gabor (right), both with frequency $f_0 = 4$.

To form the so called OSC (Orientation, Size, and Contrast) texture, standard scaling and rotational operations can be used on the Gabor function primitives and the contrast can be varied. The orientation can be scaled linearly over the range $0°$ to $180°$ since the Gabor patterns are twofold symmetric (so $180°$ to $360°$ coincides). Size (i.e., the frequency of the Gabor function) has to be scaled over a range of 2 to 16 cycles per degree of eyesight, which are limits of the human vision [11]. An exponential scale is being used. Finally contrast is scaled exponentially.

Experiments and Results Ware and Knight [9] have not conducted actual experiments on how effective OSC textures based on Gabor functions are, i.e., how good patterns can be distinguished in these textures.

Future The example textures that are presented by Ware and Knight [9] are vague due to the character of Gabor functions which seem to make detecting patterns harder. The exact influence of this vagueness should therefore be researched.

Furthermore research still has to be done in how humans associate some kind of 'semantics' to textures. Somehow humans seem to prefer certain texture dimensions to be used for displaying certain data dimensions. For example it seems more effective to display amounts of energy using texture contrast instead of orientation.

4   Conclusion and future work

We studied four existing techniques that use visual texture for information visualization. These techniques, using different textual dimensions, form a base for techniques that allow us to display datasets of even higher dimensionality.

With the sliver textures, other shapes could be used instead of line segments. An interesting research subject would be what shapes are suitable. It can easily be seen that a higher order of rotational symmetry of the shapes limits the freedom of the rotational dimension. Another extension on the sliver textures is

to vary the size and density. In this technique areas of high density can become too dense to be able to distinguish the other texture dimensions. Zooming in on the surface could solve this, but can limit the overall view. This trade-off can be another interesting subject for research.

The method using perceptual texture elements can be extended with orientation. The exact orientation can only be identified when the paper strip is limited to a plane. To successfully visualize data, you do not want paper strips to touch or overlap each other. So when using this new method, we first need to know which orientations can be used. Another variation is to combine pexels with natural or Gabor textures, resulting in decorated pexels. The decoration can be applied to the pexels themselves, to the underlying surface or both. However, the decorations may be distracting, making it more difficult to identify tall, short, dense, sparse, regular or irregular pexels. Also, in case decorated paper strips are used, the texture patterns may be small and not close together. This makes it harder to identify perceptual distances between texture patterns.

The proposed changes and additions of dimensions to existing techniques may interfere with the texture dimensions already present, limiting (and possibly decreasing, in the case of dimensions already present) the number of data values that can be displayed per dimension. Controlled experiments are needed to provide the required insight to make optimal use of these new methods.

References

1. Christopher G. Healey and James T. Enns: Building Perceptual Textures to Visualize Multidimensional Datasets. IEEE Visualization (1998) 111–118
2. J.M. Wolfe: Guided Search 2.0: A revised model of visual search. Psychonomic Bulletin & Review **1(2)** (1994) 202–238
3. A. Ravishankar Rao and Gerald L. Lohse: Identifying High Level Features of Texture Perception. CVGIP: Graphical Model and Image Processing **55(3)** (1993) 218–233
4. C. Weigle and W. Emigh and G. Liu and R. Taylor and J. Enns and C. Healey: Oriented sliver textures: A technique for local value estimation of multiple scalar fields. A technique for local value estimation of multiple scalar fields. In Graphics Interface (2000) 163–170
5. Georges Grinstein and Ronald M. Pickett and Marian G. Williams: EXVIS: An exploratory data visualization environment. Graphics Interface '89 (1989) 254–261
6. Phil Brodatz: Textures: A Photographic Album for Artists and Designers. Dover Publications (1966)
7. Victoria Interrante: Harnessing Natural Textures for Multivariate Visualization. IEEE Computer Graphics and Applications (2000) 6–11
8. M. Porat and Y.Y. Zeevi: Localized texture processing in vision: Analysis and synthesis in Gaborian space. IEEE Trans. on Biomedical Eng. **36(1)** (1989) 115–129
9. Colin Ware and William Knight: Using Visual Texture for Information Display. ACM Transactions on Graphics **14(1)** (1995) 3–20
10. D.H. Hubel and T.N. Wiesel: Receptive Fields and Functional Architecture of Monkey Striate Cortex. J.Physiol. **195** (1968) 215–243
11. Colin Ware and William Knight: Orderable Dimensions of Visual Texture for Data Display: Orientation, Size and Contrast. Proceedings of CHI'92. (1992) 203–210

A review of three Multi-Layer Visualization methods

Joris Lops and Mickeal Verschoor

Rijksuniversiteit Groningen, Department of Computing Science, Blauwborgje 3, 9747 AC Groningen, The Netherlands,

Abstract. Visual perception of humans plays an important role when graphical visualization of data is considered. Several psychophysical experiments showed how the low-level human visual system perceives several visual features. These features include orientation, luminance, blur and texture. When complex multidimensional datasets have to be visualized, problems may occur when the data has to be interpreted. In this paper we review three methods for visualizing multidimensional data as a set of different data layers. The final image contains all data of each layer.

1 Introduction

The user plays an important factor in visualization. It does matter how the algorithm performs, for example memory usage or/and speed, but it must communicate the data in a clear picture so that a user can interpret it without difficulties. Visualization is therefore a study in which psychological aspects are very important.

Multidimensional datasets are hard to visualize and for viewers it is hard to interpret the information stored in such images. Multidimensional datasets can be found in several disciplines of science, such as astronomy, physics and meteorology. A common way to visualize multidimensional datasets is by using a number of scatter plots. The problem with such visualizations is that the coherency of the data dimensions is lost. This can be solved by visualizing al data dimensions in one image. When we do this, perception of the human visual system plays an important role in order to distinguish different dimensions from each other.

The problems that are encountered when multidimensional data are visualized are that a viewer needs the coherency between several data dimensions. A better approach than scatter plots is using several layers of representations, where each layer corresponds with some dimension in the dataset. Each dimension in the dataset is spatial dependent from each other in such way that we can easily stack the different layers into one final image, where each data dimension is present. The problem is how we can distinguish several layers in the final image.

In this paper we review three papers about visualizing multidimensional data in a layered approach. Each paper presents some research methods and results within the context of visualizing layered datasets.

2   Review of three methods

First we review the paper of C. Weigle et al.[1], which uses sliver textures in combination with orientation and luminance to visualize the data. Next we review the work of R. Kosara et al. [2], where the application of depth of field is discussed in order to emphasize some parts of the data. At last we review the results of D. House and C. Ware [3], which describes a method for texturing surfaces in order to make two surfaces distinguishable by humans.

2.1   Oriented sliver textures

C. Weigle et al.[1] describes a texture generation technique that combines orientation and luminance to support the simultaneous display of multiple overlapping scalar fields. In order to display a scalar field, oriented sliver textures can be used. Figure 1c gives an impression of an oriented sliver texture. The values of the scalar field are mapped to a luminance texture (grayscale swatches). Figure 1b shows the corresponding luminance texture. The luminance map and the oriented sliver texture are combined to produce the final sliver layer, which is shown in figure 1d. Each dimension in the dataset is represented by a sliver layer. When multiple sliver layers are displayed, the layers are rotated differently to make them distinguishable from each other. Figure 1 gives an impression of a visualization of two scalar fields using oriented sliver textures. The image allows users to locate values in each individual dataset and the relation between the values. To apply oriented sliver textures efficiently, it must be know when the visual system can distinguish several sliver textures with different orientations. I.e. how many distinguishable sliver layers can be shown in the final image?

To answer this question the minimal angle between two sliver textures must be known. The researchers found the minimal angle between slivers by measuring the response time and error rate. In the experiment several tests were taken with a number of subjects. Each subject had to identify one ore more different slivers, if present in the image. An example of a test image can be found in [1]. They measured that if the angles between background slivers and target slivers was smaller than 15 degrees, the subjects took more time to select target slivers and they made more mistakes. For background slivers oriented at 0 and 90 degrees they found that the minimum angle between background and target slivers could be reduced to 5 degrees. The total number of different oriented slivers in one image can be at most 14, so we can distinguish at most 14 attributes of the dataset using oriented sliver textures.
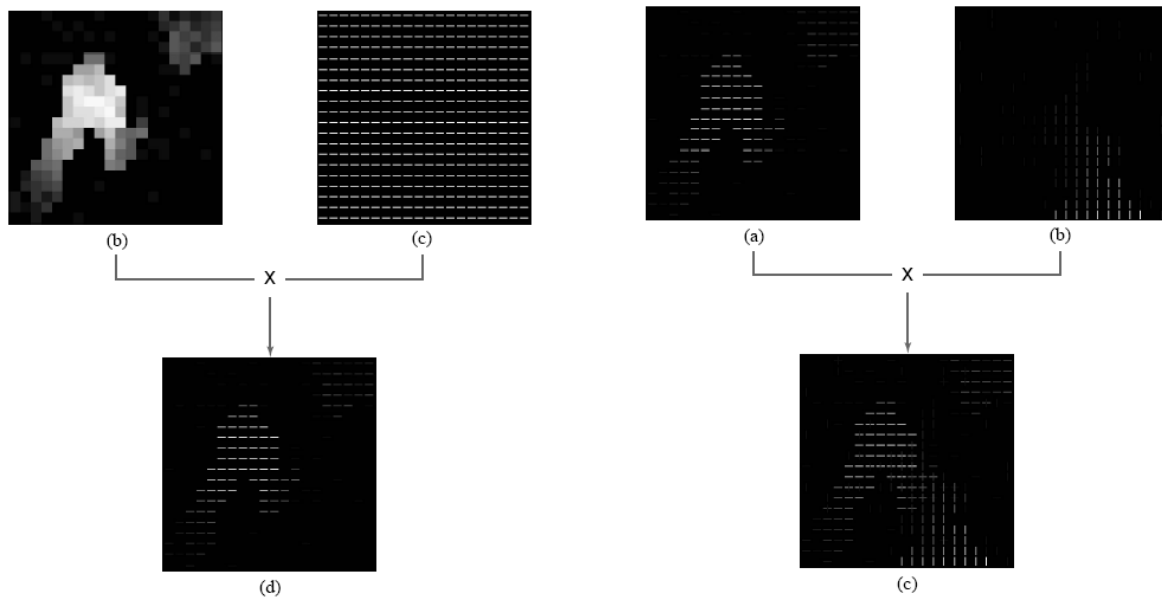
Fig. 1. Left: The construction of one layer of slivers from a scalar field. (b) shows the luminance of the scalar field. (c) Shows a layer with horizontal slivers. (d) Shows the multiplication of the luminance and the slivers. Right: The visualization of two scalar fields in one picture. (a) and (b) are two different layers which represent some data using sliver textures. (c) is the combination of both layers.

When several sliver layers with a high intensity are overlapped, shapes like +, x or stars are formed. Because these shapes are more prominent, a viewer can distinguish these regions better, but these features were not measured during the experiment. If some parts of the data have to be emphasized, a re-rotation of a certain sliver layer is desired to obtain some patterns in the final image. If the relation between two dimensions in the dataset has to emphasized, a rotation to 0 and to 90 degrees can improve the detection of both features. Figure 2 shows a re-rotation of two sliver textures in order to emphasize two scalar fields in the dataset. To remove strong artificial artifacts from the final image, each sliver layer is jittered.

Using oriented sliver textures we can visualize up to 14 scalar fields which are spatial dependent. For each dimension in the dataset we assign a different sliver texture and the value of the scalar field defines the luminance of the slivers. The final image contains up to 14 different sliver textures and the luminance of each sliver corresponds with the value in the scalar field. Figure 2 shows the result of a visualization of a multidimensional dataset using oriented sliver textures.

Using this technique several scalar fields can be display in one picture. A viewer can use these images to:

– Determine which field is prominent for a region.
– Determine how strongly a given field is present.
– Estimate the relative weight of the fields values in a region.

– Locate regions where all the fields have low, medium or high values.
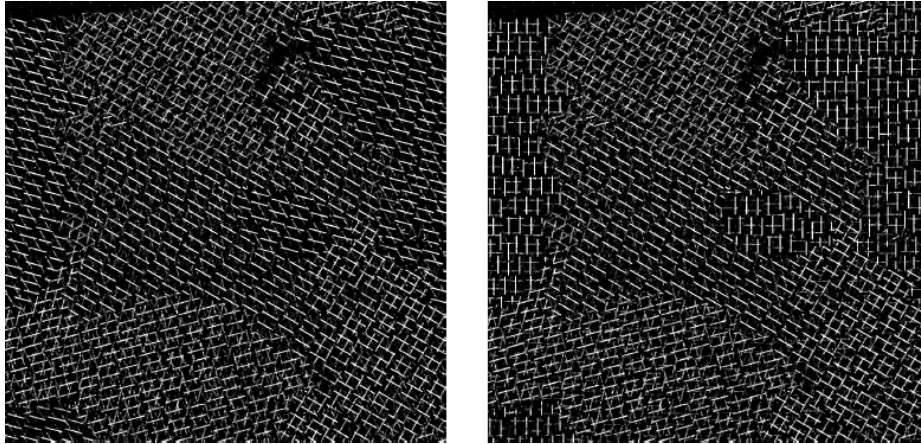
Fig. 2. Left: an image containing several data layers using oriented sliver textures. Right: the same data visualized after a re-rotation of some sliver layers.

2.2 Semantic depth of field

R. Kosara et al. [2] investigates the useful properties of semantic depth of field. Semantic depth of field is a focus+context technique that uses selective blur to make less important objects less prominent, and thus point out the more relevant parts of the display to the user. The technique is based on the depth of field technique used in photography, which depicts objects sharply or blurred depending on their distance from the lens. This is used to guide the viewers attention, and is quite effective and intuitive.

The researcher's goal was to find out how semantic depth of field is an effective method for guiding the user's attention, and how it is useful in applications. They first investigate how depth of field or blur can be applied. By measuring response times of the subjects they found how depth of field can be applied. After that, depth of field is compared with other techniques such as coloring. Finally the blur perception of humans is measured.

Preattentivity Preattentive processes take place within about 200 ms after the stimulus is presented. These processes involve a limited set of features for which certain tasks can be performed easily. These features include: orientation, closure, color, proximity, etc. The tasks are: detection, location, count estimation, recognition of groups, etc. Using preattentive features for visualization makes the information easier to see in order to get an overview. Especially methods for pointing out information have to make the relevant objects immediately stand out.

Test procedure The test procedure tests two preattentive abilities: able to detect and locate sharp objects, and estimate percentage of targets among distractors. Each presented test image contains 3, 32, or 63 distractors and one or none targets. For each distractor a blur level of 7, 11 and 15 pixels was used. The blur level is measured by how much one pixel is spread when blurred. A blur level of 1 gives a perfectly sharp image. An example of a test image is given in [2]. Each participant was shown 210 different randomly generated images. They had to answer in which quadrant contained the target, if present. Each image was visible for 200 ms.

In the second test each participant had to estimate the number of targets with a number of distractors in the background. Possible answers were few, intermediate and many. The results of this experiment was that the accuracies for correct locations of the targets were very high (>90%) or high (>60%) depending on the blur level. When the lowest blur level of 7 pixels was present, the accuracy dropped significantly, because we can not distinguish the distractors from the targets.

Because humans can locate the target and estimate the number of sharp objects within 200 ms, it must be a preattentive process.

Interplay Because semantic depth of field is not very likely used without any other visual cues, the researchers investigated how semantic depth of field interacts with other features, such as color, and orientation. The test was similar as the previous test but now the target was always visible. The test used three methods; simple, disjunctive and conjunctive searches. Simple searches are based on the presence of one target feature in the image, with distractors not different from one other. In a disjunctive search, the subjects looked for one target feature, but now the distractors could also differ from another one (e.g. if the red object is the target, all distractors were black but could be sharp or blurred). Conjunctive search required the participant to look for a combination of two target features (e.g. the red and sharp object), while the distractors could have any other combination of the two.

The researchers found that in terms of search time that semantic depth of field was not significantly worse than color. This was what the researchers found the most and surprising findings of the experiment. Furthermore, there was no significant difference between a simple search for colored or for sharp objects. Conjunctive searches for color and blur, orientation and blur, and color and orientation differ significantly from each other, with color and orientation being the slowest. Each of these two features combined with semantic depth of field is faster than the simple and disjunctive searches, which was quite contrary from what the researchers expected, because conjunctive searches are usually slower.

Blur perception The researchers were planning to use semantic depth of field as a separate visualization dimension that could be used in addition of existing visualization dimensions, such as color and orientation. To do this, they needed to know the minimal differences between blur that humans can perceive. The researchers expected an exponential relation between the blur level and the perceived blur (like the way luminance is perceived). The test showed that semantic depth of field cannot be used as a full visualization dimension. Participants were able to detect differences between objects with different blur levels with good accuracy, but they were not able to correctly identify objects with the same blur level. A blur higher than 7 did not improved the results. Furthermore, participants commented that they disliked having to look at blurred objects and to compare them. It therefore appears to be necessary to make sure that no important parts of the image is blurred and that the user can switch to a different view to see a completely sharp image at any time.

Application results Some applications (Textdisplay, scatterplots and mapviewer) were tested which uses semantic depth of field. The results of these tests are much less conclusive, due to both technical and design problems of the applications. The main results for the tested applications was that there was no significantly difference between using semantic depth of field or other features, such as color and orientation. Figure 3 gives an impression of the mapviewer application.
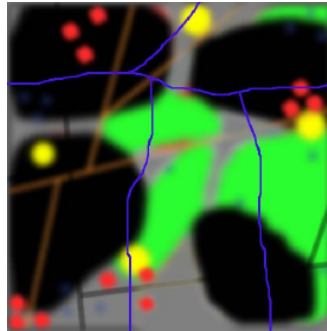
Fig. 3. An application of semantic depth of field. The image shows the mapviewer application.

2.3 Perceptual Optimization of Complex Visualizations

D. House and C. Ware [3] describes a method for visualizing multiple layers of three dimensional surfaces using a stereoscopic setup. A common problem within visualization applications is to display multiple overlapping surfaces. They found that texturing plays an important role for revealing surface shapes, but no general guidelines are present for creating such textures. The problem they wanted to solve was how to choose the pair of textures so that they both optimally reveal

surface shapes that are maximally visible to human subjects. Because of the complexity of the problem, no standard psychophysical experimental methodologies could be used, so they developed a new method for solving such problems.

Perceptual properties of textures Texturing surfaces is especially important when they are viewed stereoscopically. This becomes obvious if we consider that a uniform non-textured polygon contains no internal stereoscopic information about the surface that it represents. When a polygon is textured, every texture element provides stereoscopic depth information, relative to neighboring points. Contours that are drawn on a shaded surface can drastically alter the perceived shape of a surface. Figure 4 has shaded bands that are added to provide internal contour information. The contours that are drawn on both surfaces are different and the two rectangular areas contain exactly the same shading. The combination of contour information with shading information is convincing in both cases, but the surface shapes that are perceived are very different. This tells us that shape information is inherently ambiguous. It can be interpreted in different ways depending on the contour.
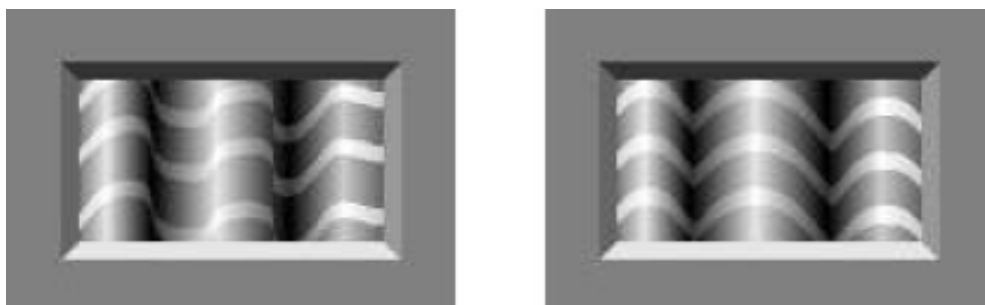
Fig. 4. Left: a surface with a shading and contour texture. Right: the same surface with the same shading, but with a different contour texture.

Problem of layered surfaces The problem which they encountered was that is was not clear how to choose the texture characteristics that made use of both stereoscopy and motion. They developed a parameterized texture space that allowed a number of texture attributes that might bear on the layered surface problem. The most important attributes they used for the textures are orientation, foreground transparency, density of pattern, regularity of pattern, softness and background color. For the texture elements the most important attributes were transparency, size, linearity, orientation and color. Because of these attributes, the search space for textures was too large, so another method must be used to find the optimal texturing for both surfaces. They used a genetic algorithm to find the attributes for creating optimal textures.

Test results Each participant was shown two layered surfaces combined with two generated textures. The textured surfaces were presented in stereo and with motion. The participant should be able to detect the shapes of both surfaces. For each pair of textures, participants grade the set of textures. At the end of the test, new textures were generated according to the results of the previous test. After about 20 generations, the genetic algorithm produces proper results. Figure 5 shows a result of the algorithm. The problem of this approach is that the process produces proper solutions for complex visualizations, but the resulting textures are not necessarily the simplest and most elegant. Another problem was that the algorithm produces proper, but different results for different participant, so no general solutions were found using this method. By combining the results of multiple participants, the researchers expect better and general results.
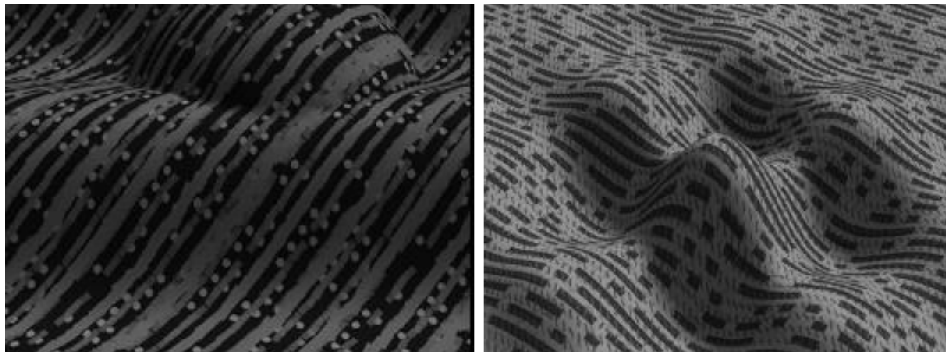
Fig. 5. A result produced by the genetic algorithm. Left: the upper surface. Right: the lower surface.

3 Discussion

From the reviewed papers we can conclude that some properties of visual perception are useful when layered visualization of multidimensional data is considered. Especially preattentive features, such as blur, color, luminance and rotation can be used to distinguish different layers. Some features can be used as a display dimension, such as luminance, which corresponds with the values of the scalar field for a certain dimension.

C. Weigle et al.[1] presented a method which uses oriented sliver textures combined with luminance to visualize a scalar field. The question the researchers asked themselves was how humans can distinguish the sliver textures. From a number of experiments they found that humans can distinguish sliver textures when the minimal angle between the slivers is 15 degrees or more. The presented method is very useful in the suggested application domains. Participants who viewed at the final images were able to interpret the presented data and the

relations between the several dimensions. The paper itself gives a constructive method which can be used for visualizing multidimensional data as a set of layers. The researchers suggested to extend the method by adding a thickness of the slivers, but the impact of this was not tested within the pertinent experiments.

R. Kosara et al. [2] conclude that the use of blur is a functional addition to the preattentive features, but it was not found significantly better than other preattentive features. When blur was used in combination with other features, it was found to be faster than any other combination of preattentive features.

The paper itself gives a clear indication where blur can be applied and where not. As said before, they concluded that blur is a good addition, although they also explicitly concluded that humans dislike looking at blurred objects. Furthermore, humans can not distinguish objects with different blur levels, besides that a blur level of 7 pixels is about the minimum that humans can distinguish. Therefore, they concluded that the use of blur has to be avoided to express some values of the scalar fields, but can be used to emphasize important objects by blurring the less important one.

The authors found that semantic depth of field is not very likely to be used in combination without other visuals cues. Why they mention this is not very clear, we assume this is related to the results of the conjunctive search tests. The result of each test was significantly better when visual cues were combined with semantic depth of field.

D. House and C. Ware [3] presented a method for generating textures which are used to visualize two different surfaces in combination with motion and stereoscopy. The researchers gave a proper indication of the problem with texturing three dimensional shapes, but the paper does not give a constructive solution. The solutions of the method are satisfying, but only for a specific type of person. Combining the results of the tests, a more general result can be obtained. They liked to develop a process for abstracting the texture attribute values in order to make a simpler and cleaner solution. From this some guidelines can be given about how certain texture attributes can be applied. One interesting question could be how can humans distinguish both surfaces when no stereoscopy is used?

The reviewed methods can be applied when visualization of multi-layered data is considered. Each method can be applied in a different way. The oriented sliver method is perhaps the most intuitive method for visualizing multi-layered data. The method can be applied when the dimensions of the data is limited by 14. Semantic depth of field is not a technique to visualize multi-layered datasets, but it can be used in combination with other techniques in order to guide the user's attention to some parts of the data. In [2] the authors tested a map-viewer application were multi layered maps were visualized. Important layers were emphasized by blurring other layers using semantic depth of field. Figure 3 gives an impression of this application. The method described in [3] can be

used where three dimesional surfaces are overlaying other three dimensional surfaces. Applications can be found in medical visualization where different kinds of tissue overlaying other structures/surfaces. The surface of each stucture has to be recognized properly. By choosing the proper texture, humans can reveal the surfaces better.

Semantic depth of field can perhaps improve the oriented sliver texture technique. In [1] the authors did a re-rotation of the silvers to emphasize some parts of the data. We think that semantic depth of field can also be used in combination with oriented sliver textures to emphasize some parts of the data.

The test procedures of both semantic depth of field and oriented sliver textures were very similar. For each test participants have to find some targets surrounded by distractors. The main goal of the test for oriented sliver textures was to find the minimum angle between slivers. For semantic depth of field the search time was a goal. Comparing the search times of both tests using semantic depth of field, a participant was much faster in finding the target than the other test. Because blur is a preattentive feature, targets can be found within 200 ms. The search time for the oriented sliver textures was higher because participants have to search for the targets, which took more time. We think that the oriented sliver textures technique can be improved with semantic depth of field, but this has to be investigated.

Semantic depth of field could also improve the multi-layer visualization mentioned in [3]. For example, one texture can be blurred such that the other layer is better perceived. Maybe humans can perceive the surfaces better when semantic depth of field is used in combination with this technique without stereoscopy?

References

1. C. Weigle, W.G. Emigh, G. Liu, R.M. Taylor, J.T. Enns, and C.G. Healey. Oriented sliver textures: A technique for local value estimation of multiple scalar fields. In Graphics Interface, pages 163–170, May 2000.
2. R. Kosara, S. Miksch, H. Hauser, J. Schrammel, V. Giller, and M. Tscheligi. Useful Properties of Semantic Depth of Field for Better F+C Visualization. Proceedings of the Joint Eurographics–IEEE TCVG Symposium on Visualization (VisSym 2002), pp. 205-210.
3. D. House and C. Ware: A method for the perceptual optimization of complex visualizations. Advanced Visual Interface, Trento Italy, pp. 148-155, 2002.

Multi-layer Visualization: A Review of Selected Methods

Caesar Ogole, Julius Kidubuka
Institute for Mathematics and Computing Science
University of Groningen, The Netherlands
{C.Ogole, J.Kidubuka}@student.rug.nl

Abstract

While the advances in scientific visualization have made it possible to convert contextual data sets into conspicuous meaningful images, some areas still need further exploration. One of these challenges, which is the focus of this review, is the problem posed by the question: "Given large, complex and multi-dimensional data sets that represent overlapping surfaces and fields in the real world, what visualization technique can be applied to optimize the display of this class of images?" This problem is particularly difficult owing to the fact that solution methods to multi-layer visualization problems do not only involve many variables such as textures, colors, orientation and (the degree of) transparency of overlaying surfaces, but also, have to integrate user-centered issues such as user feedback. Human perception is core to the considerations. Moreover, the integration of these factors into a typical visualization system takes the form of parametizations of complex and highly interactive algorithmic procedures. No standard guidelines to select suitable sets of parameters exist. In this paper, we review three different techniques of enhancing multi-layer visualization.

1 Introduction

The goal of scientific visualization is to transform sequences of numbers and character strings into images from which useful information can be inferred. These sequences fall into categories, each of which may possibly be representing some attribute(s) of the entity in question. The representative values are called datasets. Since most entities are complex, it is only possible for them to be represented by large and multi-dimensional datasets, which in general, have many different data elements. The complexity of the datasets poses a big problem to visualization processes, particularly, in the case where the encoded image information in the data patterns is multi-layer in nature. This problem is not merely a classic case of investigation in theoretical computations but it is also an area having vast applications in the real world, for example, in medical imaging where, in practice, a subject has to view the tissues overlain by skeleton or vice-versa.

The problem of multi-layer visualization can be understood quite easily by thinking of an image scenario where there are a collection of N surfaces (or fields) that overlap one another in space (Figure 1). *How do we view the overlaying surfaces*? Suppose the top surface is more opaque, can we still view the bottom surface conspicuously without having its shape distorted in any way? Obviously, the viewer will not be able to extract useful information. This is likely to render the visual system, and hence the applied technique, not user-friendly or even useless.
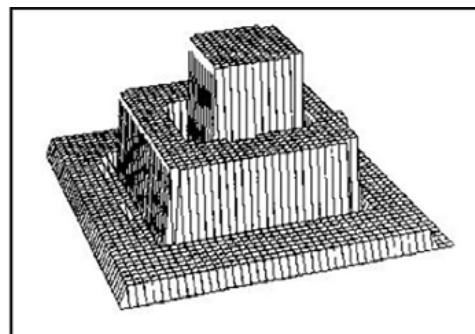
Fig.1. The "wedding cake" illustration of overlapping surfaces

Thus, this paper provides a brief review of some of the existing methods of enhancing multi-layer visualization. At some point during its use, the technique employed in such a system requires that a user chooses values for system parameters so as to guide the system in improving or refining its output. This is the actual transformation phase where the representative datasets are converted into multi-layer (overlapping surface) images. Variables of special interest include, but are not limited to, surface texture colors, orientation, opacity (or selective blur), shapes, distribution, sizes and segmentation. While it would be a good idea to study the effect of each of these variables over the others (one at a time), it is practically not possible given that the number of variables is very large.

2 Related work

The quest in finding effective solution to the problem of multi-layer visualization is not a new thing. Several attempts had hitherto been made by various researchers and it is these contributions that served as a starting point for the techniques

reviewed in this paper. We give a brief overview of related work.

Methods of analyzing image textures using statistical techniques have been shown to work well under certain conditions [1], although it focused mainly on a single task (texture). The big setback with the previous methods was that they did not take into account other relevant image attributes. The texton theory [2], a contribution by Juliész, pointed out that early vision detects three types of texture features, namely, elongated blobs with specific visual properties (for example, colors and orientation), ends of line segments and crossings of line segments. Closely related to multi-layer visualization applications, Interrante[3] uncovered that if one or both surfaces are given spatially transparent texture, this can help to define and distinguish them. Interrante further reported that additional depth information provided through stereoscopic viewing and motion parallax can make them stand apart. However, none of these studies gave guidelines for choosing texture pairs that both optimally reveal the surface shape and do not interfere with one another. The problem of perception thus crops up.

All the works related to multi-layer visualization are about image surface texture, because surfaces in nature are generally textured. As explained by Gibson [4], texture is an essential property of a surface. A non-textured surface, he said, is merely a patch of light.
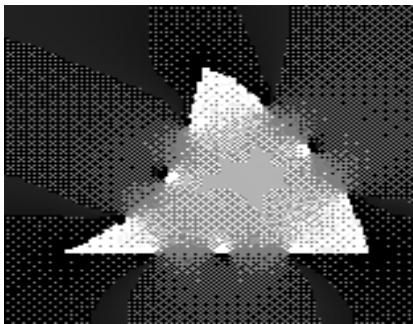
Fig.2. Textures as an important surface property

Some authors such as Dawkins [5] and Sims [6] went as far as giving useful hints into methodological boundaries in generating solutions to visualization problems. Notably, it is observed that the structure of a genetic algorithm provides a convenient means for visualization and user feedback. This is the heuristic that has been used extensively in the visualization techniques reviewed in this paper. Genetic algorithms are particularly good for this class of computational problems because they allow prioritization of operations within regions that are believed to be "promising" in the context of generation of better or improved results.

3 Review of Methods

In this survey, it is observed that no single technique is best. To this effect, each technique is first described to some satiable length and level of detail before the pros and cons are shown. The hope is that detailed understanding of these attempts will lead to further refinement by other researchers who will venture into addressing any pitfalls in the methodological designs.

3.1 Method for perceptual Optimization of Complex Visualizations

In this first technique [7], the tasks involved in the solution method can be divided into three stages. The first problem is to represent the datasets using appropriate data structures. Arrays (or vectors) of data elements, whose sizes depend on the dimensionality of the datasets, are used for this purpose. The term *gene* is used to refer to the encoded parameter vector, tuned as an input to the transformation procedure in the next phase. The idea of encoding datasets at this stage is to define the search space within which all derived solutions must lie. In the second phase, gnome is fed into the conversion process which is actually a routine that falls under the family of genetic algorithms. Notable at this phase is the involvement of the user in supplying parameter values that reflect preferences and hence, guiding the algorithm during the iterative output refinement. The third and final phase is to characterize the results so as to select the palatable solution. This is done according to the clustering criterion.

Why a genetic algorithm? The search space can be overwhelmingly large for exhaustive search. A genetic algorithm is useful because the search proceeds only towards promising regions, and is somewhat resilient in avoiding poor local minima.

3.1.1 Data Structure

Fig 3. A 6X6 lattice with dots

In this representation, each texture tile is structured as a set of three lattices. Using standard texture mapping approach, a complete texture across a surface is tiled from a single base file. The tile is divided into a uniform square grid by the lattices.

3.1.2 The algorithm

The pseudo-code for the genetic algorithm is shown in Appendix 1. It is important to understand the terminology used in the procedure description. Each encoded parameter is referred to as a *gene*. Arrays of gen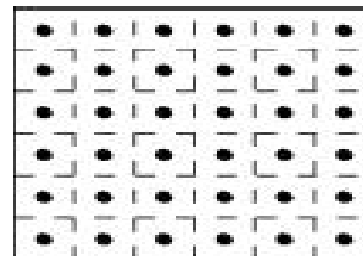es, or *genotypes,* are in turn stored in an array of *generation.* It is the generations (arrays of arrays, or multi-dimensional arrays) that are fed into the algorithm as inputs. The gene is usually encoded as an integer or floating point and each gene controls an aspect of a texture pair. Prior to the algorithm's first run, the generations are initialized to randomized values. Subsequent values depend on the user preferences that guide the interactive algorithm towards desired optimal generation values.

3.1.3 Results

On average, it was found that generation of acceptable solutions through the repetitive steps took two hours per subject. This process was successful in producing good results to the problem of visualizing overlaying surfaces for all the subjects. However, initial randomized data (generation) values for the texture pairs did not seem to have a good representation of the optimal solution values. Iteration run time decreased with increasing number of iterations, and after about twenty minutes, there seemed to be less distinction in the next and previous results. It was observed that opacity of the top surfaces greatly affected the visualization of overlapping surfaces (Figure 4). However, variation in colors apparently had negligible effects. This rendered colors only important in attribute display.
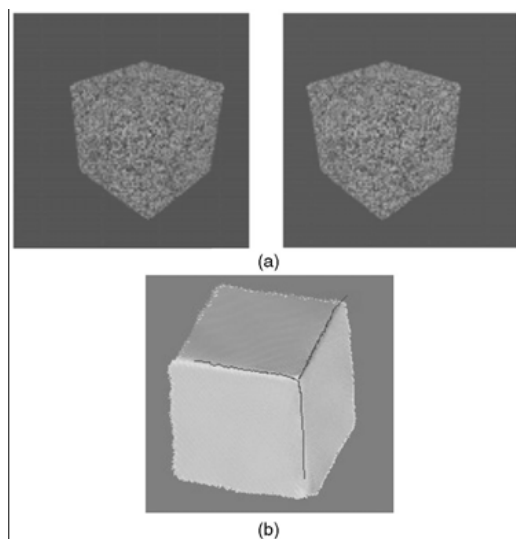
Fig.4. Degree of transparency affects visualizing overlapping surfaces. (Compare a and b)

3.2 Focus + Context Technique

Semantic Depth of Field (SDOF) [8] is the second technique employed in the display of overlaying surfaces. SDOF is based on the depth of field (DOF) effect borrowed from cinematography and photography that depicts objects (sharply or blurred) depending on their distance from the lens. Selective blur images based on relevance (rather than geometry) are used to guide the viewer's attention to the unblurred objects in the image. The aim of this technique is to efficiently and effortlessly present information to the user/viewer. The sub-techniques (stages) involved are split into three.

In the first stage, we make use of a process known as *preattentivity*. Two preattentive abilities are tested: being able to detect and locate a sharp object, and being able to estimate the percentage of targets among distractors. Experimental results showed preattentivity provides a reliable technique of finding sharp targets among blurred distractors. The accuracies for correct location of targets were very high (atleast 90 percent) or high (atleast 60 percent) depending on the blur level. Significant drop in accuracy was attributed to presence of the lowest blur level. Preattentive processes take place within a very short time (~200ms) and involves a limited set of features (such as orientation, closure, color, proximity, etc.) for which certain tasks (e.g. location, detection) can be performed with ease.

In the second phase, *interplay* is used. Figure 5 depicts an example of the images used for this part. The focus of interest is in the interaction of SDOF with other features (color and orientation are selected for use) because it is very likely that SDOF will not be used without any other visual cue apart from sharpness. Simple, disjunctive and conjunctive searches are tested as a way of detecting the presence of one or both features in the target.
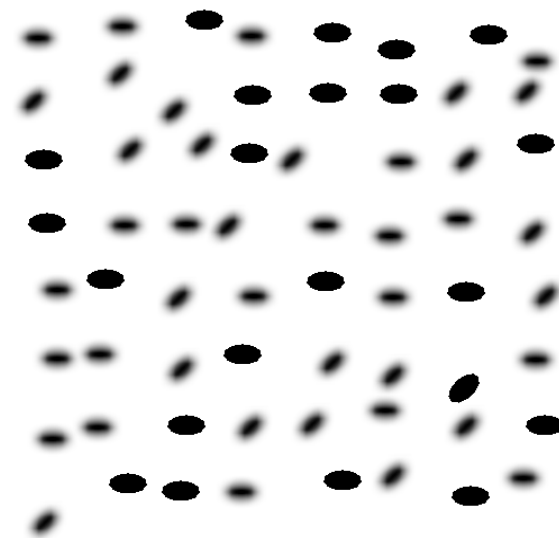
Fig.5. Example image for interplay

In the last stage (known as blur perception), the smallest difference that can be perceived in blur, and the rate at which "steps" in blur are perceived are assessed. This is based on the assumption that there is an exponential relationship between the blur level and the perceived blur. To do this, a test is performed (with the help of some participants) and it consists of a number of parts: testing the ability to tell whether or not two objects have the same blur level, the absolute thresholds of blur perception and finally to tell the perceived relation in blur in terms of a ratio of two numbers. Effectively, multilayer visualization is enhanced in that SDOF can then decide for every object whether to display it sharply or blurred (Figure 6). The decision is based on the object's current relevance.

Fig. 6. Application example of a chess board, with the chessmen threatening the knight on e3 in focus (taken from Kosara et al.6)

3.2.1 Results
In terms of search time, SDOF was observed not to be significantly worse than color; this was perhaps the most interesting and surprising finding of the study. There was no significant difference between a simple search for colored or for sharp objects. The conjunctive searches for color and blur, orientation and blur, and color and orientation differed significantly from each other, with color and orientation being the slowest– each of these two features combined with SDOF was faster. Also, the conjunctive search for color and blur was not significantly slower than the simple and disjunctive searches, which was quite contrary to what was expected, because conjunctive searches usually were slower.

3.3 Local Value Estimation of Multiple Scalar Field using Oriented Sliver surfaces
To support the simultaneous display of multiple overlapping scalar fields, this texture generation technique [9] combines orientation and luminance that are selected based on psychophysical

experiments that studied how the low-level human visual system perceives these visual features

3.3.1 Data Representation
Datasets in numerous practical applications can be viewed as a collection of n scalar fields that overlap spatially with one another. Rather than using n visual features to represent these fields, only two features are used: orientation and luminance. For each scalar field (representing attribute $\mathbf{A_i}$) a constant orientation $\mathbf{o_i}$ is selected; at various spatial locations where $a_i \in \mathbf{A_i}$ value exists, a corresponding *sliver texture* is placed oriented at $\mathbf{o_i}$. The luminance of the sliver texture depends on a_i: the maximum $a_{max} \in \mathbf{A_i}$ produces a white (full luminance) sliver, while the minimum $a_{min} \in \mathbf{A_i}$ produces a black (zero luminance) sliver. A perceptually-balanced luminance scale running from black to white is used to select a luminance for an intermediate value. This scale was built to correct for the visual system's approximately logarithmic response to variations in luminance.

3.3.2 Procedure
Values in a given scalar field are given orientations (in degree angular measure, for example). Combining these orientations form sliver layers. Multiple scalar fields are displayed by compositing their sliver layers together.
With varying backgrounds orientation, say from 0º to 45º, in the intervals of 5º, (resulting in 10 different background subsections (0, 5, 10, ..., 45º), a discrete function $f(bg)$ is defined for the different background orientations. The function f returns rotational differences in the intervals (*e.g.,d*= 5º, *d*= 10º, etc). Every possible target orientation was tested for each separate background. Several trials were run during the experiment

The goal of the experiment was to find how much counter clockwise rotation is needed to differentiate a group of target elements oriented $tg=bg+ d_{ccw}$, *and* $tg=bg- dccw$ *where bg is* set of background elements orientations, d_{ccw} is the counter clockwise rotation.

3.3.3 Results
In general, using multi-factor analysis of variance (ANOVA) and least-squares line fitting, it was found out that target oriented $d = \pm15°$ or more from its background elements resulted in the highest accuracies and the fastest response times, regardless of background orientation.

4 Discussion
In this survey, we looked at three different methods applied in multi-layer visualization. It is observed that the major difference among the three lies in the steps followed in generating acceptable solutions to

multi-layer visualization problem. Some of the procedures encompass more parameters than others

Whereas the first method (described in section 3.1) focuses mainly on selecting appropriate texture pairs that represent backgrounds and foregrounds of superimposing surfaces, the F+C technique (section 3.2) concentrates on utilizing the useful properties of Semantic Depth of Field (SDOF) for better visualization. The property used in SDOF is the fact that the selective blur aids in guiding user attention to the most relevant objects. On the other hand, the third visualization technique (section 3.3) is a texture generation method that combines orientation and luminance to support simultaneous display of multiple scalar fields.

As noted before, each method comes along with trade-offs. While the method for perceptual optimization of complex visualization has the power of handling multivariate characteristics of complex data, the criterion applied is not simple. It does not automatically produce solutions that are elegant. More abstractions are needed.

The method of local value estimation of multiple scalar fields, too, has a number of limitations, for example, as the number of attributes grows, it becomes difficult to find additional features to represent them. This technique does not provide a mechanism to handle interference (a phenomenon where different visual features will often interact with one another producing visual distortion). Nevertheless, it's more practical.

The SDOF approach (section 3.3) seems to be intermediate with respect to the first two techniques except that the optimality of its solution needs further investigation. Also, SDOF cannot be used as a full visualization dimension since in most cases it is used without any other visual cues.

From this survey, we may conclude the following.

Firstly, we observe that any approach to solving visualization problems, multi-layer in particular, requires interactivity between the user and the system so that perceptual problems can be solved through parameter adjustments, for example, by changing color, brightness etc. Analysis of the visualization techniques also reveals that genetic algorithms are a promising way of improving multi-layer visual problems. The solution to visualization problems cuts across various disciplines including computer science / graphics, psychology, photography, cinematography, etc. Finally, we see that appropriate combination of image attributes, such as textures, and orientations is important in visual display of overlapping surfaces. Poor combination of the values of these parameters results in orientations that may not be distinguished.

5 References

[1] REED, T. R. AND HANS DU BUF, J. M. A review of recent texture segmentation and feature extraction techniques.
Computer Vision, Graphics, and Image Processing: Image Understanding 57, 3 (1993), 359–372.

[2] JULÉSZ, B. A brief outline of the texton theory of human vision. Trends in Neuroscience 7, 2 (1984), 41–45.

[3] Interrante, V., Fuchs, H., and Pizer, S.M. (1997) Conveying shape of smoothly curving transparent surfaces via texture.
IEEE Trans. On Visualization and Computer Graphics 3(2)
98-117.

[4] Gibson, J.J. (1986) The ecological approach to visual perception. Lawrence Erlbaum Associates, Hillsdale, NJ.

[5] Dawkins, R. (1986) The Blind Watchmaker, Harlow
Logman.

[6] Sims, K. (1991) Artificial Evolution for Computer Graphics,
Computer Graphics 25, 319-328.

[7] House, C (2002) A method for the Perceptual Optimization of Complex Visualizations

[8] Kosara, R (2001)Useful Properties of Semantic Depth of Field for Better F+C Visualization

[9] Weigle, C (2000) Sliver Textures: A Technique for Local Value Estimation of Multiple Scalar Fields

[10] Christopher Healey and James Enns. Large datasets at a glance: Combining textures and colors in scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):145–167, April 1999.

[11] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization:
A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, January-March 2000.

[12] SMITH, P. H. AND VAN ROSENDALE, J. *Data and visualization corridors report on the 1998 CVD workshop series.* Technical Report CACR-164 (sponsored by DOE and NSF), Center for Advanced Computing Research, California Institute of Technology, 1998.

Appendix 1: Pseudo-code for Perceptual Optimization of Complex Visualizations

```
G is current generation, V is next,
N even
Generation G, V of size N;
Evaluation E of size N;
Phenotype P;
Restart from last saved or new
random generation
if restarting from a previous
session then
     (G,E)= LoadFromHistoryFile();
goto restart;
else
RandomlyInitialize(G)
endif
Main evaluate – breed – mutate
loop
loop
extract visualization, display and
evaluate
for each genotype Gi in G do
P = Phenotype(Gi);
Ei = UserEvaluation(Display(P));
endfor
SaveEvaluatedGenotypesToFile(G, E);
restart:
breed probabilistically based on
evaluation, one
breeding pair produces 2 offspring
for k = 1 to N step 2 do breed
(i, j) = SelectBreedingPair(G, E);
(Vk, V k+1) = CrossoverBreed(Gi,
Gj);
endfor
G = V; make new generation the
current one
for each genotype Gi in G do
Mutate(Gi); mutate with low
probability
until UserRequestsExit; keep going
until user quits
```

Acknowledgement

Java versus C++

Bart Postma[1]and Remko de Jong[1]

[1] Department of Computing Science, Rijks*universiteit* Groningen
{s1339095, s1277081}@student.rug.nl

Abstract. Java was originally intended to replace C++. However, nowadays both languages still coexist and neither one has been able to replace or subdue the other. In this paper we will make a critical comparison of both languages and try to discuss both differences and similarities of the two languages.

1 Introduction

C++ (originally named C with classes) was developed during the 1980's as an enhancement to the regular C programming language. It introduces for instance some object-oriented (OO) features to C. It offers classes, which provide the four features commonly present in OO (and some non-OO) languages: abstraction, encapsulation, polymorphism and inheritance. Since the 1990's C++ has been one of the most popular commercial programming languages.

Java is an object-oriented programming language developed by Sun Microsystems during the early nineties. Initially it was called Oak and it was intended to replace C++. Because of its platform independence Java quickly conquered the web. After a few years of popularity, Java's dominant place in the browser gradually diminished due to the introduction of replacements such as Macromedia Flash. On the server-side however, Java presently is more popular than ever.

Nowadays both languages coexist and neither one has been able to replace the other. Java and C++ each have their advantages and limitations and that is why the choice for one of them usually depends on the type of application that has to be built. In this paper we will discuss the differences and similarities between Java and C++ and use these findings to fence off a global area of purpose for both programming languages. We will look at different properties of each language, varying from types and expressions to memory management and execution speed.

2 Types

Java has 8 different fundamental types and C++ has 13 different fundamental types. In Java the fundamental types have a size that is specified by the language. This means that for example an `int` is always 32 bit and a `long` is always 64 bit in Java. In C++, the sizes of the fundamental types vary from one implementation of C++ to another.

C++'s `char` uses the Latin-1 encoding, which includes the ASCII encoding, and it is usually 8 bits. The Java `char` uses the Unicode encoding for its `char` type and it is 16 bits. By making use of the Unicode encoding, Java permits the use of characters from many languages, such as Greek, Hebrew, Japanese, etc.

The `boolean` type in Java can only have two different values, it is either `true` or `false`. In C++ a `bool` represents `false` if it has the value zero and `true` otherwise. C++'s `bool` is a relatively new fundamental type of the language. This might give some problems, because old compilers might not always recognize it. Also older software that defined a type `bool` itself may get errors because the type is now built-in.

Both Java and C++ have support for enumeration types. Java supports enumeration types only in later versions, starting from JDK 1.5. Two simple examples of enumeration types in C++ are:

```
enum Days (sun, mon, tue, wed, thu, fri, sat)

enum    (chemistry,    mathematics,    computer    science,
economics)
```

Variables in Java either have to be initialized or receive a default value. This is not the case in C++.

The difference between Java and C++ is not much when it comes to fundamental types. C++ can cause confusion with its fundamental types, because the sizes of them vary from one implementation of C++ to another, its `bool` type also might cause problems with older compilers and software. Java handles booleans more elegantly than C++, C++'s use of a zero to represent `false` and `true` otherwise, can cause confusion to certain users. C++ `unsigned` types make it possible to use space allocated for fundamental types more efficiently. Also C++ does not ask for the initialization of variables nor gives default values to them. This saves a little bit computing time, but one has to be careful with this, otherwise errors arise when variables are not initialized when they should have been. Java's use of Unicode encoding has the advantage that programming is easier for people speaking Greek, Hebrew, Japanese, etc. But the software developed is only well understood by those people and not by anyone else.

3 Pointers and references

C++ has pointers and Java has reference types. Java's reference type has some resemblance to C++'s pointer. A variable that is of reference type can be used to refer (or point to) an object, which is done using the `new` operator. An example of this in Java is:

```
Car someCar;
someCar =
  new Car("Toyota", "Avensis");
```

3.1 C++ reference operator (&)

C++'s pointers require some more explanation. The memory of a computer can be imagined as a succession of memory cells. Usually these are bytes and they are numbered in a consecutive way. The address that locates a variable within memory is called a reference to that variable. This reference to a variable can be obtained in C++ by preceding the identifier of a variable with an ampersand sign (&), known as the reference operator, and you can translate it to "address of". An example is:

```
var1 = 14;
var2 = &var1;
```

Assuming the number of the memory cell of `var1` is 1078, the value of `var2` is 1078. The second statement does not copy the value of `var1` to `var2`, but a reference to it. It can be read as "`var2` becomes the address of `var1`". The variable that stores the reference to another variable (`var2` in the example) is called a pointer. Pointers "point to" the variable whose reference they store.

3.2 C++ dereference operator (*)

Using a pointer, it is possible to directly access the value stored in the variable to which it points. To do this, you have to precede the pointer's identifier with an asterisk (*), which is known as the dereference operator and can be translated to "value pointed by". An example is:

```
var1 = 14;
var2 = &var1;
var3 = *var2;
```

Again assuming the address of `var1` is 1078. Then `var3` would take the value 14, since `var2` is 1078, and the value pointed by 1078 is 14. The third statement can be read as "`var3` equal to the value pointed by `var2`".

3.3 Declaring pointers

Because pointers can directly refer to the value they point to, it becomes necessary to specify the data type it points to in its declaration. It is not the same thing to point to a `char` as to point to an `int`. For example, a pointer that points to a `char` and a pointer that points to an `int` are declared as following:

```
char * pointerToChar
int * pointerToInt
```

Users not familiar with the syntax of C++ might get confused with this declaration. The asterisk (*) used in the declaration is not the same as the deference operator, which also uses an asterisk.

Pointers have different purposes. For example, they can be used with passing arguments to functions or for manipulating arrays. Pointers can also be used in association with the new operator to allocate space for a value at runtime:

```
Car * someCar;
someCar =
  new Car("Toyota", "Avensis");
```

At runtime the `new` operator is executed and this creates an object of the class `Car`. After that, the assignment statement causes the pointer variable `someCar` to point to the Car object `someCar`.

The key difference between Java references and C++ pointers is that a reference always refers to a valid object in memory. This is not the case for a pointer. A pointer can be reassigned to point to something else than to what it first did. It is also possible to apply arithmetic operations on a pointer, also causing it to point to something else than to what it first did. Good use of pointers will speed up the application and when they are used properly there should be no problem. But many people do get confused by pointers and are unable to use them properly. In C++, pointers can be a major source of undetected inconsistencies, which result in failures. C++ is notorious for its pointers and therefore Java uses references. Pointers have their advantages, but one has to be careful in using them.

4 Arrays

Declaring an array in both Java and C++ is very much similar. But also the implementation of an array in both languages has much resemblance. To create an array containing 3 floats, you use the following declaration in C++:

```
float v[3];
```

In Java this is:

```
float[] v = new float[3];
```

In both C++ and Java, the elements of the array can be accessed using the notation `v[0]`, `v[1]` and `v[2]`. C++ allocates storage for 3 floats at compile-time, while Java allocates storage for 3 floats at run-time. So in Java it is possible to specify the size of the array at run-time. In Java the array type is a class; in C++ it is not.

Pointers and references are very closely related to arrays. A C++ array is a pointer to the first element of an array. Java has a reference variable that points to an array, in the given example, `v` is the reference variable pointing to an array of 3 floats.

Perhaps the biggest difference between Java arrays and those of C++ is that C++ does not perform arrays bound checking and Java does. This might result in errors in C++ applications. The downside of array bound checking in Java is that it can report an array exception at run-time and exit the application. But the question remains: what is worse? Exiting an application when array bounds are exceeded or continuing with perhaps the wrong values? Applications are probably easier debugged using array exceptions, because the application explicitly exits and reports an error.

6 C++ functions and Java methods

In C++ functions can be called with two different types of parameters. A parameter is called a *call-by-value* parameter if it behaves like a local variable of the function which initial value is obtained from the argument passed in the call. Consider the following for example:

```
int main() {
  int x = 4;
  f(x);
}

void f(int a) {
  a = a + 1;
}
```

Within the function `f` a local variable `a` is created with an initial value that is equal to the value of `x`. After the function call `a` has value 5 and the value `x` still equals to 4. In C++ it is also possible however to pass a reference to a variable as a parameter. These kinds of parameters are called *call-by-reference* parameters.

```
int main() {
  int x = 4;
  f(x);
}

void f(int &a) { // notice the &
  a = a + 1;
}
```

Now we define `a` to be an alternative name for `x`. They both point to the same location in memory space. Thus by incrementing `a` we also increment `x`, since `a` and `x` are just two different names for the same thing.

It is often stated that Java passes primitive types by value and objects by reference. This should prove that Java supports calling-by-reference. This is a widespread misunderstanding. Java *never* uses call-by-reference. This can be proved

by means of a simple test. Consider a traditional swap method for swapping two integer values:

```
void swap(int a, int b) {
  int temp = a;
  a = b;
  b = a;
}
```

Notice that this does not work in Java because it does not support passing by reference. *Objects are not passed by reference. Object references are passed by value!* However, passing by reference *can* be faked and that is where the confusion comes from. A holder or wrapper object can be passed to the callee for example.

Disallowing passing by reference in Java makes the language easier to use for beginners, but more experienced users may find this a limitation. Working with references (or pointers) on the other hand is extremely error prone and can lead to very obscure, hard to find errors. If it is applied correctly however, one can gain some performance improvement.

7 Memory management

7.1 Garbage Collection

In a good high-level language programmers can declare data without worrying about memory allocation. In C++ however, the programmer must manually manage all memory storage due to the lack of garbage collection. Some have the opinion that this is something that should be implemented in C++, since it is 'a difficult bookkeeping task that leads to two opposite problems.' [1] Here Joyner is referring to the problems with dangling pointers and memory leaks. Dangling pointers occur when objects are deallocated prematurely and valid references still exist. Memory leaks are the opposites of dangling pointers. Dead objects that are 'forgotten', i.e. not deallocated, gather in memory and fill up its resources. Both problems can lead to very obscure and hard to find failures and attempts to correct either of them can lead to overcompensation and the other problem occurring. Nowadays there are tools available such as *Purify* that can detect memory leaks and memory corruption, making the programmer's life easier.

Garbage Collection (GC) is another way to solve these problems and therefore it would be a logical choice to implement it in C++, one would say. As we can see today, much research has been done in this field, but still GC does not form an integrated part of C++. Some say that this is due to performance problems of early garbage collectors, which caused it to have an undeserved bad reputation.

Research is even done on implementing GC in hard real-time systems with promising results. [2] There are some side notes however. The garbage collector must be predictable in both execution time and memory usage. And of course it must also be guaranteed that the system does not run out of its memory because not

enough of it is reclaimed. There are a few garbage collectors that can handle this, but all possibilities have not been thoroughly explored yet.

Java does provide GC and this does greatly reduce the amount of programming effort needed to manage the dynamic data structures. But does this also make Java slower? Opinions differ on this one. Many still clinch to the idea that garbage collection uses up a significant amount of CPU-time. However, in most cases this is poorly grounded. Zorn [3] shows in his paper (dated from 1992!) that some programs that were converted to use a garbage collector, e.g. Perl, ran even faster with a garbage collector than without one. An interesting fact that he pointed out was that both Perl and other applications spent roughly 25-30% of their time in `malloc/free` calls, far more than the allocation and garbage collection overhead of a normal Java application.

Nowadays however, various garbage collectors for C++ can be found on the Internet, but these are not as good as Java's GC. As shown above, there are a lot of advantages with automatic GC. So why do garbage collectors still not form an integrated part of the C++-language? Again, the main problem connects with the backward compatibility of C++ with regular C. Because of the low-level aspects that C++ provides due to this compatibility, programmers can intentionally undermine the structures required for implementing correctly working GC. To circumvent this, programmers should adopt a more restrictive way of programming, but this would mean that the compatibility of C++ with C would be compromised.

7.2 Security

In Java there is no possibility to manage memory manually. There is a sound reason for this. Manual memory management can easily lead to security violations. Since Java applets are often downloaded and run in web browsers, security is an important aspect. If manual management were allowed, it would be possible to write unsafe applets that would be able to collect private information from unsuspecting users and transmit this information back to the writer of the applet.

Concerning memory management, we can say that Java adheres most to the standards of good OO practice. Because GC is there and cannot be avoided, it is always used and makes Java easier to use than C++. If you are a more experienced programmer and need full control over memory management, C++ should be your choice. Because of the higher complexity this is not put away for the inexperienced programmer though.

8 Exception handling

Exceptions provide a way to react to exceptional circumstances in an application by transferring control to special statements, called handlers. The syntax of exception handling in Java and C++ is almost identical. A `try` statement with one or more `catch` statements is used to indicate that some code will handle the exceptions. When an exceptional circumstance happens in a `try` statement, an exception is

thrown that transfers control to the exception handler. The `catch` statements are the exception handlers and take care of the exception.

Java has an extra possibility: in Java, a `try` statement can also have a `finally` statement. This will be executed either after the `try` statement has executed (when there was no exception), or after the `catch` statements (when there was an exception). For example, in the `finally` statement one can put code for freeing resources that were held, when an exception occurs. In C++ one has put this code in the destructor of an object that has been created before the exception occurs. This is a quite inconvenient way and often mistakes are made here leading to errors.

9   Data Abstraction

Data abstraction is a strong, well-defined division between interface and implementation. This greatly improves code reuse and sharing. As long as the interface remains the same the implementation (internal representation) can be modified without changing anything else in the application. The key aspect of object oriented programming languages is that they enable a high level of data abstraction. In Java and C++ *classes* are used to express this. Due to earlier mentioned compatibility issues C++ also features the `struct` type constructor next to classes. A `struct` is the same as a `class` that has `public` components by default. Purely seen, this means that the whole `class type` is redundant and could be removed from C++.

From the two languages, Java practices most OO aspects in a purer way than C++. This is mainly because there is no backward compatibility and Java has been designed from the beginning as an OO language.

9.1   Friends

In C++ class A can access private and protected members of class B if class B declares class A to be a *friend*. Friendship is not inherited. This means that classes derived from class B have no access to these members. Java does not support explicit friends, but does allow classes within the same package to access each other's instance variables. Strictly seen this is not considered good programming practice and OO design, since things can be accessed without going through the published interface of a class.

9.2   C++ templates and Java generics

Many applications use common data structures such as queues, vectors or stacks. The contents of these structures may vary however. There may be a queue of customers and in the next case a queue of packages. Instead of creating a queue for each type it seems to make more sense to implement a queue that can handle arbitrary types. This is called type parameterization and is more commonly referred to as *templates*.

Besides template classes C++ also supports template functions. Below an example of a template function in C++ is given.

```
Template <class AType>
AType min(AType a, AType b) {
  if (a < b) {
    return a;
  } else {
    return b;
  }
}
```

In the past Java didn't support templates or something alike, but this has changed since J2SE 5.0. They are called generics and though they are not exactly the same as C++ templates, they are very much alike. Differences are for instance that Java generics provide compile-time type safety and C++ does not. When a template in C++ is instantiated with a new class, the entire code for the class is reproduced and recompiled whereas Java Generics use *type erasure*. This means that the compiler erases all generic type information, replaces type variables with their upper bound and inserts explicit casts where needed.

10   Inheritance

Inheritance allows classes to be defined in terms of other classes. Each class inherits variable declarations and methods/functions from its superclass. An example of inheritance is a class `horse` that inherits the properties of a superclass `animal`. Both Java and C++ allow inheritance, but C++ also provides so called multiple inheritance. Multiple inheritance means that a class can inherit properties from multiple superclasses. Instead of multiple inheritance Java has *interfaces*. With its interfaces Java implements multiple inheritance with only one important difference: all inherited interfaces must be abstract classes.

There are some drawbacks to multiple inheritance. A well-known example is the 'Diamond of Death'. It refers to the possibility that a class may have the same base class appear more than once as an ancestor. Consider for example the following structure:

```
        A
       / \
      B   C
       \ /
        D
```
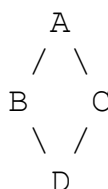
Class `D` inherits from `A` twice; once via `B` and once via `C`. C++ tries to catch this by introducing an extra feature: *virtual inheritance*. If `D` should only have one `A` as a superobject, `A` must be declared as a virtual base class in `B` and `C`. Otherwise `A` can be a normal, nonvirtual base class. This raises questions about what is going to happen

if `A` is declared virtual in only one of `B` and `C`. And what will happen if we define another class `E` that wants to inherit multiple copies of `A` via `B` and `C`? This stresses once again that C++ has unclear semantics and is much more difficult to comprehend than Java.

10.1 Polymorphism

Functions or methods may have more than one definition. Consider for example a Java method `add(int i, int j)` that adds two integers. Now consider a similar function that adds two floating point numbers: `add(double i, double j)`. In this example the method `add` is said to be *overloaded*. Although Java supports overloading of methods it does not support overloading of operators. C++ on the other hand does not have this restriction. A language that supports overloading is called polymorphic.

Another aspect of polymorphism is *overriding* or *shadowing*. This occurs when two methods or functions with the same name, the same number of parameters, and the same parameter types are defined in different classes, one of which is a superclass of the other.

In Java all methods are potential candidates for overriding. In C++ functions must be specified `virtual` in order to allow this. This means that the programmer has to foresee that a descendant class might need to redefine a function. Some consider this a serious flaw in C++ because it reduces the flexibility of the software components and therefore the ability to write reusable and extensible libraries.

As can be read above pleas can be made for and against multiple inheritance. As a sort of workaround Java supports interfaces which can be used to simulate a sort of multiple inheritance. But if the programmer needs the full power of inheritance, C++ is to be recommended. Using multiple inheritance implies that you know what you are doing since there are a lot of pitfalls regarding that field, so it might not be suitable for the inexperienced programmer.

11   Execution speed and portability

How fast an application executes is in many cases an important issue. Also, the ability of an application to execute on different hardware platforms (portability) can also be very important. Java and C++ have specific properties that have a direct impact on both these two important issues.

11.1 Portability

C++ compilers compile the source-code of the application into machine-instructions that can then be executed. Java uses a different approach. In Java, the compiler is called a class-compiler. This class-compiler compiles Java source-code into assembly-like instructions known as Java bytecode. This bytecode can be interpreted and then executed by the Java virtual machine. There are many implementations of

the Java virtual machine for a wide variety of hardware platforms, but all of them support the same bytecode. Because the source-code is no longer directly compiled to machine-instructions, it is possible with the virtual machine to execute the same application on different hardware platforms, even without recompilation. Especially, the Internet has benefited from this, since it exists out of a wide variety of computers with different hardware platforms.

The advantage of C++ of getting easy access to machine level details comes with a down side. By making too much use of this, it is very difficult or expensive to port applications. One of Java's greatest strengths is its portability.

11.2 Execution speed

In early implementations of Java, there was only an interpreter that translated the bytecode to machine-instructions. Since interpretation is slow, Java was unable to compete with C++ regarding execution speed. In general, a Java application executed about 20 times slower than the same application written C++. To handle this problem, Just-in-time compilers were invented. When some bytecode now is translated to machine-code, the Just-in-time compiler keeps a copy of the translated code for potential reuse. This way, a piece of code never has to be reinterpreted a second time. Only new bytecode that has not been previously executed needs to be interpreted. Applications written in Java therefore start relatively slow, but after that, they can achieve speeds comparable to those written in C++.

Benchmarks on the Intel architecture, which is the most used for personal computers, show that in performing numerical calculations, Java is getting very close to C++ [5]. On the Intel architecture and Linux as operating system, a Java application executes about 7% slower than the same application in C++. On Windows, this is about 23%. On other architectures, the gap between Java and C++ is greater. On an Ultrasparc, Java applications execute about 61% slower than equivalent applications in C++, and on the Compaq Alpha, Java applications execute about 4 times slower than equivalent applications in C++. This is mainly due to the greater efforts spent in optimizing Java for the Intel architecture, than for less popular architectures.

We are interested in how Java and C++ compare to each other in large and complex object-oriented applications, but were unable to find any results on this topic. This is probably due to the fact that nobody is willing to spend the time and money in building large and complex applications twice, only to see how they perform relative to each other.

If an application has to execute on different hardware platforms and execution speed is not of the utmost importance, then Java is a very good option to consider. If speed is of the utmost importance and portability is not, then C++ is a better option than Java, especially on non-Intel architectures. One should keep in mind that the execution speed listed above might not count for large and complex object-oriented applications. Just-in-time compilers are relatively new and they have much potential [6].

12 Libraries

Libraries are important when it comes to lowering development times for software. By making use of libraries, the programmer can focus on his application, instead of security, threading, database connection, distributed objects, compression, e-commerce, etc. Java has many API's, which provide a huge set of classes that can be used for these tasks. Such as, database connection, XML, security, threads, graphical user interfaces, sound, many data structures and algorithms, etc. In this case, C++ lies far behind Java, because it relies mostly on non-standard third-party libraries. It does have a standard library providing basic functionality to interact with the operating system and has standard classes, objects and algorithms that may be commonly needed. There are a lot of open-source libraries available for C++, but they often have certain licenses to which one has to comply if one wants to make use of it. This may not need to cause a problem, but there are also many cases where it does cause a problem.

13 Conclusions

After comparing the most important features of the two programming languages, we can conclude that both languages will probably coexist for some years. Java has gained much popularity since it was introduced.

Java is used extensively in enterprise applications. The simplicity, portability, scalability, and legacy integration of the J2EE (Java 2 platform, Enterprise Edition) platform make it very popular for enterprise applications. At this time of writing, hard real-time systems are probably better off with C++, since C++ does not have an unpredictable garbage collector. GC in hard real-time systems is still in research. Also, performance-hungry applications (e.g. 3D-games) are better off with C++ than Java, because they need that extra performance C++ can give.

From a programmer's view, we can conclude that C++ is more complicated than Java. It is not suitable for inexperienced programmers, because errors are made very easily with C++. The C++ standardization committee warns: "C++ is already too large and complicated for our taste". Nevertheless, its high performance makes it still popular today. Java is safer to use and also its many API's relieve the programmer from concerning himself with things like security, threading, database connection, etc. and enable the programmer to focus on the application. Its high portability makes Java ideal for running the same application on different hardware platforms. However, it still cannot meet the levels of performance of C++. On the other hand, Java's performance keeps getting better, so C++ might lose some terrain over the coming years.

References

[1] Ian Joyner, *C++??: A Critique of C++ and Programming and Language Trends of the 1990s* (3rd edition), 1992.

[2] Tobias Ritzau, *Memory Efficient Hard Real-Time Garbage Collection,* 2003.

[3] Benjamin Zorn, *The Measured Cost of Conservative Garbage Collection Software - Practice and Experience* 23(7): 733-756, 1992.

[4] David L. Shang, *Transframe, Java & C++: A Critical Comparison*, 1996.

[5] J.M. Bull et al, *Benchmarking Java against C and Fortran for scientific applications*

[6] Kirk Reinholtz, *Java will be faster than C++*

[7] Barry Cornelius, *Java versus C++*

Tree-based Image Representation, Filtering and Segmentation

Joris Best(s1494848), Roel Donker(s1492144)

Abstract. Images can be stored in many ways, one of the most promising way is storing an image in a tree-structure. Processings on an image means in this case processing the tree-structure which represents the image. The use of a tree-structure makes processings on a tree both simple as efficient. This paper describes three different ways of representing images by a tree-structure. The three representations being researched in this paper are: "binary partition tree", "component tree" and "foresting transform". The last section of this paper evaluates the efficiency of all three tree-structures related to image-processing (especially filtering and segmentation).

1   Introduction

There are many different ways to represent images in a tree-structure. But each way has its own advantages and drawbacks. The problem that occurs is that for example an image-processing technique is easy to perform on one tree whereas it is difficult to do it on another tree. This can be explained by looking to the processing techniques like segmentation and filtering which are used to process an image and building such a tree. We want to explain which method to represent an image in a tree is best for a particular image-processing technique. This is done by comparing the methods "binary partition tree", "component tree" and "foresting transform". The first two methods are region-based and the foresting transform stays on the pixel-level.

The next section explains how to build such trees. Section three handles the image-processing techniques "filtering" and "segmentation" in relation to these trees. After that the major advantages and drawbacks are summarized followed by a conclusion in section four.

2   Methods

2.1   Binary Partition Tree

A binary partition tree is constructed by using two techniques which are used in the fields of image processing. These techniques are called connected regions and

segmentation. In this section we will explain what these two techniques involve and after that we will explain how a binary partition tree can be computed.

Connected regions. Connected regions are some kind of filters which can be used to describe a piece of an image which is constant in a certain way. That means that pixels in an image which are lying next to each other and have the same color or gray-value are merged into a region. In 3-D representations the filter could merge those pixels which are not only lying next to each other, but are also constant in a certain direction (for example the x-component of the pixels is constant). Most times a connected region is used in combination with certain rules, for example a region consisting of merged pixels may not be smaller or greater than 500 pixels.

Segmentation. Segmentation techniques are used to merge pixels and regions (note that regions are a group of pixels which are similar in some way). These pixels or regions are related to each other with so called links. The merging of pixels and regions (linking) relies on three notations, namely: merging order (the order in which the links are merged), merging criterion (the criteria on which is decided if something has to be merged or not) and the region model (when a merging has taken place, this defines how to represent the merged regions).

Construction of a Binary Partition Tree. A binary partition tree can be constructed using filters (connected regions) and segmentation. The filters are responsible for making regions and segmentation can be used to merge these regions into a binary tree. In this way the nodes of the tree are representing both the segmentation steps that are taken and a region. The leaves of the tree are only representing regions. It appears to be clever to put large regions close to the root of the tree, while the details of an image are put into the leaves (merging of regions, efficiency, information retrieval, etc.). The biggest difficulty of constructing the tree is the criteria on which the merging of the regions takes place. If we take a picture of a face with a blue hat on top of it and on the background a blue sky. If the hat has approximately the same color as the sky, then the hat could end up in a different part of the tree than the face. Obviously this may result in various problems like shape recognition, information retrieval, etc. An example of a constructed Binary Partition Tree is given in Appendix "Binary Partition Tree".

2.2 Component tree

The Component tree is a structured representation of a gray-level image. The component tree was developed to deal with properties such as anti-extensivity, idem potency, increasingness and utilizing links. The component tree can only represent binary or gray-level images. In order to construct the component tree, the image can be considered as a 3D image. Each node in the tree is a representation of a component at a specific gray-level. A component can be defined by the following notation which

is different from a flat-zone (A flat-zone is a connected set of pixels with the same gray value):

$$C_{t,n}: C_{t,n} \text{ is the nth component in } X_t(f); t \text{ is any gray-level in } f \qquad \textbf{(1)}$$

Where $X_t(f)$ is a threshold set at threshold t:

$$x \in F: f(x) \geq t; F \text{ is the gray-level image grid}(f \text{ maps each image coordinate} \qquad \textbf{(2)}$$
$$\text{in the grid F to a single gray level}); t \text{ is the threshold value}$$

Summarized briefly: a component $C_{t,n}$ in a image f is defined as a connected set of pixels in a threshold set $X_t(f)$. An important property of the threshold set is that $X_{t+1}(f)$ is a subset of $X_t(f)$. This means that there exists some component $C_{t,m}$ for every component $C_{t+1,n}$. These two components ($C_{t,m}$, $C_{t+1,n}$) are linked with each other, where $C_{t,m}$ is the parent and $C_{t+1,n}$ is the child. The root node ($C_{min,n}$) at the minimum gray-level is a superset of all the components in the image. A component that is not linked to a component with a higher gray-level is called regional maximum (leaves). As said earlier the node is an abstract representation of a component. For the filtering and segmentation (see 3 Segmentation and filtering) a node only needs to store four properties.
1. Gray-level of the component
2. Location of one of the pixels within the component
3. The value of some attribute of a component (area, perimeter, etc.)
4. Is the component active or not?(discussed in section segmentation and filtering)
How long is the path from a leave node to the root node? In the component tree this path is called a branch. A branch is defined as the shortest sequence of linked nodes from a leave node to the root node. The path cannot be longer then the maximum gray-level + 1, because root node has gray-level 0. Many other types (min tree, opening tree and max tree) of trees look almost the same as the component tree according to Jones[2] but they are not, especially in relation to connected filters. An example of the component tree is illustrated in figure 1.
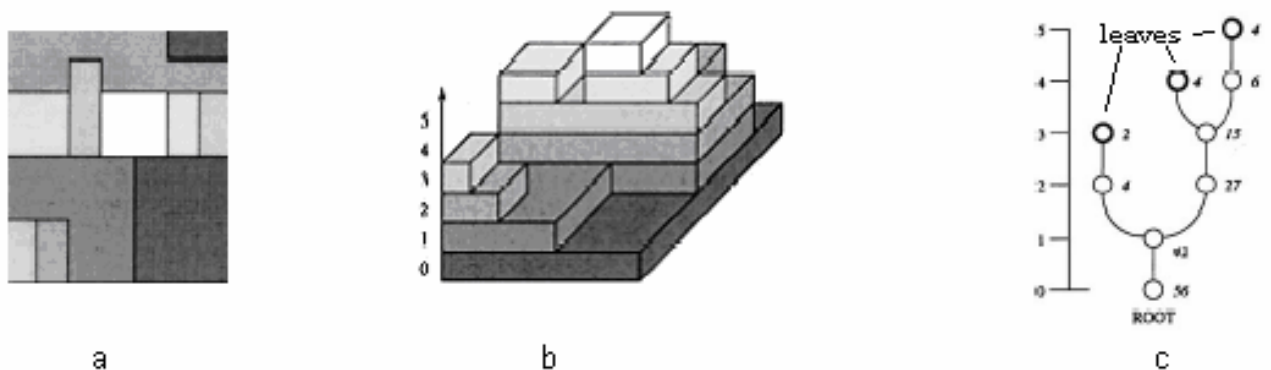
Fig. 1. Construction of a component tree: a) gray-level input image b) perspective view of threshold sets (3D image) c) Component tree.

The figure 1a above contains nine components (same as number of nodes in the tree) and has six different gray-levels. This is different from the number of flat-zones

because that is eleven, so a component tree is less complex and smaller then a binary partition tree. The number of branches is three because there are only three leaves in the tree. The lines in the tree indicate the links between the nodes in the tree. The gray-levels stored in the nodes are displayed on the left side of the tree. As example of an attribute value stored in a node, the component area is shown as a number by the node. A second example of a Component Tree is given in Appendix "Component Tree".

2.3 Image Foresting Transform

The IFT is meant to unify and extend existing image analysis techniques. In an IFT every pixel of the image is a node. Every node is connected to its surrounding nodes and has a certain attribute value (colour-value, grayscale, etc.). An image forest consists of several trees. The number of trees depends on the number of seeds. The seeds are the nodes with the minimum attribute value. The values are given to the nodes by using an image analyses technique. A drawback of this manner is that we have complex and large trees in the forest.

We can construct a tree with a seed as root by looking at the connectivity and the minimum cost path (extended Dijkstra algorithm) between the nodes. One starts looking at the root of the tree and chooses a surrounding node, the node with the lowest attribute value will be the child of the parent (in this case the parent is the root). It may be obvious that this is the minimum cost path. Because there are multiple seeds in the image it is necessary to have some rules on claiming a node. Consider the situation that there are two or more nodes claiming the same node. For example we could use the rule that the first claim made is the one that gets connected. Furthermore FIFO or LIFO could also be used (see example in figure 2). A real example of an IFT Forest is given in Appendix "Image Foresting Transform".
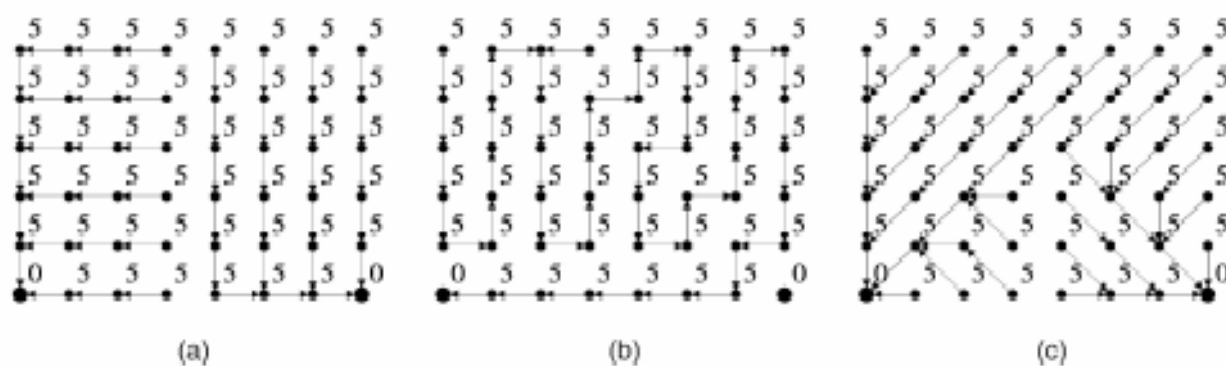
Fig. 2. The values above the nodes represent the attribute value of the pixel. (a) FIFO policy and 4-connected adjacency. (b) LIFO policy and 4-connected adjacency. (c) FIFO policy and 8-connected adjacency.

3   Filtering and segmentation

3.1   Filtering

Filtering techniques are derived from mathematical morphology. A filter remove parts of the input image content, however the remaining parts will contain their contour information. Summarized briefly: A filter is able to remove or merge regions.

Binary Partition Tree. Filtering an image means that one throws away the details which are not interesting. In the case of a binary partition tree this means that throwing away a region results in throwing away a node. But throwing away a node means that all its children are also thrown away. If this is not a problem then the filtering algorithm will be very efficient (since it does not have to filter the complete tree). This is often the case when the tree is built up in an increasing way. For example each node consists of more pixels than its siblings. On the other hand we have the case in which it is undesirable when a sibling is also removed. Then the filtering algorithm can use a so called Viterbi algorithm. Shortly said the Viterbi algorithm is an algorithm which tries to find the most efficient way through a problem. We will not explain the algorithm in this paper, the interested reader can easily find articles about this subject.

Component Tree. Filtering of the component tree is very efficient and simple. A tree node will be pointed out as active (see section "2.2 Component tree") if the component is to be preserved by the filter. The root node of the tree is always active. The success of filtering a tree depends on the decision we have made related to the type of attribute used for the filtering. There are two kinds of component tree filters: The flat image filters and the more general non-flat image filters.

- flat image filter
  A flat image filter can be notated by the following definition:
  "A node with a certain number and gray-level value is active when it satisfies a certain criterion".
  The flat image filters ignore the links between the nodes. The problem which can arise (because of ignoring the links) is that it is possible to have gaps between active nodes. An ugly solution is to make all the nodes from n to the leaves "not active".
- non-flat image filter
  A non-flat filter is based on the concept of an attribute signature. An attribute signature is the sequence of node attributes in a branch. The non- flat image filter can be notated by the following definition:
  "A leaf is active if the branch (see section "2.2 Component tree") satisfies a certain criterion"
  The complexity of this kind of filters is hidden in the criterion. To ensure that there are not gaps between active nodes the whole branch of a leaf must be active.

The following definition is the general gray-level filter of a filtered component tree:
"Look for the maximum gray-level value node in the tree of a specific region which is active".
In many cases it is not possible to get the right result with one attribute signature (area, eccentricity, etc.) in the criterion. A solution in most cases is to combine several signatures, but sometimes even this will not help. Example: if you have a bright background and the object is bright, then the filter cannot separate the two regions.

Image Foresting Transform. Earlier in this paper we explained that an IFT consists of multiple trees. Each tree represents a region of the original image. The shape of a region can be derived from the leaves of a tree (since the leaves are forming the border of the region). So when a threshold (threshold is based on the attribute value) is performed a region shrinks. The threshold method is particularly used to recognize a certain shape in an image.
The seeds represent the most interesting pieces of the image. Because the seeds are forming the roots of the trees. So the places of the image that are interesting to look at are most likely found in the seeds. Keep in mind that the seeds were found by some kind of filtering technique.
Imagine that we are performing a threshold between two values. Then it is possible to have a gap between two nodes. So the whole tree has to be rebuilt, which is very inefficient.

3.2   Segmentation

Splitting an image into several regions which are related to each other is known as image segmentation. Each region is homogeneous to a certain property and is labeled. As called earlier image segmentation can be used for image representation and interpretation (see section "2 Methods").

Binary Partition Tree. In section "2.1 Binary Partition tree" we already pointed out that segmentation is used to construct a tree. Now we show that a binary tree representation of an image can be used to generate segmentation results. This can be done in a simple way by merging the regions that are pretty much the same and in a hard way by giving certain regions a marker. The simple way is kind of straightforward and could result in the problem where regions are divided into different sub trees while one does not want that to happen.
We can give regions a marker automatically or manually. In this paper we will not go into the algorithms that give markers to regions automatically, but giving markers (automatically or manually) gives a simple solution to the problem of the blue hat. Let's take again the blue hat example of section "2.1.3 Construction of Binary Partition Tree" where we had a picture of a face with a blue hat and a blue sky on the background. Now we give the hat (assuming that it consists of one color) a marker and the regions which represent the face also a marker. Now the merging is done based on the markers of the regions resulting in a tree in which both the face and the

hat are in the same sub tree. Keep in mind that unmarked regions will be put into another sub tree, like the sky in this example.

Component Tree. The component tree can be used for image segmentation as well for image filtering (see section "3.1.2 component tree"). Segmentation means to point out one or more regions of an image (example: only display the face of a human). The segmented image in the component tree is a binary image, which marks only those regions that were changed by filtering. The exact definition is:
"If the original region (pixel) has the same value as the filtered region (pixel) then that region (pixel) get the gray value zero else the gray value is one"
A great advantage of this definition is that it is really simple to implement as a fully automated procedure to segment the interesting regions. Furthermore this approach lends itself well to filtering based on attribute signatures.

Image Foresting Transform. In section "2.3 Image Foresting Transform" we explained that an image analysis technique can be used to give a certain value to the pixels (nodes). For example we want to analyze if a pixel is dark or bright. If it is bright it will get a high value and when it is dark it will get a low value. Let's flow water into this "map" of values, starting at the low values (these pixels forms the seeds). The water will flow to the lowest values in the neighborhood (lowest cost path). If two flows meet each other then it will stop flowing at that point. This is called the Watershed transform and forms the base of the segmentation, which is used to build an IFT. The detection of boundaries and shapes is in this case easily because the leaves are the boundaries. A drawback of this method is that we have to analyze several trees (with a lot of nodes), which will take more time to analyze the trees in contrast to the other methods.

3.3   Discussion

The aspects related to filtering, segmentation and building a tree have been discussed in the sections above. These aspects can be compared to each other. In table 1 the three discussed methods are shown and scored on their performance, complexity, tree-size, etc.

	Binary partition tree	Component tree	Image foresting transform
Tree complexity	--	++	+/-
Tree size	+	++	--
Time to perform an operation on the tree	+	++	-
Automatic segmentation	--	++	++
Shape recognition	+	--	++
Anti-extensive filters	+	+	-
Scalability	+	+	-

Human made decisions	--	-	+

Table 1. Comparing the aspects of three image-trees

In the next section the scores in table 1 are elaborated.

4   Conclusion

Based on the arguments in the last sections and the scores that are given in table 1 of section 3.3 the following conclusions can be made:

Binary partition tree: The segments that form this tree have underlying relations, which could not be seen by an algorithm, for example the tree builder want to have a certain part(e.g. head) of the image in the right side of the tree. Therefore it is necessary to monitor the building process by a human. Resulting in a complex tree, which is very efficient in later filtering techniques.

Component tree: In contradiction to the binary tree, the component tree is easy to build. But the success of filtering depends on the complexity of the filter and human made decisions (area, eccentricity, etc.).

Image foresting transform: This tree stays on the pixel level, resulting in a tree with more nodes than the other trees. The details of the image are preserved, but because of the tree-size all later operations will take long to perform. Boundaries of an image are easy to find in this tree, since the leaves of a tree forms the boundary. Because of this shape-recognition is very easy.

We can make the final conclusion that the component tree method is the simplest one. The image foresting transform is the opposite of it (since it is more complex). The binary partition tree hangs between the other two trees and its efficiency is based on the human role played during the building of the tree. If time does not play a role, one can choose to use the image foresting transform. Since it leaves the details intact.

References

1. Alexandro X. Falco, Jorge Stolfi, Roberto De Alencar Lotufo: The Image Foresting Transform: Theory, Algorithms and Applications. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 26, No. 1, January 2004.
2. Ronald Jones: Connected Filtering and Segmentation Using Component Trees. Computer Vision and Image Understanding Vol. 75, No. 3, September 1999(215-228).
3. Philippe Salembier and Luis Garrido: Bainary Partition Tree as an Efficient Representation for Image Processing, Segmentation an Information Retrieval. IEEE Transactions on Image Processing, Vol 9. , No. 4 , April 2000.

Appendix: Binary Partition Tree
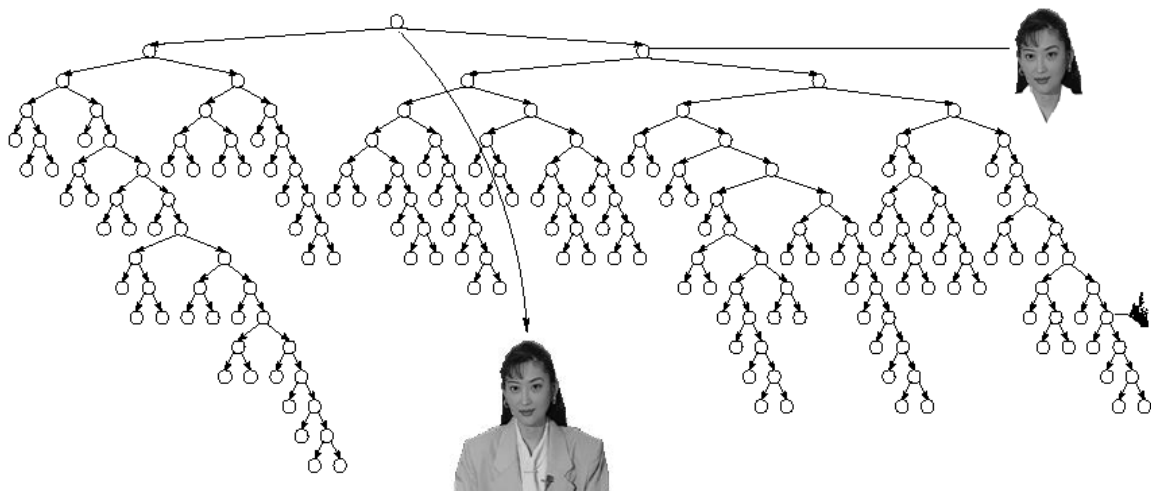
Appendix: Component Tree
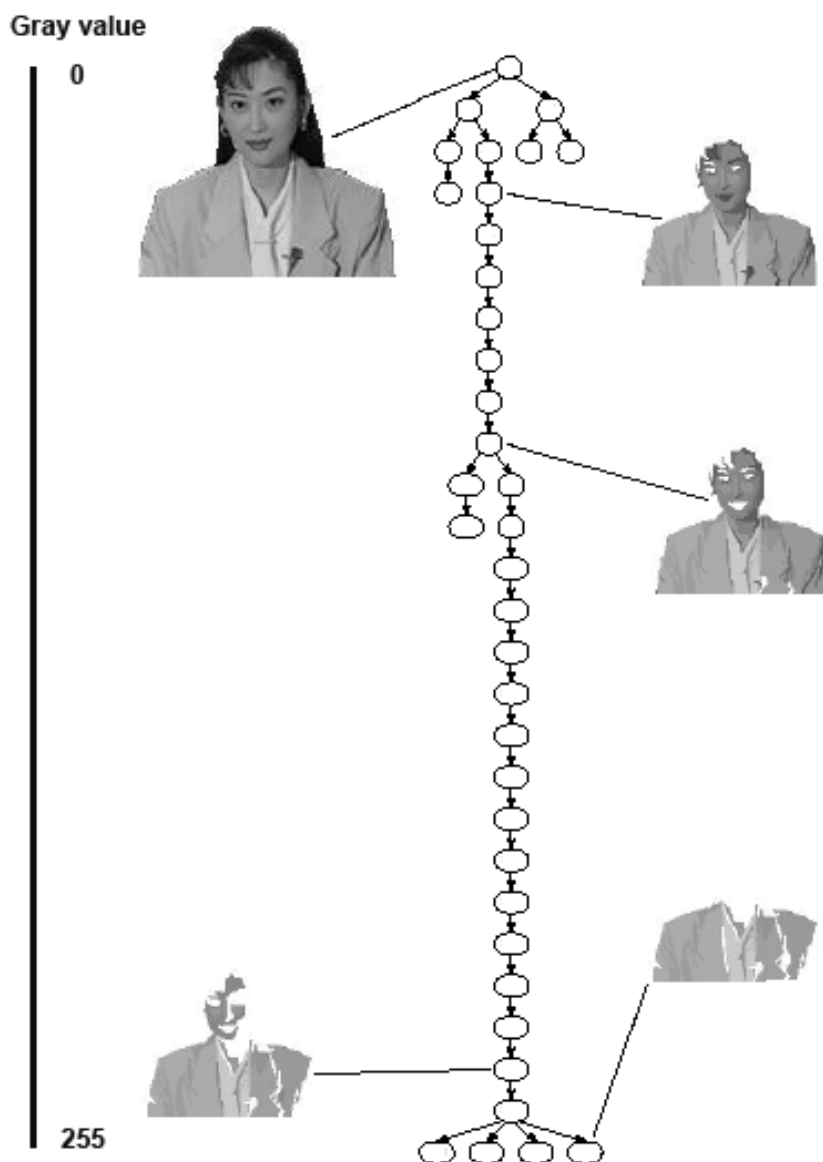
Appendix: Image Foresting Transform
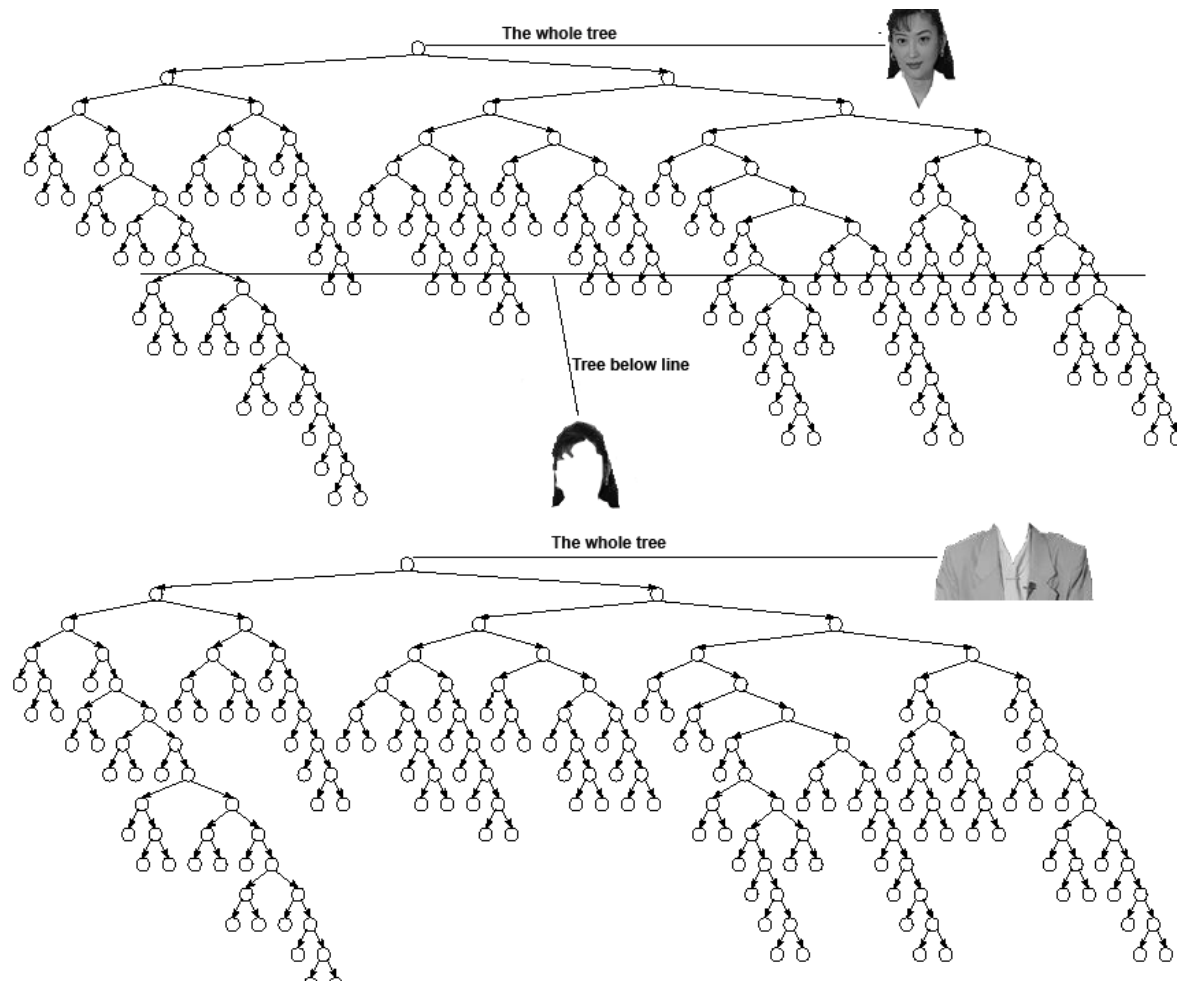
(In)security of the Needham-Schroeder public-key protocol

Freek Vandeursen   [f.vandeursen@student.rug.nl]

Mark Speelman   [m.speelman.1@student.rug.nl]

Abstract.

The role of security in computer communication is greatly increasing since more and more important communication takes place over computer networks, e.g. the Internet. The starting point of secure communication is the authentication, verifying the identity of the participants. Authentication can be established with the Needham-Schroeder protocol. This protocol was first published in 1978 by Roger Needham and Michael Schroeder, and it took 17 years before it was broken by Gavin Lowe. Lowe proved that this protocol could be broken by a "man-in-the-middle" attack. In 1990 Gong, Needham and Yahalom published an article in which they presented a method to reason about security protocols. This method emphasizes the reasoning about what the participants in the protocol know and belief. We will show that the vulnerability found by Lowe could also have been found using this method from Gong, Needham and Yahalom, and subsequently we show that the improved protocol is secure. The fact that the vulnerability was not spotted before shows how difficult reasoning about security protocols is, and how difficult it is to guarantee full security.

1. Introduction

In the world of computer communications security is getting more and more important. Since communication tend to take place over insecure media like the Internet, it is important to shield your messages from others than the intended receiver.

The starting point of secure communication is the authentication. The participants in the communication (Alice and Bob) have to be sure they are talking to each other and nobody else. Authentication can be established with the Needham-Schroeder protocol.

This protocol was first published in 1978 by Roger Needham and Michael Schroeder [NS78], and it took 17 years before it was broken by Gavin Lowe [Lowe96]. This long time is typical for the subtle and hard-to-spot vulnerabilities of security protocols. Lowe proved that this protocol could be broken by a "man-in-the-middle" attack. He introduced a third participant in the communication; the intruder (Trudy). Trudy communicates with Alice and Bob, but Bob doesn't know he's communicating with Trudy, because Trudy makes him think she is Alice. Lowe also presents a solution to this problem as described in the section "Methods and materials".

We will show that this same problem can be discovered with the method that Gong, Needham, and Yahalom introduces in 1990 [GNY90], 5 years before Lowe broke the Needham-Schroeder public-key protocol.

The rest of the paper is organized as follows. In the next section we will outline the methods and material used to come to our results that are presented in section 3. Section 4 will contain a discussion; comparing the results of Lowe and ourselves.

2. Methods and materials

2.1 Notation for messages

We will use the following notation for messages:

$A \rightarrow B : X$ A sends message X to B.

$A \rightarrow B : \{X\}_{+KB}$ A sends message X to B, encrypted with the public key of B

$A \rightarrow B : \{X\}_{-KA}$ A sends message X to B, encrypted with the secret key of A

2.2 Notation for reasoning

We make use of the reasoning mechanism as described by Gong, Needham and Yahalom [GNY90] in order to prove the insecurity of the N-S protocol [NS78]. This mechanism allows us to reason about the beliefs associated with the protocol.

We use the following notations:

$P \triangleleft X$ P is told formula X. In other words: P has received X, possibly after decryption.

$P \ni X :$ P possesses formula X. This includes all the formulae available at the start of a session, all the formulae generated by P, and all the formulae received by P.

$P \mid\sim X :$ P has once conveyed formula X. So P has once sent X to someone else.

$P \mid\equiv S :$ P believes statement S.

$P \mid\equiv \phi(X) :$ P believes that formula X is recognizable. In other words: P would recognize X when X is received from someone else.

$P \mid\equiv \#(X) :$ P believes formula X is fresh. This means that X has not been used before for the same purpose. Some articles refer to X as a "nonce".

$P \mid\equiv \overset{+K}{\mapsto} Q :$ P believes that K is a suitable public key for communication with Q.

3. Results

3.1 Description of the protocol

The Needham-Schroeder public-key protocol normally has 2 or 3 participants: **A**lice, **B**ob and possibly Uncle **S**am (a trusted authentication server).

The protocol starts with the message *A,B* from Alice to Uncle Sam, meaning: "I am Alice and I want the public key from Bob".:

1) $A \rightarrow S : A, B$

Uncle Sam returns the public-key of Bob, encrypted with his own private key. Because of this encryption Alice can be sure that the message really originated from Uncle Sam.

$$2) \, S \rightarrow A : \{+KB, B\}_{-KS}$$

Now Alice sends a message to Bob, to indicate that she wants to start a communication. In order to do this she sends a *nonce*, a unique number which has not been used before in previous sessions. The message is encrypted with the public-key from Bob, so that Alice knows that only Bob can read it.

$$3) \, A \rightarrow B : \{N_A, A\}_{+KB}$$

Now the first two steps are repeated, as Bob needs to know the public key of Alice.

$$4) \, B \rightarrow S : B, A$$
$$5) \, S \rightarrow B : \{+KA, A\}_{-KS}$$

With this key present, Bob can respond to Alice. In order to prove that he really is Bob he returns the nonce sent by Alice. Furthermore he sends a new nonce so that Alice can prove that she really is Alice. In order to prove this, the message is encrypted with Alice's public key.

$$6) \, B \rightarrow A : \{N_A, N_B\}_{+KA}$$

The final step is the verification of Alice's identity, by returning the nonce from Bob.

$$7) \, A \rightarrow B : \{N_B\}_{+KB}$$

If Alice and Bob store the public-keys of each other then Uncle Sam can be left out of the communication. Only messages 3, 6 and 7 remain.

3.2 Gavin Lowe

In 1995 Gavin Lowe showed that this protocol was not as secure as everyone believed [Lowe96]. He showed that an intruder (we call her **T**rudy) could fake part of the authentication process by faking the identity of one of the participants. For simplification we assume that the public keys are known, so we leave out the authentication server.

The security breach starts with Alice who starts a conversation with Trudy:

$$1a) \, A \rightarrow T : \{N_A, A\}_{+KT}$$

Instead of responding Trudy starts a conversation with Bob, while pretending to be Alice (notated T(A)).

$$1b) \, T(A) \rightarrow B : \{N_A, A\}_{+KB}$$

Bob has no means of detecting the fraud, so he sends a normal response to Trudy.

$$2b) \, B \rightarrow T(A) : \{N_A, N_B\}_{+KA}$$

Trudy can not decrypt this message, since she does not possess the secret key of Alice. Therefore she forwards the message to Alice, pretending to have written the message herself.

2a) $T \rightarrow A : \{N_A, N_B\}_{+KA}$

Alice decrypts the message and returns the nonce from Trudy (that originated from Bob).

3a) $A \rightarrow T : \{N_B\}_{+KT}$

Now Trudy has received Bob's nonce, so she sends it to Bob while pretending to be Alice.

3b) $T(A) \rightarrow B : \{N_B\}_{+KB}$

Now the authentication has finished. Bob is under the impression that he is communicating directly with Alice while in fact Trudy is standing between them.

3.3 Reasoning

In 1990 Gong, Needham and Yahalom published an article in which they presented a way to reason about security [GNY90]. We will use their method to show how the insecurity of N-S protocol could have been discovered.

We start with the initial knowledge. In short: both Alice and Bob have a nonce which they believe to be fresh and recognizable. Furthermore they have their own keys and believe them to be suitable keys. And finally they possess the public keys of each other, which allows us to leave out Uncle Sam (the authentication server).

$$A \ni N_A, \quad A \mathrel{|\!\equiv} \#(N_A), \quad A \mathrel{|\!\equiv} \phi(N_A)$$
$$A \ni +KA, \quad A \ni -KA, \quad A \mathrel{|\!\equiv} \stackrel{+KA}{\longmapsto} A$$
$$A \ni +KB, \quad A \mathrel{|\!\equiv} \stackrel{+KB}{\longmapsto} B$$
$$A \mathrel{|\!\equiv} \phi(B)$$

$$B \ni N_B, \quad B \mathrel{|\!\equiv} \#(N_B), \quad B \mathrel{|\!\equiv} \phi(N_B)$$
$$B \ni +KB, \quad B \ni -KB, \quad B \mathrel{|\!\equiv} \stackrel{+KB}{\longmapsto} B$$
$$B \ni +KA, \quad B \mathrel{|\!\equiv} \stackrel{+KA}{\longmapsto} A$$

Now Alice starts the communication with her first message. Reception of this message by Bob is formally described as:

$$B \triangleleft \{N_A, A\}_{+KB}$$

Since the message is encrypted with Bob's public key, Bob is able to decrypt the message. Therefore we can add the contents of the message to Bob's knowledge.

$$B \ni \{N_A, A\}_{+KB}, \quad B \ni -KB \quad \Rightarrow \quad B \ni N_A, \quad B \ni A$$

Now we know that Bob possesses the nonce from Alice, his own nonce and the public key from Alice. These are all components needed for the next message.

$$B \ni N_A, \quad B \ni N_B, \quad B \ni +KA \quad \Rightarrow \quad B \ni \{N_A, N_B\}_{+KA}$$

Now Bob sends his response to Alice.

$$A \triangleleft \{N_A, N_B\}_{+KA}$$

Upon receiving the message Alice decrypts it. She recognizes her own nonce, and adds Bob's nonce to her knowledge. Because she recognizes her own nonce she knows that Bob has received her message since he was the only one who could have decrypted that message.

$$A \ni \{N_A, N_B\}_{+KA}, \quad A \ni -KA \quad \Rightarrow \quad A \ni N_A, \quad A \ni N_B$$
$$A \triangleleft N_A, \quad A \models \phi(N_A) \quad \Rightarrow \quad A \models B \ni N_A$$

At this point Alice should be sure of Bob's identity. However, a summary of Alice's current knowledge shows the contrary.

$$A \ni N_A, \quad A \models \#(N_A), \quad A \models \phi(N_A)$$
$$A \ni +KA, \quad A \ni -KA, \quad A \models \overset{+KA}{\mapsto} A$$
$$A \ni +KB, \quad A \models \overset{+KB}{\mapsto} B$$
$$A \models \phi(B)$$
$$A \ni N_B, \quad A \models B \ni N_A$$

In short: Alice knows that Bob received her message, but she has no knowledge of the origin of the last received message.

Gavin Lowe encountered the same problem using the Failures Divergences Refinement Checker (FDR). Because the problem consists of uncertainty about the sender, Lowe proposed to include a reference to the sender in the second message.

$$B \rightarrow A : \{N_A, N_B, B\}_{+KA}$$

So now we can add the sender to the knowledge of Alice. Since Alice is able to recognize B (as stated in the initial knowledge) she will recognize Bob as the intended receiver.

$$A \triangleleft B, \quad A \models \phi(B) \quad \Rightarrow \quad A \models B \mid\sim \{N_A, N_B, B\}_{+KA}$$

So at this point Alice is sure about Bob's identity. We only need to make sure that Bob also is sure about Alice's identity. That is accomplished with the third message. Please note that a sender is not required in this message (as it was in the first two). Bob already knows who Alice is supposed to be ($B \ni A$), he only needs to confirm it.

$$B \triangleleft \{N_B\}_{+KB}$$

So Bob decrypts the message and recognizes his own nonce.

$$B \ni \{N_B\}_{+KB}, \quad B \ni -KB \quad \Rightarrow \quad B \ni N_B$$
$$B \triangleleft N_B, \quad B \models \phi(N_B) \quad \Rightarrow \quad B \models A \mid\sim \{N_A, A\}_{+KB}, \quad B \models A \mid\sim \{N_B\}_{+KB}$$

So now both Alice and Bob know whom they are talking with, and can start their real conversation.

4. Discussion

Using methods like Lowe's FDR [Lowe96] and Gong, Needham and Yahalom's reasoning method [GNY90], it is possible to find vulnerabilities in security protocols. But because the subtle and hard-to-spot vulnerabilities of security protocols it is not guaranteed you will actually spot the vulnerabilities. Because we knew the vulnerability of the N-S protocol from Lowe's research, we were able to spot it using the reasoning method of Gong, Needham and Yahalom, but without that prescience it would have been hard to spot. So these reasoning methods give a fair indication about the vulnerability of a security protocol, but the question remains in which degree these methods can guarantee full protection.

5. References

[GNY90] Gong, Needham, Yahalom, Reasoning about beliefs in cryptographic protocols, 1990
[Lowe96] Lowe, Breaking and fixing the Needham-Schroeder public-key protocol using FDR, 1996
[NS78] Needham, Schroeder, Using encryption for authentication in large networks of computers, 1978

The (in)correctness of a security protocol

Gerard Knap and Bart Hoenderboom

Rijksuniversiteit Groningen
`csg0006@wing.rug.nl,csg0031@wing.rug.nl`

Abstract. In this paper we explain the mutual authentication protocol from Needham and Schroeder. Although in the beginning the protocol was assumed to be correct, it was later shown that there was a crucial weakness in it. The protocol was vulnerable for the so called "man-in-the-middle attack". Different adaptations to the protocol are proposed to remove this weakness. The vulnerability is superficially analyzed and an improved version is given. Finally it is shown that the new version of the protocol is more secure. The importance of systematic analysis is also covered.

1 Introduction

Different agents in a distributed computer system should be able to authenticate each other. Because of the sensitive information they exchange, agents want to be sure that they don't communicate to an intruder impersonating the other agent. Therefore they make use of an authentication protocol. In 1978 Needham and Schroeder proposed such an authentication protocol [1]. The Needham-Schroeder Public-Key Protocol was assumed to be correct. In 1987 however a revised protocol was proposed by Needham [3]. In the years after several scientists tried to develop a formal logic to reason about authentication protocols. Around 1989 the development resulted in something which is referred to as BAN logic, called after the developers Burrows, Abadi and Needham. The BAN logic inspired several scientists to formally analyze the Needham-Schroeder protocol. In 1990 an article was published by Gong in which the Needham-Schroeder protocol was analyzed with a new method inspired by the BAN logic [5]. It was one of the first articles which actually proved the correctness of the improved protocol. In 1996 Gavin Lowe proved again that the protocol was vulnerable for the man-in-the-middle attack [6]. He proved it with a new method, CSP (language developed by C.A.R. Hoare) and FDR (a model checker for CSP).

This paper shows how the Needham-Schroeder protocol works with a Key Distribution Center and also with public keys. The original protocol was revised because of a weakness in it. We will explain the weakness and show how this weakness can be exploited by a man-in-the-middle attack.

Since this article is a review we will omit the logic reasoning from the reviewed articles. In those articles different notations and methods are used, which is out of the scope of this article. Gavin Lowe used in [6] CSP in combination with FDR to prove that his improved version is secure. CSP and FDR are completely

described in [2] and [4] respectively. In chapter 4 of this article there is however a short description of what FDR and CSP is. Finally we discuss our findings.

2 Man-In-The-Middle Attack

This term is generally used in cryptography for an attack in which an attacker is able to read, insert and modify at will, messages between two parties. Both parties are unaware of this attacker. The attacker must be able to observe and intercept messages going between the two victims. The first version of the Needham-Schroeder protocol [1] was vulnerable for this type of attack. In this paper the protocol is explained and the vulnerability for a specific man-in-the-middle attack is shown.

3 Needham-Schroeder Protocol

The Needham-Schroeder protocol can be used for establishing mutual authentication between an initiator A and a responder B. The first version of the protocol had a serious weakness. Needham revised his protocol in [3] to remove this weakness. Both the original version and the revised version will be explained in this chapter.

The protocol can be used in combination with different key systems, with the Key Distribution Center or with public keys. Because the revised version of the protocol has a subtle difference depending on the key system used, both approaches are described.

3.1 Authentication with a Key Distribution Center

The basic principle of authentication with a Key Distribution Center (KDC) is to exchange unique numbers between A, B and a secure server. This server, called Key Distribution Center, shares a secret key with each of his clients. The role of the KDC is to provide the keys which will be used by A and B for encrypted communication. The unique numbers exchanged are called nonces. Every session requires a new nonce. To prevent others from reading those nonces, encryption is used. To authenticate each other the nonces will be manipulated. This must be done in a predefined way. In this example the nonce is decreased by 1. When the nonces are correctly manipulated by both A and B, they have established each other's identity.

3.1.1 Initially proposed protocol

The first version of the Needham-Schroeder protocol [1] will be showed in figure 1. It shows the five steps of the original protocol. After successful completion of the five steps, A and B have authenticated each other.
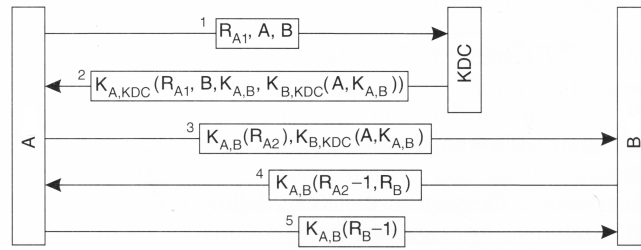
Fig. 1. First version of the Needham-Schroeder protocol.

A sends a request to the KDC containing a nonce (R_{A1}) and the identity of A and B. The KDC responds with a message. This message is encrypted with the key that A shares with the server. This message contains:

- the nonce sent by A (R_{A1})
- the identity of B
- the shared key between A and B ($K_{A,B}$)
- an encrypted message containing the identity of A and the shared key between A and B. ($K_{B,KDC}$(A,$K_{A,B}$))

A sends a message of two parts to B. The first part contains a new nonce (R_{A2}) encrypted with $K_{A,B}$ ($K_{A,B}(R_{,A2})$). The second part of this message contains $K_{B,KDC}$(A,$K_{A,B}$). B decrypts message $K_{B,KDC}$(A,$K_{A,B}$) and uses the shared key $K_{A,B}$ to decrypt message $K_{A,B}(R_{A2})$. B sends a response R_{A2}-1 along with a new nonce R_B ($K_{A,B}(R_{A2}$-1,$R_B)$). A decrypts this message and responds with R_B-1 ($K_{A,B}(R_B$-1)).

3.1.2 Weakness in the protocol

The Needham-Schroeder protocol, as presented above, is vulnerable for the man-in-the-middle attack. The first two messages are not linked to message 3. This means that the first two messages could be skipped and an intruder could send a previous intercepted message 3 in the protocol. In the rest of this article the sending of an previous intercepted message is referred to as replaying a message. If the intruder also has the shared key between A and B, he can perform a fake authentication with B, while B thinking he is communicating with A. This weakness can be fixed by linking step 3 to the first part of the protocol. This link makes it impossible to skip the first messages. If the intruder replays message 3, the message will be invalid.

3.1.3 Revised Version by Needham

The improved protocol will be expanded with two more messages which establishes the needed link.
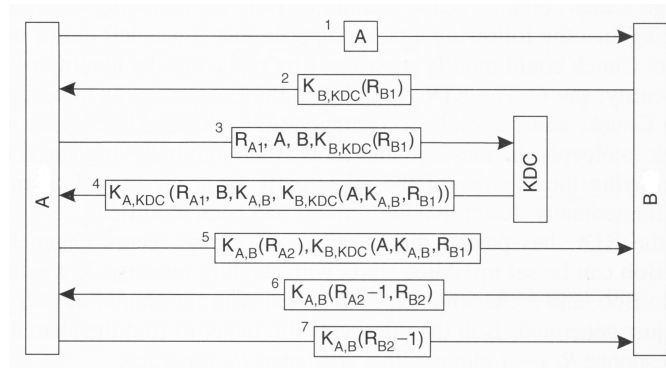
Fig. 2. Revised version of the Needham-Schroeder protocol.

The initiator A sends a message which contains his identity to responder B. B replies with a message containing a nonce (R_{B1}). This nonce is encrypted by the key shared by B and the KDC ($K_{B,KDC}$). A sends this message to the KDC. The KDC constructs a new message ($K_{B,KDC}$(A,$K_{A,B}$,R_{B1}))containing R_{B1} and sends it back to A. A sends $K_{B,KDC}$(A,$K_{A,B}$,R_{B1}) to B. B now can check if R_{B1} is identical to the nonce in message 2.

This revised version of the Needham-Schroeder protocol is introduced in [3].

3.2 Authentication with Public Keys

In chapter 3 the first two steps were necessary to obtain the keys required for the communication between A and B. When public keys are used these first steps can be omitted, since A and B can communicate with each other by using their public keys. All the messages to A and B are encrypted using their public key. Gavin Lowe also analyzed the Needham-Schroeder Public Key protocol [6]. First the system is formally described using the CSP notation [2]. This notation is used as input for the FDR checker to see if it is correct. The analysis is done by considering the protocol as a state machine where every step in the protocol can result in a new state. Every action that can be taken in the protocol can result in a new state. When all the steps in the protocol are performed, the authentication is completed. So when certain steps in the protocol can be skipped while the authentication can still be accomplished, the protocol is not safe. Lowe showed by systematic analysis of all the states and actions in the system that it is possible to skip an action in the protocol and still finish the authentication successfully.

3.2.1 Initially proposed protocol

 A sends an encrypted message (PK_B(R_A,A)) to B containing a nonce (R_A) and his identity. B replies with a message (PK_A(R_A,R_B)) consisting of the nonce R_A and a new nonce R_B. A replies with the nonce R_B.
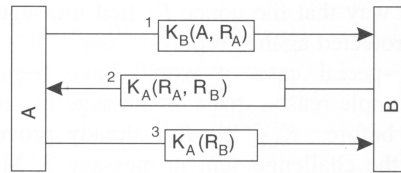
Fig. 3. Needham-Schroeder public key protocol

3.2.2 Weakness in the protocol

Again there still is a weakness which can be exploited with the man-in-the-middle attack. To show an exploit of this weakness, an example will follow.
In this example M is the intruder as described in section 2. A wants to establish a secure connection with M. In the meantime M will use the messages from A to do an authentication with B in a separate session. Since M will use the messages from A, B thinks he is communicating with A. At the end of both sessions, A will have authenticated M, and B has falsely authenticated M as A. For completeness all the steps are shown below.

A sends a message which is encrypted with the public key of M (PK_M). This
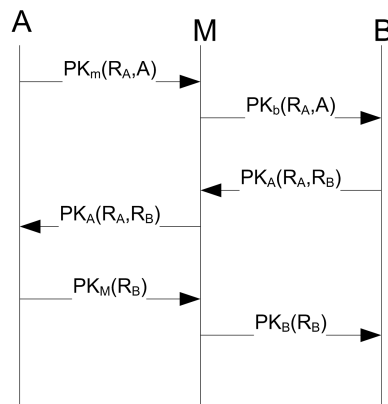
Fig. 4. Example of a man-in-the-middle attack

message consists of a nonce (R_A) and the identity of A. Instead of replying to this message, M decrypts the message and encrypts it with the public key of B (PK_B). This message is sent to B. Because of the included identity of A, B thinks the message comes from A. B replies to M with a message encrypted with the public key of A. M replays this message to A. A decrypts the message to obtain nonce R_B. A encrypts this nonce with the public key of M and returns the message. M now can decrypt the message and can obtain the nonce R_B which he encrypts with the public key of B. Authentication can now be finished by sending the encrypted nonce to B.

3.2.3 Improvement by Gavin Lowe

Gavin Lowe proposed a solution for this vulnerability [6]. This solution is similar to the one described in the previous section (3.1). By linking the different messages M can't just reply the message coming from B to A. This way the vulnerability is removed. The improvement is done by a very small adjustment
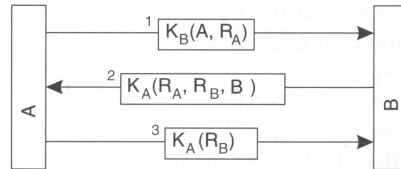
Fig. 5. Improved Needham-Schroeder public key protocol

in message 2. Instead of only sending nonce R_A and nonce R_B, the identity of B is also transmitted. Since A knows where the message is coming from, it is impossible for M to perform the man-in-the-middle attack as described in the previous section. This improved version of the protocol is also analyzed by using CSP and FDR. This analysis did not show any safety flaws.

4 Systematically Analyzing Communication Protocols

It was only after nine years that Needham published an article with an improved version of the protocol [3]. It still lacked any hard evidence about the security. So it is clear that more powerful and scientific tools were needed. In the article of Gavin Lowe [6], CSP was used together with FDR to analyze the original Needham-Schroeder protocol. He proved that the original protocol had a serious security flaw. He came up with a solution and used the same tools to prove the solution was adequate. When these systems are used, protocols can be checked faster and more efficiently. We shortly describe what CSP is and what can be done with it in combination with FDR.

4.1 Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) is a formal language for describing patterns of interaction. CSP was designed by C.A.R. Hoare in 1978. It is used to describe general interactions between (parallel) processes. Because of its properties CSP is well suited to describe communication protocols in general. Gavin Lowe used it to describe the Needham-Schroeder protocol. CSP is only a describing language. For a complete description of this language see [2]. For further analysis CSP can be used as input for FDR.

4.2 Failures-Divergence Refinement (FDR)

FDR (Failures-Divergence Refinement) is a software package which allows the automatic checking of many properties of finite state systems. When a process fails a check, it is possible to investigate the process with FDR. To do so, FDR takes two CSP processes as input, the implementation and the specification. With those two processes FDR can check whether the implementation is in accordance with specification. Gavin Lowe used FDR to analyze the Needham-Schroeder protocol [6].

5 Results

In the first part of this paper the development of the protocol is described which did not use any formal analysis methods. This informal approach makes it impossible to make any meaningful assumption about the safety. The second part described how Lowe used formal analysis methods on the protocol and proved that there was a flaw. He also suggested an improvement and proved that this version did not contain any flaws anymore.

6 Conclusion

The Needham-Schroeder protocol was first published without any formal analysis about its security. This protocol was critisized because of a security flaw. Several solutions followed, of which some were even incorrect. After almost 10 years after the first publication (of the protocol), Needham mentioned a revised protocol. This revision was not accompanied by any formal prove. Lowe showed again the same flaw in the protocol using proper analysis and also gave a revised version. However, Lowe provided a formal prove for the revised version. Only after this prove the protocol was suitable for safe usage. This paper has shown that proper logic tools are of vital importance when developing communication protocols.

References

1. Needham, Schroeder.: Using Encryption for Authentication in Large Networks of Computers. Communications of the ACM (1978) 993-999
2. C.A.R. Hoare.: Communicating Sequential Processes. Prentice Hall (1985)
3. Needham, Schroeder.: Authentication revisited. ACM SIGOPS Operating Systems Review (1987)
4. Formal Systems (Europe) Ltd.: Failures Divergence Refinement-User Manual and Tutorial (1993) Version 1.3
5. Li Gong, Roger Needham, Raphael Yahalom.: Reasoning about belief in cryptographic protocols. (1990)
6. Lowe.: Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. (1996)
7. Andrew S. Tanenbaum, Maarten van Steen.: Distributed Systems. Prentice Hall (2002) 432-440

Capturing The Missing Link: Design Decisions

University of Groningen
The Netherlands

George Craens, Hielko van der Hoorn

G.D.Craens@student.rug.nl; *H.van.der.Hoorn@student.rug.nl*

Abstract

One of the most important artifacts in the software engineering process is the software architecture. Errors made in the design can lead to huge problems in a later stage of the project. Usually only the architecture itself is described, but not the decisions that are made and the rationale behind these decisions. Rationale in the context of architectures describes and explains the used concepts, considered alternatives, and structures of systems [2]. Leaving design decisions and the rationale behind them in the minds of the architects has several drawbacks. The first problem is that this information can easily be lost. Not only when the architect leaves the company, but also the reasoning behind a decision can easily be forgotten. This can cause problems when the architecture will be expanded. Another problem is that an architecture without explicit documented design decision is less clear to stakeholders such as developers and customers. The logical solution that is proposed is to capture this information in the architecting process. It is not clear how this should be done. Some people suggest to include this information in text in the architecture document, while other people propose the use of software tools to capture this information. In this paper we describe and compare different approaches to capture design decisions.

Keywords: Software architecture, architectural knowledge, design decisions

Introduction

Knowledge of an architecture can be divided in a design and a set of decisions (Fig. 1) [1]. These decisions explain why the design is made in the way it is. The problem is that in practice, design decisions will not be documented and in the end they might get lost. Losing this knowledge makes it harder to evolve the system, increases the complexity and it makes reuse of the design experience difficult [2]. The purpose of our research is to describe and compare different approaches to capture design decisions and the rationale behind them. We will first explore what design decisions are and why it is important to capture them in more detail. Finally we will describe and compare different approaches to capture these decisions. The methods we used for the creation of the paper are desk research and literature review.
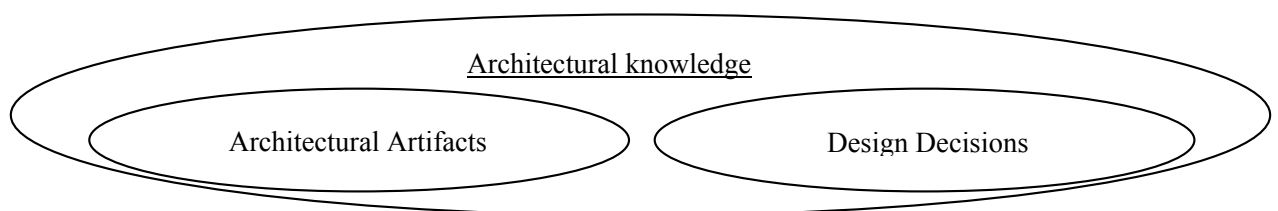
Figure 1: Venn-diagram, describing Architectural knowledge

Design decisions in detail

As already explained design decisions and the rationale behind them are, together with the software architecture a large part within the domain of architectural knowledge (Fig. 1). Design decisions can be categorized in four types [1]:

- Implicit and undocumented: the architect is unaware of the decision, or the decision will be assumed to be very clear for everyone anytime. For example retrieved experiences from other projects, tacit company policies to use certain approaches, standards, etc.
- Explicit but undocumented: the architect takes a decision for a very specific reason (e.g.) the decision to use a certain user interface policy, because of time constraints). The reasoning is not documented and slowly gets lost over time.
- Explicit and explicit undocumented: the reasoning is hidden. There may be tactical company reasons to do so, or the architect may have personal reasons (e.g. to protect his position).
- Explicit and documented: this is the preferred, but quite likely exceptional situation.

The last type of design decisions is preferred, but this is rare. Usually the decisions are not documented. For some decisions this is not a problem since these will be visible in the software architecture. These are usually so called existence decisions which state that some artifacts will be in the architecture. This is however not the case for non-existence decisions which state that something will not be in the architecture. Also hard to trace are decisions that affect the whole system or decisions related to the business environment.

Reasons to capture design decisions

Although it is at the moment not common to capture the design decisions, there are good reasons to do so. One of the key challenges within the software architecture community is the fact that architectures need to be designed very carefully. This is because changes are very costly after the initial design has been completed. It is believed that this is difficult and expensive because of knowledge vaporization [4].

All important design decisions will be embedded and implicitly present in the software architecture, but they are not explicitly documented. This is directly a problem, because decisions can get lost easily when they are not explicitly documented. They are also often closely related to other decisions and fragmented throughout the design, making it hard to find a decision. Another problem is that changes to the software architecture can lead to violation of earlier decisions, reducing the value of the architecture even further [5].

There are also other reasons to capture the design decisions. Thanks to explicit design decisions the understandability of the software architecture increases. This can benefit other users such as architects that are working on the same system, reviewers, software developers and even software tools when the decisions are captured in a structured way [1].

Problem statement

Although it is clear that capturing design decisions is desirable, it is not clear what a good or what the best approach is. Different authors suggest different methods of capturing design decisions (or architectural knowledge in general) [1][2][3]. Another problem is that it is not clear

what exactly should be captured. Capturing every single decision sounds attractive, but is usually not feasible due to time and money constraints. There are no guidelines on what should and should not be captured. This is however outside the scope of this paper. In the following part we will focus only on comparing different methods of capturing design decisions. We will not answer the question on what exactly should be captured.

Solutions

How can architectural knowledge be captured, in particular design decisions? Different approaches are discussed in the next paragraphs.

Perhaps the easiest solution is to capture all key architecture decisions in a separate document [3]. In the method all 'key' architectural decisions are written down in a document using a fixed template. For every decision it is necessary to specify why the decision is important, what the decision is, the rationale behind the decisions, the assumptions, constraints and alternative decisions considered. Also things like implications, related decisions, related requirements and a few other fields are required.

In addition to the template filled in for every important decision, it is recommended to create a second view for the management and business stakeholders. This view focuses on the key decisions and the implications of these decisions. Another suggestion that is done is to use a simple coloring scheme to point out incomplete or controversial decisions.

Although this approach is probably already a lot better than capturing no decisions at all, it is not perfect. Research has shown that capturing design decisions will improve the understandability of a software architecture, making reuse and changes easier [2]. It is however very time-consuming to create and maintain a document with design decisions and rationale. This is in addition to the problem that capturing design decisions usually has no direct benefit for the person that should be capturing them, the software architect. The real value of capturing design decisions become visible in a later stage of the lifecycle, when the system should be changed, or when the desire arises to reuse parts of the architecture [2]. Another problem of this approach is the loose coupling between the design decisions and the software architecture, making it hard to use the rationale and to keep it up to date [2].

A solution to this problem is to make the capturing of the design decisions an integral part of the software architecting process [2].

The process of capturing the rationale behind decisions is closely related to the process of designing the software architecture. The problems are derived from the requirements document and the solutions that have been chosen will shape the software architecture. Based on this link, the idea is to combine the artifacts of the software architecture design and on the other hand rationale management.

To make this possible a tool called Archium has been created. Archium is capable of capturing an architecture together with the rationale behind it [6]. It is an extension of Java, consisting of a compiler and a run-time platform. It supports architectural concepts like components and connectors that can be described using Architectural Description Language (ADL) concepts. Together

with the architectural artifacts the design decisions and the rationale behind them can be captured. The tool also captures changes in the history of the design since architectural modifications are part of design decisions. Archium forces the user to identify design decisions and to name them, but providing the rationale behind the decisions is optional.

Using Archium reduces the amount of work needed to capture these decisions since they can easily be entered while creating the software artifact. As already explained this is important since capturing design decisions is a time consuming task, which does not have direct benefits for the software architect.

Another important positive point of Archium is the fact that the software artifacts are linked with the design decisions. This makes it easier to keep the design decisions up-to-date when the architecture is evolving. It can make the developer aware of certain design constraints and when a certain software artifact is removed, an obsolete decision can be removed automatically. It is also capable of checking if rules and constraints are not violated, when they are imposed on the architecture by certain design decisions.

A second tool to capture design decisions is Sysiphus [1][7]. This tool has not been build to show how easy it can be to integrate the process of capturing design decisions in the existing workflow, but to show what is possible when design decisions and architectural knowledge in general are captured in a systematic way. Sysiphus is a tool suite to provide a solution to manipulate system models and rationale in a distributed environment.

The implemented system models range from documents, requirements, use cases, nonfunctional requirements to UML classes, components, sequence diagrams or test cases. Various use cases are supported, some are basic like adding a decision, but also more complex ones are possible. Examples are the evaluation of a change in a component or cleaning up the system (making sure all consequences of a removed decision are also removed). There are also some use cases that could be supported, but are not yet implemented. Some examples of this are the use cases "spot the subversive stakeholder" and "detection and interpretation of patters". The first use case is for spotting the stakeholder that has caused the most changes/damage to the system. The second use case is for automatically finding patterns in the architecture(s) that could lead to guidelines for new designs [1]. This tool is still very experimental, but it shows a glimpse of what is possible when design decisions are captured in a language a computer can understand.

Instead of using templates or tools, it is also possible to use design patterns. Patterns are devices that allow programs to share knowledge about their design [8]. The general form for documenting patterns is to define items such as:
1) The motivation or context that this pattern applies to.
2) Prerequisites that should be satisfied before deciding to use a pattern.
3) A description of the program structure that the pattern will define.
4) A list of the participants needed to complete a pattern.
5) Consequences of using the pattern., both positive and negative.
6) Examples

When using these patterns, it is not necessary to capture (most of) the architectural knowledge, because it is already done. Another advantage of this approach is that it is cheap to use. A disad-

vantage is that it not covers knowledge for example about a very specific reason, tactical company reasons or interface policy.

Discussion

Advantages of capturing design decisions are that it supports reuse and change of the architecture, because earlier made decisions are transparent [2]. The risk of violating rules or constraints from previous decisions will be minimized and it improves the quality of the architecture, because an overview of decisions will make it easier to get consistency between decisions [3]. The third advantage is the support of knowledge transfer over time and location [2]. The last point is very important since the main goal of a software architecture is to transfer knowledge, but without design decisions not all relevant knowledge can be transferred.

There is little doubt within the software engineering community about the value of capturing design decisions. Although these are at the moment almost never explicitly documented, there is a consensus that they should be captured to make the software architecture easier to understand and to prevent knowledge erosion when the system is evolving [4].

One big reason why design decisions usually are not captured is the fact that there is no standard approach to do this. Documenting the software architecture has become a common practice, but the realization that capturing the design decisions is also valuable is new. Therefore there is not yet a standard approach or process like there is for capturing the software architecture. In this paper we have compared three different approaches to capture design decisions (Table 1).

The simplest approach is to store this information in a separate document while designing the software architecture, but this solution is not perfect. Keeping this document up to date and consistent will be time-consuming while the rewards of capturing the knowledge are not directly visible. It is also not very easy to use since the coupling with the software architecture is loose, making it hard to use the knowledge in the document and to keep it up to date when the system is evolving.

The proposed solution for this problem is a tool based approach. The rationale is created and stored together with the design artifacts making it easier to use. The use of tools can also offer additional benefits such as showing the relationship between decisions, showing the impact of certain changes or checking if no constraints imposed by certain design decisions are broken.

We have looked in this paper at two different tools, Archium and Sysiphus. At the moment Archium is the only tool that has the capability to be used in a real software development project. Sysiphus is just a proof of concept tool that is far from finished. Archium offers a good way to store design decisions together with the software architecture giving the advantages of the template based approach without the disadvantages. The tool is however only suitable for software development in Java and it is still in a beta status.

Approach	Advantage	Disadvantage
Template	• Easy to implement	• Time-consuming to create • Loose coupling between design decisions and architectural artifacts
Archium	• Easier to integrate in workflow, less time-consuming to use • Software artifacts linked with design decisions	• Java only • Beta status
Sysiphus	• Powerful functions available to review, change and use architectural knowledge	• Experimental status, more a proof of concept
Design Patterns	• Cheap	• Does not cover decisions about e.g. a very specific reason, tactical company reasons or interface policy.

Table 1: Approaches to capture design decisions

Conclusion

Although at the moment there is not yet a definite solution on how to capture design decisions, we expect that in the next years the situation will change. There is consensus that capturing design decisions is important, so it should only be a matter of time before better tools will become available and capturing design decisions will become a standard practice.

References

Articles:

[1] Philippe Kruchten, Patricia Lago, Hans van Vliet, Timo Wolf
     Building up and Exploiting Architectural Knowledge, 2005
[2] Jan S. Van der Ven, Anton G. J. Jansen, Jos A. G. Nijhuis, Jan Bosch
     Design Decisions: The Bridge between Rationale and Architecture
[3] Jeff Tyree and Art Akerman
     Architecture Decisions: Demystifying Architecture, 2005
[4] Jan Bosch
     Software Architecture: The Next Step
[5] Anton Jansen, Jan Bosch
     Software Architecture as a Set of Architectural Design Decisions

Internet:

[6] http://www.archium.net
[7] http://sysiphus.in.tum.de
[8] http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns

Determining the impact of design decisions

Frans Kremer, Herman van Rink

University of Groningen

Abstract. *Recent research in software architecture has resulted in an increasing effort into explicitly capturing architectural design decisions in the software architecture using a variety of means. Although these design decisions are captured, the relations between them are as of yet undefined, or defined on a very basic level. The result is that it is unclear how a change in one design decision affects other design decisions. We propose a means to classify, identify and measure the impact of a change in one design decision, to other design decisions. This is done by providing an ontological view of design decisions where relations are made and graded based on key characteristics, such as component relations (if any), quality attributes and architectural constraints. We present a means how these relations can be defined in AREL.*

keywords: design decisions, impact measurement

1 Introduction

The software architecture plays a crucial role in the development of software systems. However, the software architecture is only the end-result of an enduring process. Indeed, the architecting process is often an iterative process, and continues for the life-span of the product. During the life-span of a software system, changes are made to it to meet new or changing requirements. The cost of changes depends greatly on the maintainability of the software system. Often, cost of change is a key factor why software systems are replaced. This is because as software systems age, they become harder to maintain. One of the causes for this is design erosion. Design erosion is in essence the difference between the software architecture and its implementation. Often, design erosion occurs where incremental changes are made to an existing software systems. In these cases, the maintainers of the system violate constraints or neglect to update the software architecture documents. Thus, mismatches start occurring between implementation and architecture. There are several causes why design erosion occurs. One of the major reasons for this is because the architectural knowledge is lost (an effect known as knowledge evaporation). As mentioned above, the software architecture only resembles the end-product, not the process. The software architecture only captures the results of the decisions that are made during architecture design. The aspects that result in these decisions however are often

not captured. Such aspects range from rationale to context and dependencies on other decisions. In effect, what is missing are the design decisions.

Recent research efforts by Kruchten, Van der Ven and Tyree has changed this [3,1,2]. Several researches have tried to capture design decisions as an explicit part of the software architecture. a design decision is a description of a choice of a solution-set that (partially) realize one or more requirements. Each solution consists of a set of architectural modifications, rationale, design constraints, design rules and additional requirements. Kruchten's research into design decisions is in an exploratory phase; he identifies several categories of design decisions and proposes a tool in which these decisions can be captured. Tyree has a more concrete definition; he proposes a method to explicitly define the design decision description in the architecture document. Jansen & Van der Ven go a step further by explicitly defining the design decisions as part of the implementation, therefore linking the software architecture and the implementation directly. However, this model is insufficient to define executive decisions, such as the use of a specific technology. the section hereafter discusses their research more broadly.

It is a great asset to be able to define design decisions; with it, the scope of a decision can be tracked throughout the architecture. However, these models don't provide a manner to predict how a change or an addition of a design decision impacts the architecture; it isn't possible to define a link, other than a hierarchy, to define which design decision has a dependency on an aspect of another design decision. Tang provides an excellent technique called AREL to link design decisions together using Bayesian belief networks [4]. He believes, that by defining and grading cause-effect relations between architectural elements (anything from a view to requirements) and design decisions, the impact of a change can be measured throughout the architecture. The grading is done based on experience of the architect. His research lacks however distinction in characteristics of a design decision to which a cause-effect relation can be based.

In this paper we propose a modified version of Tang's technique that enables the definition of impact-relations on a per-characteristic level. We also propose a means to weigh these relations automatically, so to remove the dependency of an expert's opinion for the weighing of relations. The remainder of this paper is organized as follows; the following section discusses the problems architects currently face with changing architectures. Next, section 3 discusses the research efforts made to elevate this problem. After that, section 4 proposes an addendum to this research designed to improve the accuracy of impact prediction. Finally, we conclude this paper in section 5.

2   Problem statement

In most development projects much of the tacit knowledge is lost after the project is completed or even during the project. Although there are methods of documenting these knowledge fragments, most of them are time consuming

and tiresome. When the action of updating the software architecture documents is omitted once, the software system erodes due to inconsistencies. Due to this, architecture documents of aged systems are often inadequate for the maintainers. Moreover, current architecting techniques are unable to capture the intricate relations between design decisions made during the architecting process.

When a design decision is reevaluated after some time it is important to make an accurate impact prediction. And if the data is already inconsistent, the chance of making an accurate impact prediction is relatively low. Thus it becomes clear that current ways of change management are inadequate, since they require intimate knowledge of the architecture. As stated above, this knowledge is lost after iterations of architecture design.

There has been some development in the field of software engineering to elevate this problem. One of these solutions is the explicit definition of design decisions. The next section discusses this in detail.

3 Current research

This section discusses the research efforts of Kruchten, Tyree and Jansen & Van der Ven. These researchers all try to capture design decisions more explicitly in the architecture [3,1,2]. To better understand the results and problems remaining with these research efforts, we discuss their research effort below.

Small derivations aside, the research efforts of Tyree and Jansen & Van der Ven share their definition of design decisions. Jansen uses the following definition for design decisions [5]:

> *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.*

A design decision has a set of properties, Tyree captures these in the architecture document using a table [2]. In this table, important information about the decision is captured, such as assumptions made, alternatives considered, constraints, rationale, implications and related design decisions and requirements. The main difference between Tyree's research and Jansen's efforts are in the fact that Tyree captures decisions in the architecture document whilst Jansen tries to capture the design decisions in the implementation itself. The latter results that system-wide decisions (e.g. platform used) cannot be adequately defined.

Kruchten's efforts are more exploratory; Kruchten tries to identify different characteristics in design decisions and identifies three major types. These types of design decisions are:

executive design decisions Executive design decisions are made on a management level and are mostly communicated through emails and other natural language documents. Executive design decisions are design decisions which pertain on a global scale of the software architecture; they constrain the architecture in a global manner. This makes it hard to automatically create relations with other design decisions. Also the architect receiving these communications has to cope with the often quickly changing perceptions of the management team. Within a development project the management can come up with many new (potential) requirements which have to be evaluated for technical implications and cost of realization. Capturing these executive design decisions with its rationale in the technical documentation can be considered a challenge.

existence/constraint design decisions This type of design decisions can be found on most levels. They can be derived from executive design decisions and other requirements. Often as a natural consequence. Therefore they are important as links between property and executive design decisions.

property design decisions Property design decisions are the smallest fragments of a design decision. They specify small properties from larger Architectural Elements. While being fairly localized for the most part. A property that requires a specific protocol version to be supported, would for instance put a constraint on all the communication partners of that component.

While design decision capture knowledge of the software architecture, the relations between them are not yet exploited. Changes in the architecture require that (related) design decisions need to be reevaluated. Since changes always occur during the life-time of a software system, reevaluation of design decisions is imminent. However, as much effort there has been into design decisions itself, little has gone into how design decisions relate to one another. The result is that there is still little knowledge of how changes in one design decision affect the other, and moreover how this can be measured up-front (thus, it is still difficult to determine how a change impacts the architecture and where problems may arise). The research efforts discussed above do try to capture these relations. Below is discussed what results have been made.

3.1 Relations between design decisions

Because relations between design decisions are so important to be able to make an accurate impact prediction of a change in the architecture, Tang, Tyree and Kruchten all try structure these data. Kruchten gives the most detail in his breakdown of structural elements. He gives a detailed view on the different ontological perspectives one can have on an architectural design decision. He also differences with Tyree and Jansen by identifying states of a decision (idea, decided, rejected, etc...). These characteristics come into play because of dependencies to other design decisions with varying states.

An area that both Kruchten and Van der Ven touch is that of design decision relations. Van der Ven talks mostly about alternatives and constraints, while Kruchten goes into more detail. He mentions relations like *forbids*, *enables*, *conflicts with*, *overrides* and *comprises of*. Other relations are *alternatives to* and the very general form, *is related to*.

These relation types still fail however to be able to determine which design decision has an impact on another design decision. Basically, there is only one method of determining impact. This is to manually analyze each relation of a design decision. Also the number of incoming and outgoing relations can say something about the relevance of a decision. If many decisions reference a certain design decision, then chances are that a change in that area has larger implications.This is a very crude and tiresome method of formally analyzing change. Tyree has a different approach to this. He mostly uses static tables and color coding to represent his data. While this would work for a limited number of design decisions. Finding out what the impact is of a new decision will be increasingly difficult as the number of decisions grows. In which case the number of possible impact relations shows an exponential growth. Keeping track of a large architecture in this fashion will be nearly impossible.

Tang is one researcher who has taken a step further in relations between design decisions [4].He proposes the AREL model, which is a tool which can calculate the likeliness and the impact of change of a design decision in correspondence to the other design decisions. AREL uses a Bayesian belief network technique to achieve this. Tang distinguishes two types in his model; *architectural elements* (AEs) and *architectural rationales* (ARs). AEs are requirements, designs or implementation artefacts. ARs are the reasons of why a design decision is made. An AR consists of the assumptions, constraints and tradeoffs of a design decision. ARs correspond to Jansen & van der Ven and Tyree's definition of design decisions. Therefore, we will refer in the remainder of this paper to ARs as design decisions. Tang then discusses how cause/effect relations are made between design decisions and architectural elements and, using a Bayesian belief network the impact of a change of one design decision affect all other design decisions. Although this method works well, it is still based on the gut-feeling of the architect (he/she still has to do the grading). Moreover, AREL doesn't make use of the different types of design decisions and their characteristics. In the next section, we propose a refinement of AREL, which incorporates these features.

4   Refinement of AREL

In this section we propose a refinement of AREL's approach of measuring impact. We provide three additions to AREL:

- Global versus local decisions
- Relation types

– Self-refinement of the network

These three addendum's are discussed below.

4.1 Global versus local decisions

The influence (or impact) of a design decision can be categorized as being *global* or *local*. This categorization is based on a design decision affecting the entire architecture, thus affecting all other design decisions, or only a small part of the implementation, i.e. only a few design decisions. To compare this with Kruchten's categorization of design decisions, *executive design decisions* always have a global impact (e.g. the decision to use a specific programming language).

Typically, global decisions have a huge structural impact on the implementation and thus on all other decisions, local and global alike. However, local decisions often have a low impact on global decisions because they only affect a part of the architecture (the only exception to this is when a requirement, followed by a local design decision is in conflict with a global design decision; e.g. a local decision that is impossible to realize given global constraints). The reason we categorize decisions as being local or global is because global design decisions always need to be reevaluated after a change in the architecture and it is useless to connect a global decision in the AREL network (since it would be connected to almost every other node in the network). To summarize, in our view, the AREL network should only consist of local decisions, and global decisions should be reevaluated every time the architecture changes. In the next section we discuss the different types of design decisions and how it affects AREL.

4.2 Relation types

Tyree, Kruchten and Jansen & Van der Ven all identify properties, such as related artefacts, of design decisions. Each property of a design decision can be related or may affect other design decisions. In AREL these relations between design decisions are defined in a generic way; a single type of relations. Instead of AREL's generic way of defining relations, we identify relations between design decisions based on their properties. The following lists consists of all properties which may result in a direct or indirect relation to another design decision:

requirements As defined by Jansen & Van der Ven, design decisions are used to (partially) realize one or more requirements. Naturally, there exists a strong relation between the requirements and the design decision.

realization The implementation of a design decision involves a segment of the implementation; a set of components. Each component (partially) realizes at least one or more design decision.

constraints Constraints can be divided into two groups; global and local constraints. As mentioned earlier, we define global as "involving the entire architecture". Hence a global constraint (such as "must use J2EE") will automatically imply a global design decision. A local constraint is a constraint which is only applied on part of the architecture, such as "module A must respond in 20 millisec.").

implications Implications are decisions that have no direct relation to the current design decision through the architecture, but are resulting from the current decision. For example the decision to develop for Windows-only leads to the decision to use specific (platform-dependant) libraries.

By defining relations according to their type, the network is more segregated, which results in a better understanding of the influences of design decisions. Obviously, by only defining (and grading) that some relation exists, it is still difficult to determine how such a relation influences other relations.

4.3 Self-refining networks

In the previous two sections we stressed how the effectiveness of the AREL network can be improved. In this section we discuss another huge factor that determines the behavior of the AREL network; the grading system. In Tang's research, the relations in the AREL network are graded based on the experience of the software architect. This method relies on the tacit knowledge of the architect. Due to the volatile nature of this type of knowledge (the architect may leave the company or some other form of knowledge evaporation may arise), it might be better to use an additional method for grading relations.

We believe that grading should be determined by the design process rather than tacit knowledge. To enable this, feedback into the AREL network is required; the architects should not only use the AREL network to try to determine impact, but also feed the AREL network how changes propagate throughout the architecture. By doing so, the AREL network can determine what the "real-life" impact factors are and thus refine its impact calculations. Of course, when first setting up AREL, it is still extremely valuable to grade the relations by hand, thus providing a basis for AREL. However, the intention for self-refinement is to enable AREL to cope with a changing environment in the architecting process. Thus, once chief architects leave the project team, AREL is still able to cope with changing behavior of the architecture design.

5 Conclusion

In this paper we discussed what the current state of research involving design decisions is, and how this knowledge can be combined with the AREL network

to be able to determine the impact of design decision changes. We also provided a few modifications of the AREL network, which enhance its use in practical environments. To test our claims made in this paper, the modified version of AREL has to be tested in a practical environment. This is however beyond the scope of this paper to do so.

References

1. *Building up and exploiting architectural knowledge* P. Kruchten, H. van Vliet, T. Wolf
2. *Architecture disisions: Demystifying architectures* J. Tyree, A. Akkerman
3. *Design desisions: The bridge between rationale and architecture* J. van der Ven, A.G.J. Jansen, J.A.G. Nijhuis, J. Bosch
4. *Predicting change impact in architecture design with bayesian belief networks* A. Tang, Y. Jin, J. Han, A. Nicholson
5. *Software Architecture as a Set of Architectural Design Decisions* A. G. J. Jansen, J. Bosch. WICSA 2005

Three methods for modelling variability in software products families

Mohammad Babai and Henk van der Veen

University of Groningen,Blauwborgje 3 9747 AC Groningen
{m.babai,h.h.van.der.veen}@student.rug.nl

Abstract. Variability is one of the increasingly important attributes of the modern software development. Variability makes it possible that the software artefacts can be reused and configured to different contexts thus easing development. But managing variabilties brings its own difficulties and there is no consensus on how this must be done.

In this paper we describe three methods for modelling variability: Koalish, VSL and COVAMOF. We will give a brief description of each method in order to evaluate them based on their approach and strengths and weaknesses. These properties will be presented and a comparison will be given based on their respective effectiveness in processing different types of requirements (e.g. functional or non-functional).

Key words: Variability, product families, variability management, variability in software systems, Variation specification.

1 Introduction

During the past few years the software industry has made a shift toward developing software with an increased amount of variability. Consequently variability is rapidly becoming an important factor in software development. Modern software products are frequently developed and delivered in multiple combinations and variants. As a simple example, we may consider development tools that are delivered in light, professional, enterprise versions or alternatively the operating systems that are delivered in home, professional, data center edition with support for different number of clients and CPU's. They all have the same basic set of features but they differ in the number of extra available functional components. Furthermore, most of the products can be run on different software or hardware platforms. additionally, most products can be configured during installation, startup and run-time.

A software product family or product family for short can be described as a range of related products that share or reuse software functionality as much as possible. The architecture of a product family has to be flexible enough to make it possible to handle attributes such as functionality, performance and adaptability across all the members of the family. It may also have support for the future products that are planed to be part of the family (yet to be developed products). The differences between the products in a product family are mainly based on the variability concept. This may be defined as the ability to adapt

a software system to a specific application context. These adaptations may be realized by adding, removing or altering parts of a software system to customize it for a specific customer or platform. For example the configuration settings may be altered at compile time, after deployment or even at runtime.

Variability is one of the concepts that is studied in the field of product families. One of the common approaches found there is to identify features that are shared between all the members of the family or the properties that are specific for one or a few of the members. If features can be altered (included, excluded, changed) then they represent a variation point in the system for which different variants can be made available. This approach is used as a basic for designing and implementing the shared and specific features of the products.

There are a lot of different methods and techniques to implement variability. For example, Conditional compilation allows us to select different variants during compilation. We may specify that a piece of code should be build in the system or not and the build parameters that affect the behavior of the components can be specified as well. Configuration selection works on the software configuration and gives us the possibility to include or exclude a particular component. For example we may choose to use GUI system A instead of GUI system B for the same software system. Run-time parameter binding makes it possible to select the variables in the almost the latest stage. For example by choosing to run the program in server of client mode or (de)activating a particular feature of the program.

There are two main reasons to realize variability in software products. First, variability has been recognized as a key to systematic and successful reuse. In software product lines or families, variability is a way to handle the inevitable differences among the systems in the family, while they use the common properties. In this case the re-usability can be increased by applying variability to the system.
Second, by providing more variability in a software system the flexibility and maintainability of those systems may be improved [1]. The features can be added, removed or changed even at runtime, without releasing a new product.

Increasing variability in software systems leads to the situation where managing the variability becomes a very complex task and sometimes one of the main concerns during the software development process. The variability management appears to be a non-trivial task. The first fact which causes difficulties in the consistent management of variability's is that they often can not be localized. Other complication which may arise as a result of variability is the interdependency among the solution parts. For example, variability's may exclude or require each other, resulting in further interdependencies.

The remainder of this paper is structured as follows:
In section 2 we will give a brief description of three different approaches to handle and manage variability in systems and in section 3 we will close the paper with a discussion. In the latter section we will compare the described methods based on their qualities and effectiveness.

2 Methods to realize variability

In this section we give a brief description of three methods to handle variability in software systems. First Koalish, then VSL and finally COVAMOF.

2.1 Koalish: A Koala Based approach

Koalish is based on the variability management of hardware. It concentrates on a subclass of product families, namely configurable software product families. The main idea is to divide the system in different components with a clear interfaces and formal constraints. The components are defined in a declarative manner, which makes it easy to read. Components have a required and a provides interface. They can also contain other components, of which the possible types and cardinality of those components can be specified as well as some simple constraints.

The main goal is that a customer can simply enter his requirements through series of choices in a user friendly program and a valid configuration for his desired comes out. After every step a partial stable model (The model with some of the variation points bounded) can be calculated to remove choices that are bound by the previous choices. There is little to no need for glue code between the components.

Both the component specification and the customer requirements are translated to Weight Constraint Rule Language (WCRL), a logic language. In this representation the partial stable model is found and eventually the stable model (all variation points bounded). The stable model is translated to a specification that can be read and checked by an engineer or to an input for a realization tool that can compile the code automatically.

We can see two important advantages of this system. The fully automated and guided derivation of family members enables the creation of hundreds of different family members, all customized for a specific customer in very little time. From the perspective of the designer you have a clear and readable syntax to which everybody must uphold, with easily proven soundness.

But Koalish is only usable for simple systems with simple dependencies (requires, include, exclude). Dependencies on non-functional requirements can not be modelled, since Koalish only supports formal specifications. Also, since interfaces need to fit exactly, it is very hard to change them later on and impossible to cope with incomplete knowledge. There's no expert in the middle who can think of a different way to bind components (with a little bit of programming). This affects the evolution of the system.

2.2 Variability Specification Language (VSL)

VSL provides a meta-model for variability management in software product families. It is an attempt to specify a more systematic approach to manage variability which can be used as a base for building a variability management system.

A product family in VSL is at the core the same as with Koalish: A collection of Assets which make family members by configuration. But in Koalish the Asset is always a component and in VSL it can be a partial solution in any form, like a design document or a component. An asset can contain variation points. The various variabilities can have different binding times like compile time, configuration time or run-time. The configuration process is much more complex and it will usually be only partially possible to automate.

There are two types of variation points: Dynamic and static. Dynamic variation points can be bounded after the software is released. The configuration of the static variation points defines a specific family member and the binding is done during the design and implementation of the member. This configuration is called a profile.

In this model two main levels are recognized while creating a family member, the specification level and the realization level.
The specification level contains mainly meta-data and the realization level the software. So for example the realization level speaks of variation points in the software, which contains the different options for creating the system and mechanisms to do that as well as simple dependencies between variation points (require, exclude). The specification level speaks of variability, which contains rationales for certain choices, complex dependencies and binding times. Variations points are linked to the variabilities. The profile also lives in the specification level and binds the static variation points through resolution rules.

This model has some very nice features. For example a clear and understandable structure. There is some structured meta-data, which allows for complex dependencies as well as partial automatic processing. Binding of the variation points can postponed to a later phase.

There are still some points of improvement. For example, VSL doesn't support the modelling of non-functional requirements.

2.3   Modeling complex dependencies using COVAMOF

COVAMOF is based on the idea that it is not always possible to describe the difference between all choices of all variation points in a formal, computable way. This because of the nature of the problem or the time it takes to do it. It tries to combine all the different types of knowledge into one system: Formalized (can be interpreted by computers), documented (informal descriptions or models) and tacit (exists only in the minds of some experts) knowledge.

COVAMOF also knows variation points and most things that are in VSL, but there is not such a strict distinction between the realization and the specification level. What it does have is the possibility for different levels of abstraction (like feature model, architecture and component implementation). We'll concentrate on the features that we think really makes it better than VSL and other system.

The most distinctive property of the COVAMOF system is the Dependency. A Dependency is a relationship between a set of variation points and specifies a System Property (SP) who's value is based on the choices in these points. This is a different definition than in VSL.

Each *Dependency* is an object containing:

- A system property with its constraints (e.g. based on functional or non-functional requirements).
- A set of associations with variation points that influence the SP.
- A (partial) function with the formal knowledge of how the SP is influenced by the associations
- A set of references to files with the documented knowledge and/or the contact information of the experts with tacit knowledge
- A set of reference data containing the value of the SP for a specific binding of variation points.

Each *Association* with a variation point can contain either a logical function (which will be part of the Dependency-function), a description of how it will influence the SP or simply the fact that it will influence SP. The references to knowledge and test-data help the designer to find the information needed for a smart choice or an educated guess.

The main benefits of COVAMOF are:

Incremental externalization. Knowledge can gradually be documented or formalized when the maturity of the product grows or the need arises. Allowing for a quick start.

Linking knowledge. With one location for all knowledge concerning a variation point or dependency product engineers have less trouble finding all relevant information.

Reduced expert dependency. Experts can concentrate on the points where they are really needed. Leaving formalized or well documented choices for cheaper personnel.

Reduced complexity through abstraction. Specifying complex relations on groups of variation points on a more abstract level makes a more insightful system than a network of simple dependencies.

Less iterations. By explicitly dealing with complex dependencies, experts can spot a safe margin or a risk for certain dependencies earlier in the process, thus saving iterations.

Recorded reference data This can be used to estimate the value of a dependency in the future.

3 Discussion

The three methods are all clearly build from a certain view/goal. Koalish is build as a component system with the goal to easily create hundreds of different systems with different functionalities, good for reuse and differentiating between different packages (economy, gold, platinum ..). As long as dependencies can be formally specified, the system can easily be divided in components, and low-cost customization for the masses is desired, this system is perfect.

VSL and COVAMOF are alike. VSL is fairly theoretical and tries to create a general model that can be further specified by someone trying to build a variable

system. It misses some concepts of COVAMOF like COVAMOF's dependency. This would give it a better link to the requirements. Whether the levels of VSL or the layers of COVAMOF are better we cannot judge. The clear distinction between meta-data and program-data could make it preferable for some.

Both COVAMOF and VSL give you an abstraction from the hundreds of small variability points. Koalish knows a hierarchy between components and the components they are comprised of, which for the sort of systems it meant for is good enough.

COVAMOF works very good with extremely complex systems with little formalization, but will also work on a completely formal system. It won't be as fast as Koalish, but it can be completely automised. With all the benefits of the possibility to incrementally formalize the system, the links to knowledge, dependencies on non-functional requirements and available tool support, COVAMOF is for most the best choice of the three.

References

1. Towards a General Model of Variability in Product Families. Martin Becker. System Software Group, University of Kaiserslautern. Kaiserslautern, Germany.
2. Modeling Complex Dependencies with COVAMOF Marco Sinnema, Sybren Deelstra, Jos Nijhuis University of Groningen The Netherlands. Jan Bosch Nokia Research Center NOKIA GROUP, Finland.
3. A Koala-Based Approach for Modelling and Deploying configurable Software Product Families. Timo Asikainen, Timo Soininen and Tomi Männistö.
   Helsinki university of Technology, Software business and Engineering Institute. Finland.
4. Baum, L.; Becker, M.; Geyer, L.; Molter, G.: Mapping Requirements to Reusable Components using Design Spaces, Proc. of IEEE Int?l Conference on Requirements Engineering (ICRE 2000), Chicago, USA, 2000
5. Becker, M.: Generic Components: a symbiosis of paradigms, 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00), 2000
6. Svahnberg, M.; Van Gurp, J.; Bosch, J.: A Taxonomy of Variability Realization Techniques, Technical paper, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002
7. Jazayeri, M.; Ran. A; Van der Linden, F.: Software Architecture For Product Families: Putting Research into Practice, Addison-Wesley, May 2000
8. Van Gurp, J.; Bosch, J.; Svahnberg, M.: On the Notion of Variability in Software Product Lines, Proceedings of WICSA 2001, August 2001
9. Bandinelli, S.: Product Family Engineering with XML, Proc. of Dagstuhl Seminar No. 01161 Product Family Development, Wadern, Germany, 2001
10. Svahnberg, M.; Van Gurp, J.; Bosch, J.: A Taxonomy of Variability Realization Techniques, Technical paper, ISSN: 1103-den, 2002.

Evolution of Architectural Patterns
From the original concept to the architect's toolbox

Reinder Kamphorst

Rijks*universiteit* Groningen, The Netherlands, `r.j.kamphorst@student.rug.nl`

Abstract. A *pattern* is a re-usable concept that has been applied in several working systems. It is not a litteral description of the used structure, rather an abstraction from it that can be used in other systems. A pattern thus provides the architect with a recipe to compose a certain architecture or part of an architecture.

A *pattern language* is a collection of patterns, with clearly stated interdependencies. With a *generative* pattern language, a whole family of architectures can be generated.

The first to propose the idea of *pattern languages* is not a software architect, but a *bricks 'n' walls* architect: Christopher Alexander. He published a number of books on patterns and pattern languages in the late 1970s. In 1987, Ward Cunningham and Kent Beck were the first to bring in practice the archtiectural ideas of Alexander to make a pattern language for software. The authors known as the *Gang of Four* (GoF) published their book Design Patterns [GoF94] in 1994. A very recent milestone is the publication of POSA books.

There have been three stages in the evolution of architectural patterns for software. This paper identifies them by comparing three papers on this subject, and drawing parallels with the aforementioned milestones in the evolution of (architectural) patterns.

1 Introduction

Software architecture descriptions are the means through which software designers and software implementers communicate about the architecture; nevertheless this piece of information (the software architecture) is often a source of confusion and miscommunication. There are several reasons for this; one of them is that software architecture documents mostly describe *how* the software should be built, and little *why* it should be built that way.

Most problems in software design are not unique to the system they must be solved for; they have probably been tackled by dozens of software designers before. So designing everything from scratch can be seen as double work. And When one finds a solution to a particular design problem, in what way can one know that it solves the problem adequately?

A solution to these problem is the concept of patterns.

1.1 What is an architectural pattern?

A *pattern* is a re-usable concept that has been applied in several working systems. It is not a litteral description of the used structure, rather an abstraction from it that can be used in other systems. A pattern thus provides the architect with a recipe to compose a certain architecture or part of an architecture.

A *pattern language* is a collection of patterns, with clearly stated interdependencies. With a *generative* pattern language, a whole family of architectures can be generated.

One can subdivide patterns into *architectural* and *design* patterns. This paper focuses on the former; these are patterns that apply to complete systems. They are mostly non-OO. Another class of patterns is that of the *design* patterns; these are applied on a lower level and are often formulated in an object-oriented way.

Architectural patterns need to be classified and described in a standard way in order to be effectively usable in software architectures. If architects speak the same 'pattern language' (i.e. if one of them calls a pattern by it's name, others know exactly which pattern he/she means), many misunderstandings can be avoided, resulting less energy spent on design and better maintainability.

2 History

The history of architectural patterns is quite short, as the subject is quite young. There are three milestones in the evolution of architectural patterns. The first is the publication of several books by Christopher Alexander ([Alexander77,Alexander79]). The second is constituted by the first use of patterns in Smalltalk by Kent Beck and Ward Cunningham, and the publication of the book *Design Patterns* by four authors also known as the *Gang of Four* ([Gof94]). The third is the publication of the POSA books.

2.1 Christopher Alexander

Christopher Alexander is an architect (not a software architect, but a *bricks 'n walls* architect) who is considered to be the father of *patterns.*

Alexander proposes the idea of patterns. Each pattern expresses a relation between a context, a 'system of forces' that is inherent to the context, and the 'solution to the problem': the architectural concept that works in the particular context. The 'system of forces' can be seen as a set of requirements formulated as forces that define a goal for the design of an architecture. A quote from his book [Alexancer77]: *Each pattern is a rule which describes what you have to do to generate the entity which it defines.*

An important feature of a pattern *language* is it's *generativity*. Generativity of a pattern language means that the collection of patterns generates a whole family of architectures; without describing a single one of them explicitly.

Alexander describes architectural patterns (be it non-software patterns) in the following way:

- The **name** of the pattern
- A picture, which shows an **example** of the pattern.
- An introductory paragraph, which sets the **context** for the pattern
- A headline in bold type that gives the essence of the **problem** in one or two sentences
- The body, the longest section: background, motivation, variations
- The **solution**, in bold type: how to solve the problem
- A diagram, that shows the solution as a labeled picture
- A paragraph that ties the pattern to related patterns

This form is known as *Alexandrian form* and is still used as a way to describe patterns. It is quite informal: the description of a pattern is in the form of a story being told, with explanatory pictures and 'delimiting diamonds'. This is because one of Christopher Alexander's motives was to give 'ordinary people' (i.e., non-architects) a tool to communicate with architects; to achieve this, the tool had to be in ordinary people's language.

2.2   Object-oriented patterns; Gang of Four

In 1987, Ward Cunningham and Kent Beck bring Alexander's ideas in practice in the field of software design. They design a small 5-pattern object-oriented pattern language to make things easier for novice Smalltalk programmers. They present their work on the OOPSLA'87 conference.

In 1994, four authors also known as the *Gang of Four*[1] describe 23 object-oriented software patterns in their book *Design Patterns* [GoF94]. They do this in a way that is far more detailed than *Alexandrian form*, and somewhat more formal.

The GoF come up with a classification of patterns based on two criteria. The first criterion is what *purpose* the pattern serves; patterns can have either *creational*, *structural* or *behavioral* purpose. The second criterion is *scope*: whether the pattern primarily applies to classes or objects.

The GoF employ a far more elaborate way of describing patterns than Alexandrian form (13 as opposed to 5 elements of description per pattern). As opposed to [Alexander77] and [Alexander79], [GoF94] was written for software designers rather than for 'normal people'. However, as can be seen later in this paper, in high-level architecture descriptions it is often preferrable to use Alexandrian form instead of one of the more complicated forms.

The patterns described in *Design Patterns*, and the patterns invented by Cunningham and Kent are technically speaking not architectural; they are object-oriented and are best used in lower-level design. However, one of the articles in the next section (see section **??**) is about *deriving* an architecture from the patterns from *Design Patterns*.

---

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

2.3 Pattern-Oriented Software Architecture

There are two books bearing this title (with different subtitle though); the first one was published in 1996 ([POSA1]) and the second was published in 2000 (POSA2). [POSA1] looks at how patterns occur on three different levels; in software architecture, in everyday design, and in idioms (which describe how a particular design pattern is implemented). [POSA2] does the same for patterns associated with concurrency and networking.

These books show one fundamental change with respect to the previous milestone: patterns now aren't necessarily necessarily object-oriented. Furthermore, they make a clear distinction between architecture-level patterns and design-level patterns.

3 Three articles

To come to a statement of how architectural patterns have evolved since their first occurrence, this paper looks into three articles with more detail. These papers all approach the subject of architectural patterns from a different side, and thus give an overview of how architectural patterns evolve.

3.1 [Beck94]: Patterns generate architectures

Kent Beck describes how an architecture can be *derived* with the help of architectural patterns:

> Describing an architecture with patterns is like the process of cell division and specialization that drives growth in biological organisms. The design starts as a fuzzy cloud representing the system to be realized. As patterns are applied to the cloud, parts of it come into focus. When no more patterns are applicable, the design is finished.

To demonstrate this method, he derives a drawing program ('Hot Draw') by applying subsequently the patterns *Model-View-Controller*, *Composite*, *Objects for States*, *Editor*, *Observer*, *Collect Damage*, *Update at User Speed* and *Wrapper*. All of these (except *Editor*) are from 'the Design Pattern Catalog' , which is a reference to [GoF94].

Beck doesn't use the format used in [GoF94]; instead, he uses the Alexandrian form to describe the patterns.

This article marks the first stage in the evolution of architectural patterns: architectures can be derived from patterns that are described in *object-oriented* terms. The parallel to be drawn is that with [GoF94] and Kent Beck's own work on patterns for Smalltalk.

3.2   [Shaw96]: Classification of architectural patterns

In this article, Mary Shaw and Paul Clements propose a classification standard for patterns. To demonstrate their method, they classify styles described by others. Their goals are to (quote):

- Establish a *uniform descriptive standard* for architectural styles - make the vocabulary used to describe styles more precise and shareable among software architects.
- Provide a systematic *organization to support retrieval* of information about styles
- *Discriminate* among different styles - bring out significant differences that affect the suitability of a style for various tasks; show which styles are refinements of others.
- Set the stage for organizing advice on *selecting a style* for a given problem

Instead of only pointing out that a pattern has *creational*, *structural* or *behavioral* purpose, they come up with a far more complicated method based on five *feature categories*. By doing so, the authors hope to contribute to architectural styles becoming the *lingua franca* of architecture design.

As opposed to the previous article, the patterns described aren't necessarily object-oriented, for example, *Dataflow network*. The patterns are also referred to as *architectural styles*, which gives an indication that these patterns are clearly something else than lower-level *design patterns*. The parallel to be drawn is one with the publication of the [POSA1,2] books, which marks the second stage in the evolution of architectural patterns.

3.3   [Buschmann01]: A Pattern Language for Distributed Object Computing

In the introduction to his article, Buschmann says:

> A general critique of Gang-of-Four and POSA style pattern descriptions is their length and depth of details. In particular it is critiqued that these descriptions do not highlight the pattern itself, but its implementation: its general idea as well as its connection and integration with other patterns likely gets hidden.

Subsequently, Buschmann describes a number of patterns that were described in the book *Pattern-Oriented Softwrae Architecture - Patterns for Concurrent and Networked Objects* [POSA2] in Alexandrian form. He adds the notion that 'the language is not intended to replace these patterns' existing descriptions, but to complement them with an additional perspective'. Apparently, Alexandrian form is a better way to describe patterns in a high-level fashion, while low-level (detailed) design patterns are best described by using POSA or GoF form.

With this article, Buschmann puts the accent on where it was in the beginning: as Alexander did, he formulates patterns in a very readable way, and also focuses on the *generativity* of the pattern language as a whole. This is the third stage in the evolution of architectural design.

4   Conclusion

Architectural patterns have evolved from a *bricks 'n walls* architect's idea to a tool that can be used by software engineers to design architectures with; however, with a small detour they have stayed remarkably the same.

The transfer from architecture to software architecture yielded object-oriented patterns. Object-oriented patterns weren't sufficient; other ways of describing patterns were also needed. The generativity (that was also part of Alexander's definition) became important. Alexander's way of describing patterns was also re-adopted to give a good overview of pattern languages.

References

[Alexander77] C. Alexander, S. Ishikawa, M. Silverstein: *A Pattern Language*, Oxford University Press, 1977

[Alexander79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, 1979

[Beck94] K. Beck, R. Johnson: *Patterns Generate Architectures*, submission for ECOOP'94, 1994

[Buschmann01] F. Buschmann: *A Pattern Language for Distributed Object Computing*, 2001

[GoF94] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

[Lea94] D. Lea, *Christopher Alexander: An Introduction for Object-Oriented Designers*, ACM Software Engineering Notes, January 1994

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley and Sons, 1996

[POSA2] D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann: *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects*, John Wiley and Sons, 2000

[Shaw96] M. Shaw, P. Clements: *Toward Boxology: Preliminary Classification of Architectural Styles*, SIGSOFT 96 Workshop, 1996

On the evolution of architectural patterns
from the original concept to the architects toolkit

Sjouke W. Boersma

Department of Computer Science
University of Groningen
s.w.boersma@student.rug.nl

Rick van Buuren

Department of Computer Science
University of Groningen
r.d.buuren@student.rug.nl

Abstract. Throughout the years the size of software products has grown. A development which started with small individual programs has evolved to large and complex cooperating programs. This development increased the need for a structured way to design the software architecture. After the introduction of design patterns in software architecture, patterns gained a lot of popularity within a relatively short period of time. Nowadays, patterns are widely used within the software architecture and development on many levels. In this paper we will analyze how the usage of patterns has evolved from single patterns to combined patterns and pattern languages.

1   Introduction

Throughout the years software products have grown in size and complexity, as have the computers on which the software is running. From relatively simple and rigid software products, it has evolved to flexible and complex products with many possibilities. Because of this growth a good structured architecture has become much more important than it was in the early days when there were just simple software products.

With the introduction of Object Oriented programming, a new paradigm was introduced. This paradigm brought a new way of handling software architecture. The question is how this paradigm could be used as effectively as possible.
Design patterns are describing the general idea of how to organize different objects to solve several common problems.

1.1   Christopher Alexander

A new architecture can be created in several ways. Christopher Alexander was the first who described a concept of using patterns to derive an architecture for buildings. By starting at the top and systematically working his way down to the finer-grained problems he completes his architectures.

1.2   Gang of Four

In software architecture there are several problems which often occur. In almost every situation the details are different, but the core of the problem is the same. Therefore the essence of the solution for these problems can be the same. It was the Gang of Four that saw a lot of resemblances with Alexander's problems and the problems in software. The intention of the Gang of Four was to record the experience in designing software architecture, and they tried to capture their architectural experience in patterns. These patterns are described in the book [4].

These pattern descriptions have a uniform layout, which makes the comparison between patterns easier. This layout has the following parts:

- the pattern name
- the problem description
- the solution
- and the consequences

Each of these parts has a clearly defined function which can not be omitted. By naming the patterns it makes it easier to communicate about patterns in such a way that it becomes part of the vocabulary of software architects and stakeholders. The problem description tells an architect when a certain pattern can be applied and can be a feasible solution to a problem. In the solution the idea behind the pattern is described in an abstract way, complemented with some simple examples. In the end the consequences of the usage of the particular pattern, the results and tradeoffs, are pointed out. Each pattern is described individually without any relations to other patterns.

It was the book [4] that inspired a lot of software architects to start using patterns. The problems described in the book look generally familiar and also the solutions are recognizable. This book made patterns known over a wide public. In the years after the books publication, people have elaborated patterns and found more ways to use them. Nowadays patterns are used on a wide scale and serve different purposes within software architecture.

2   Review of pattern usage

There are different methods where patterns are being used (architectural patterns, architectural styles, and architectural views for example). In this paper we will review four papers which are related to patterns and their usage – two papers are written in the time when patterns where just introduced in software architecture [2][5], and two other papers are more recent [1][3]. We will look at how patterns are being used and which relations they have. We will also look how pattern usage has changed trough time.

2.1 Patterns Generate Architectures

The article 'Patterns Generate Architectures" [4] describes how the "HotDraw" architecture is derived. The HotDraw architecture is an architecture used in a graphical drawing program. Many patterns used in the HotDraw architecture are described in [4]. The goal of the article is not to explain how the HotDraw architecture works, but how the architecture is derived. By not only knowing how HotDraw works, but also understanding how it is derived, the authors claim that it gives a deeper understanding which an expert should have.

Each pattern which is used in HotDraw is shortly explained in the article. Each description has four sections:

- The Preconditions, describes which conditions should be met before the described pattern can be applied.
- The Problem, a short description of the problem which is addressed with this pattern.
- The constraints, this section describes the constraints of the usage of this pattern. The usage of a pattern often implies that other design decisions are excluded by this pattern, or that the usage of a pattern has certain consequences.
- The Solution, a short description of the solution, sometimes with a diagram in this article.

Each section of the description is pointed at the current situation in the HotDraw example in such a way that it becomes clear which problem is faced, what the problems and constraints are and how this problem is solved. Below the summation is shortly described how the framework is affected by the usage of the described pattern. At the end of the description of all the used patterns, all the faced problems which were encountered by the creation of the drawing tool were solved.

For each problem which is encountered in the creation of the architecture an pattern is applied to solve the problem. By continuing this process until all problems are solved the architecture is derived. During this process the focus lies on how to select the appropriate pattern. Also the motivation for the choice is shortly described in a couple of sentences.

2.2 Classification of Architectural Styles

An architectural style can also be called an architectural pattern. There are only differences between their descriptions. Therefore we will use the term architectural patterns trough this article.

An architectural pattern is focussed on how a system or subsystem is composed and how the different parts are connected to each other. Design patterns are focussed on the architecture of Object Oriented software. Architectural patterns are describing concepts about the global organisation of a software architecture. These patterns are not limited to the Object Oriented paradigm but have a more global view.

In the article [5] several architectural patterns are organized and classified into several categories. By organizing and classifying several architectural patterns it gives a uniform descriptive standard. This helps in the communication about architectural patterns. Also by classifying and organizing all the architectural patterns this makes it easier to identify the differences between the different styles and gives a good overview on all the possibilities. This makes it easier to select a style based on the design problem which is faced.

The architectural patterns are classified into five major categories. Within these categories finer-grained classifications are made to point out the differences.

- Constituent parts: components and connectors
  In this section the type of components and the way they interact with each other are mentioned. Connectors and components are the primary building blocks of architectures.
- Control issues
  This section describes how components work together and how control is passed trough. This classification has three subsections: Topology, synchronicity and binding time.
- Data issues
  This section describes how data is organized in the system and how it is moving around. The subsections in this part are: Topology, Continuity, Mode and Binding time.
- Control/data interaction
  This section describes the Shape and the Directionality of the control flow.
- Type of reasoning
  This section describes the type of reasoning which is the basis for the architectural style.

This classification gives a good insight in the different architectural patterns and what the characteristics of the architectural patterns are.

2.3   A Pattern Language

In [3] Buschmann describes a pattern language. This pattern language is derived from patterns from the book 'Pattern-Oriented Software Architecture – Patterns for Concurrent and Network Objects'. Buschmann tries to rewrite these patterns and adds his own perspective. He uses this approach to give a better description on the pattern itself. He mentions that it's not his intention to replace the existing patterns, but just to give a better picture of the pattern language this way.

The 17 patterns Buschmann describes in his paper are classified into four areas. Namely service access and configuration, event handling, synchronization and concurrency. Buschmann uses a two-step process to connect all the patterns and create a pattern language. Firstly, he identifies the relationships between the patterns. He looks at the pattern descriptions to clarify which of the relations should be preserved and which ones should be omitted. Secondly, he defines the order of the

patterns. He does this by looking at which patterns are entry points, which patterns follow and which patterns are leafs. Then he looks which patterns are architectural patterns and defines these as the entry points of the pattern language.

Buschmann summarizes his pattern language in the diagram shown in figure 2.1.
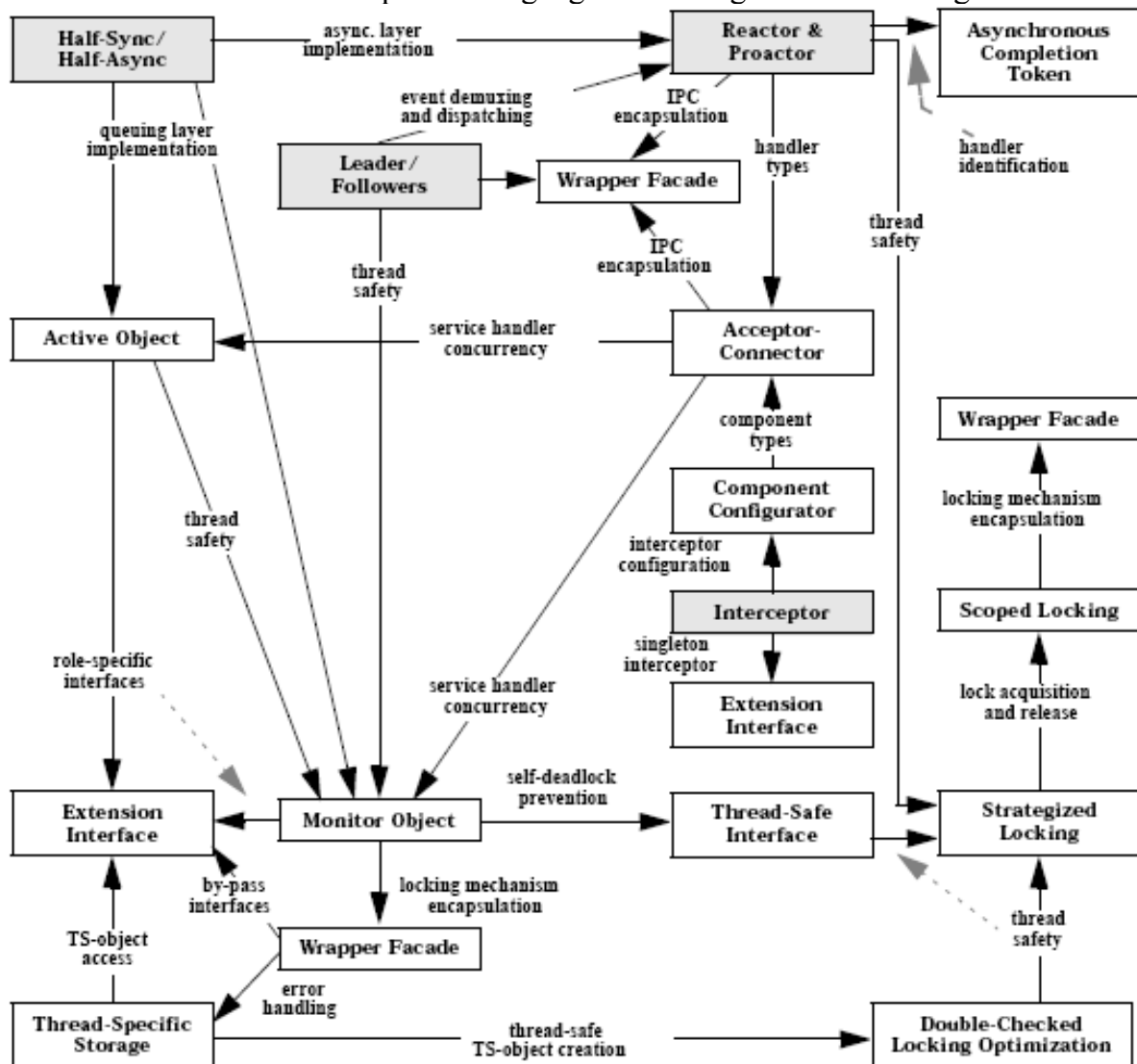

Figure 2.1 Diagram with Buschman's pattern language (acquired from [3]).

All of the patterns in the diagram are found as components and are connected to each other with so-called connectors. In this diagram we can find five patterns which are in Alexandrian form: Thread-Safe Interface, Double-Checked Locking, Strategized Locking, Scoped Locking and the Wrapper Facade. These patterns can be found in the diagram above at the bottom right. The exact descriptions of these patterns can be found in [3].

2.4   Architectural Patterns Revisited

In the article [1] several architectural patterns are grouped and classified in views. Each view looks at certain aspects of an architecture. Each architectural view raises important questions, which are pointing at the main differences between the described

architectural patterns. After the questions the architectural patterns are shortly described, in which the description is focused at the distinctive aspects of the patterns. The architectural views are giving the architects a way to talk about a certain aspect of the architecture. The combination of the architectural views and the architectural patterns are creating a language to talk about certain aspects of software architecture which are very hard to communicate without them.

3   The usage of patterns through time

The real start of pattern usage in software architecture was introduced with the book [4] which changed the way of thinking about software architecture. The vision was to build an architecture with the help of patterns. The article [2] describes how an architecture is generated and how the patterns are being applied. This is done by looking at the problem statements and applying patterns so the architecture meets the requirements. By continuing to apply patterns the architecture is generated.

In the article [5] several architectural patterns are classified in such a way that the software architect can more easily find the pattern to use in the specific situation which is faced.

In 2001, Buschmann described a pattern language in which several patterns are combined [3]. The language focuses on Patterns for Concurrent and Networked Objects and the combinations which are made between the different patterns. There is added value in this pattern language in the way patterns are combined, which cannot be found in the individual patterns. Figure 2.1 shows how the different patterns are connected to each other. In this pattern language, design patterns are used together with architectural patterns.

In [1] architectural views are described. Each view groups several architectural patterns and is clearly showing the differences between the individual patterns. This language covers many aspects of software architecture.

Trough time more things have been described in patterns. In 1994 the Gang of Four introduced design patterns. These design patterns were capturing the architectural knowledge of Object Oriented design. In a similar way architectural patterns are capturing architectural knowledge on software architecture. On this basis Buschmann describes a pattern language which shows how design patterns and architectural patterns can be combined together and what the important relationships are between them. In the description of the created combinations more architectural knowledge is captured.

In the article [1] another view on pattern languages is described. The foundation of the language lies in architectural views which are the starting point of the language. Each view covers a certain aspect of the software architecture, described in several architectural patterns. This is another approach as used in [3] where the architectural patterns are linked with design patterns. The difference between the approach in [3] and [1] is that [3] only covers one aspect of software architecture and [1] is more a general description of several architectural aspects and gives a much broader view on the subject.

4   Conclusion

Since the introduction of patterns, they have become of great influence in the software industry. Patterns have created a lingua franca in software architecture and have become a common way to document architectural knowledge.

Trough time patterns have been used on different levels within the field of software architecture. With the growth of the software and the software architecture also the levels on which patterns are used have increased.

There is still much discussion about how patterns can be used and which methods and techniques are better. Patterns are not the answer to all questions. Patterns can help give structure to thoughts and transfer knowledge. But the fact that patterns are used doesn't imply that the architecture will always meet its requirements. There is a good chance that the future will show even more methods where patterns can be used. But patterns themselves have become one of the main tools of the software architects to perform their jobs.

5   References

[1] Paris Avgeriou, Uwe Zdun: Architectural Patterns Revisited – A Pattern Language, the European Pattern Languages of Programming (EuroPLOP) 6th–10th July 2005, Irsee

[2] Kent Beck, Ralph Johnson: Patterns Generate Architectures, 1994.

[3] Frank Buschmann, Kevlin Henry: A pattern Language for Distributed Object Computing, 2001.

[4] E. Gamma, R. Hel, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[5] Mary Shaw, Paul Clements, Toward Boxology: Preliminary Classification of Architectural Styles in Proceedings of the Second International Software Architecture Workshop, pp. 50-54, 1996.

Alias-Free Digital Synthesis using Band-Limited Impulse Trains

Ilja Plutschouw and Piter Pasma

Department of Computational Science
Rijksuniversiteit Groningen
Blauwborgje 3
9747 AC Groningen
`i_plutschouw@yahoo.com`, `ritz_rvl@yahoo.com`

Abstract

The sawtooth, square and triangle waveforms are the basic waveforms produced by classic analog synthesizers. When trying to synthesize these in a digital medium, they are subject to aliasing. This paper describes a method to digitally synthesize these waveforms without aliasing, using band-limited impulse trains.

Keywords: BLIT, band limited impulse train, aliasing, synthesis, sawtooth, squarewave, trianglewave, signal processing

1 Introduction

Analog synthesizers have a typical smooth, warm "old-school" sound. They were most popular in the 1960s and 70s [Kraftwerk 74], but because they were hard to program and expensive, they were replaced by cheaper and more userfriendly digital synthesizers. While the analog synthesizers used electronic circuits to generate the waveforms, the digital ones used wavetables with sampled sounds. The digital synthesizers where better at emulating real world instruments. However, when electronic house music became more and more popular, there came a revival of the sound of the classic analog synthesizers. Because these instruments were only produced in limited amounts, and thus became rare very quickly, people started to look for other methods to create these sounds. Wavetable techniques were not very suitable for this, because they lack the versatility of the electronic circuits. Software synthesizers appeared to be a better solution, because they can actually simulate the electronic circuitry of the analog synthesizer.

Sounds from an analog synthesizer usually consist of one or more basic waveforms, which are rich in harmonics, played at the frequency at which the
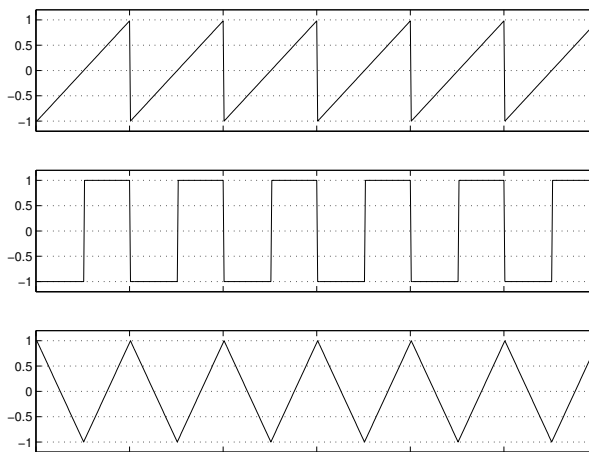

Figure 1: Basic waveforms: a sawtooth, square and triangle wave.

note that is playing, with a (resonance) filter and a volume envelope applied over it to give a more specific timbre to the instrument.

Emulating a filter or applying a volume envelope over a digital waveform is usually not much of a problem, creating the basic waveforms, however,

is. The basic, or classic waveforms that are most often used are the sawtooth, square and triangle waveforms (see figure 1).

When emulating these sounds on a digital device, we run into problems because they contain an unlimited amount of frequencies or harmonics, which can be seen as the discontinuities in the waveform or in their slope. Since digital sounds are inevitably sampled at a finite sampling frequency, the analog signal must be band-limited to less than the Nyquist frequency (defined as $F_s/2$, half of the sampling frequency) before sampling to obtain the corresponding digital (discrete-time) signal. Without proper band-limiting these signals will contain aliasing due to having to round off the discontinuities to the nearest sampling point (an example of this can be seen in figure 5, which will be further discussed in section 2).

1.1 Definitions

DC The DC is the average value of a signal. In most cases one wants the DC to be 0, so that the waveform is centered around the origin. In the frequency spectrum the DC is represented as the 0th "harmonic" at 0Hz.

Harmonics A frequency is harmonic if it is an integer multiple of the fundamental frequency. The fundamental is the first harmonic. According to Fourier Theory, all periodic functions are made up of a summation of harmonic sine-functions.

Impulse The impulse, or Dirac-delta function is defined as: $\delta(x) = \infty$ if $x = 0$ and $\delta(x) = 0$ otherwise. $\int_{-\infty}^{\infty} \delta(x) dx$ is defined as 1. The Fourier transform shows that the frequency spectrum of the impulse contains all frequencies.

Impulse train The continuous impulse train is an infinite succession of equally spaced impulses. It is defined as

$$CIT(x) = \sum_{l=-\infty}^{\infty} \delta(x + lT)$$

where $T = 1/f$ is the period, or distance between the impulses. The frequency spectrum of the impulse train contains all harmonics in equal power.

Nyquist frequency To be able to reconstruct the original continuous waveform, the sampling rate must be at least twice as high as the highest frequency component in the signal. This frequency is called the Nyquist frequency.

Sampling The process of converting a continuous time signal to a discrete time signal. A sample is one of the discrete values attained in this process, the frequency at which the samples are taken from the continuous signal is called the sampling frequency, or f_s [Wikipedia].

Sinc The sinc function (figure 2) is also known as the interpolation function or filtering function, and is defined by $sinc(x) = sin(\pi x)/\pi x$. The frequency spectrum of the sinc function contains all frequencies up to π. It can therefore be interpreted as a band-limited impulse.
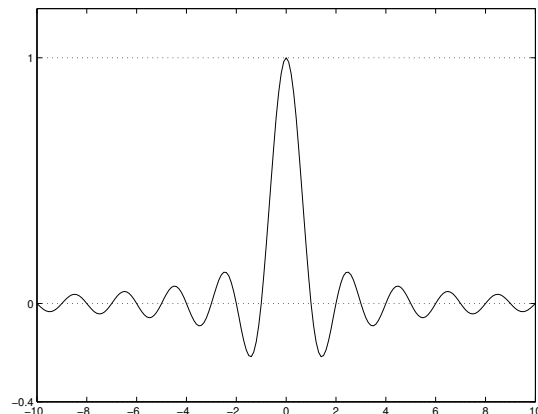

Figure 2: The sinc function.

2 Digital aliasing

A sampled digital sound has limitations on the number of frequencies that it can represent. This limitation is determined by the sampling rate of the signal. Nyquist states that the maximum frequency possible in a digital (discrete) sound is half of the sampling frequency. Frequencies higher than this cannot be reconstructed correctly, and result in aliasing. These are audible distortions of the sound. In digital recording hardware this problem is addressed by adding a lowpass filter which cuts the

frequencies above Nyquist before doing the sampling.

When constructing digital waveforms, the same problem occurs. As long as band-limited signals are used (for example a sine wave which consists of only its fundamental frequency) there is no problem. More complex waveforms like the popular sawtooth, square and triangle waveforms have an infinite number of harmonics (the harmonics of a sawtooth can be seen as the vertical lines in figure 3). This can result in heavy aliasing and care should be taken to get rid of the harmonics above the Nyquist frequency.
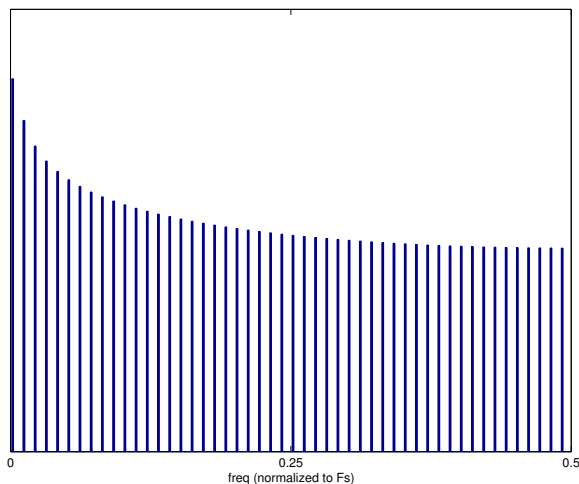
2.1 Examples of aliasing

Figure 3: perfect or analog sawtooth frequency spectrum.

The harmonics of a sawtooth drop off in power with $1/N$, where N is the number of the harmonic (see figure 3). We get into trouble when the frequencies of the harmonics go above the Nyquist frequency. Because these higher frequencies can't be represented in the sampled signal, they will "bounce" back into the below-Nyquist range (figure 4), where they don't belong and therefore will be perceived as noise or distortions.

Because the harmonics of sawtooth-waves with high base-frequencies can go above the Nyquist frequency after only a small number of harmonics, the power of these harmonics hasn't dropped very
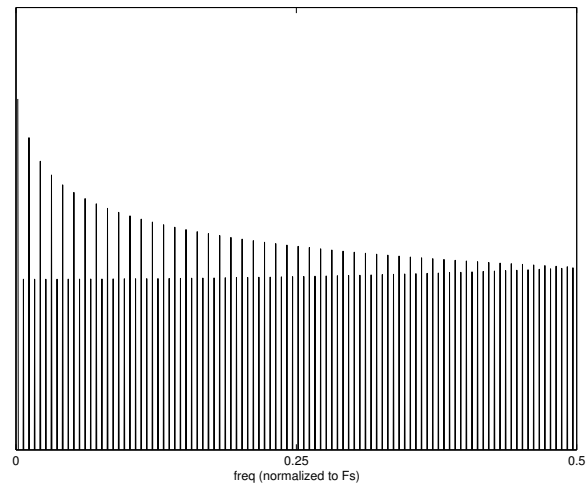

Figure 4: digital sawtooth frequency spectrum, clearly displaying the aliasing artifacts.

much yet. Therefore aliasing is much worse in high-frequency sawtooths than in low ones. As every effect in the frequency domain has a dual effect in the sample domain, we can observe this aliasing as the "wobbling" in the amplitude of sampled high-frequency sawtooths, as seen in figure 5.

The same kind of aliasing occurs with square and triangle waveforms, with the exceptions that the squarewave consists of only odd harmonics and the trianglewave has a quicker drop-off in harmonics ($1/N^2$).

2.2 Previous solutions to aliasing

Many solutions have been tried to overcome the aliasing problem, some were more succesful than others. We will discuss a few of them.

- Additive synthesis works by adding up a number of sines, one for every harmonic. Therefore it is trivially band limited simply by not generating harmonics higher than $F_s/2$.

- Wavetable synthesis [Mathews 1969] is implemented by putting one period of the desired waveform into a wavetable, sampled at a very high sampling rate. To get the waveform with the frequency F, we skip through the wavetable with steps of $N f_0/F_s$, with N the wavetable length, f_0 the base frequency
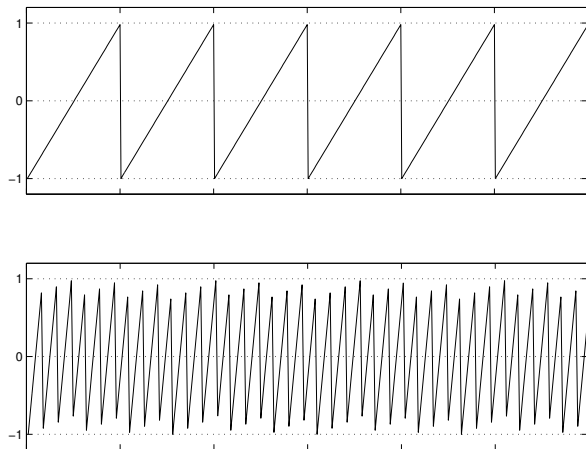
Figure 5: Two classic sawtooth waveforms. Above a low frequency and below a high frequency.

and F_s the sampling frequency. Because the wavetable "addresses" have a fractional part, we need to interpolate in some way. If the step is larger then one, we need to take care that there are not too much harmonics in the wavetable to produce aliasing. The wavetable is usually precalculated using additive synthesis.

- In [Moorer 1975] Discrete Summation Formulae are proposed for generating periodic signals that consist of exponential ramps of harmonics. With this formula one can generate a wide class of band limited waveforms. It is not possible to generate the exact classic waveforms as discussed in this paper using these formulae, although it is possible to get a reasonable sounding approximation.

3 Creating Classic Waveforms using Impulse Trains

We can integrate the impulse train in various ways to create classic waveforms. Because integration is a linear operation it can be considered as a filter, so no extra frequencies will be added. Therefore, if we find a way to create a band limited impulse train, which we will do in the next section, we can also create band limited classic waveforms.

3.1 The Sawtooth

Of the three basic waveforms, the sawtooth is created in the most straightforward way, by integration of an impulse train:

$$Saw_{CTS}(t) = \int_0^t CIT(\tau) - C_1 d\tau$$

Where $CIT(\tau)$ is a continuous-time impulse train with period τ, and $C_1 = \int_0^T CIT(\tau)d\tau$ is the DC component (average offset value) of the impulse train. When converted to discrete time, a discrete function "BLIT" is used to generate the discrete, band limited impulse-train:

$$Saw_{DTS}(n) = \sum_{k=0}^{n} BLIT(k) - C_2$$

C_2 is the average value of BLIT, it should be subtracted to keep the integration from ramping off to infinity. This function can be implemented by a "running" or "cumulative" sum.

3.2 The Rectangle

The rectangle wave is a generalized version of the square wave that can have different duty cycles (a duty cycle is the percentage of the period where the signal is positive). The square wave the special case where the duty cycle is 50%. The continuous version can be computed as:

$$Rect_{CTS}(t) = \int_0^t CIT(\tau) - CIT(\tau - t_0)d\tau$$

t_0 controls the duty cycle in range [0,Period], with $t_0 = Period/2$ generating the classic square. Because the rectangle function takes the difference between two pulse trains, the DC component is zero. Note that this formula can also be interpreted as the difference between two sawtooths, one delayed (phase-shfited) by an amount of t_0.

When transformed to discrete time we get:

$$Rect_{DTS}(n) = \sum_{k=0}^{n} BLIT(k) - BLIT(k - k_0)$$

Where k_0 controls the duty cycle.

3.3 The Triangle

The triangle is generated by integrating the angle function:

$$Tri_{CTS}(t) = \int_0^t Rect_{CTS}(\tau) - C_3 d\tau$$

Where C_3 is the DC component of the rect wave: $C_3 = \int_0^t Rect_{CTS}$.

The discrete version is:

$$Tri_{DTS}(n) = \sum_{k=0}^n Rect_{DTS}(k) - C_4$$

where C_4 is the DC offset. These DCs s] in theory be zero, but in practice they ar pendent on the initial condition of the integration, and the duty cycle of the rectangle wave: $C_4 = k_0/Period + C_5$, where C_5 is the initial condition of the integration.

A way to resolve the problem of nonzero DCs is to use a leaky integrator. The triangle function produces a triangle with inappropriate amplitude. We want this to be the same as the amplitude of the rectangle wave. Therefore the signal must be scaled by a function that is dependent on the frequency and duty cycle of the rectangle wave:

$$Tri_{DTS}(n) = \sum_{k=0}^n g(f,d)(Rect_{DTS}(k) - C_4)$$

$$g(f,d) = \frac{2f}{d(1-d)}$$

4 BLIT synthesis

We will now describe different methods to synthesize band limited impulse trains.

4.1 The sinc$_M$ function

We can obtain a band limited signal by applying an anti-aliasing filter before sampling. The perfect anti-aliasing filter, h_s, removes all frequencies above $f_{Nyquist}$. It has a continuous-time impulse response that is a sinc function with a zero-crossing interval of one sample (see figure 2).

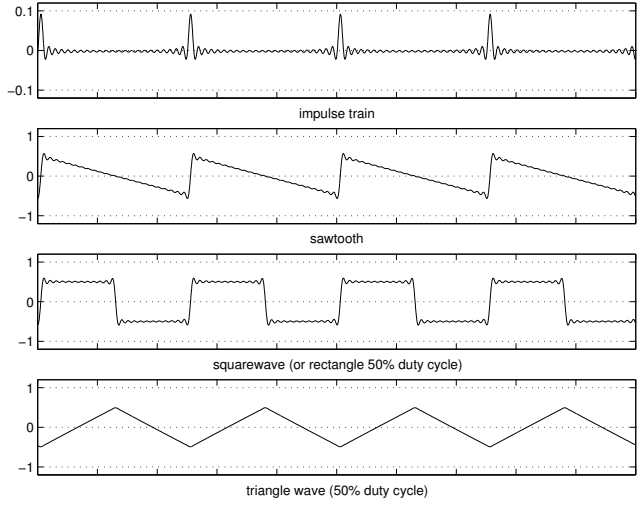$$h_s(t) \doteq sinc(F_s t) \doteq \frac{sin(\pi F_s t)}{\pi F_s t}$$

Figure 6: Four classic waveforms, limited to only a few (37) harmonics to emphasize the effect of bandlimiting on the waveform.

The ideal unit-amplitude impulse train with period T_1 is given by

$$x(t) = CIT(t) = \sum_{l=-\infty}^{\infty} \delta(t + lT_1)$$

Applying the anti-aliasing filter h_s to this signal gives

$$x_{BL}(t) = (x * h_s)(t) = \sum_{l=-\infty}^{\infty} h_s(t + lT_1)$$

$$= \sum_{l=-\infty}^{\infty} sinc(t/T_s + lP)$$

where $T_s = 1/F_s$, the duration of one sample and $P = T_1/T_s$, the period of the impulse train in samples. Since x_{BL} is now band limited, it can be safely sampled without aliasing:

$$BLIT(n) \doteq x_{BL}(nT_s) = \sum_{l=-\infty}^{\infty} sinc(n + lP)$$

This can be interpreted as a time-repeated sinc function, a sinc-train. It can be shown that this sinc-train becomes

$$BLIT(n) = \frac{M}{P} Sinc_M(\frac{M}{P}n)$$

with P the period in samples, M the number of harmonics and:

$$Sinc_M(x) \doteq \frac{sin(\pi x)}{M sin(\pi x/M)}$$

This is a closed-form expression for the sampled band limited impulse train, and can be used for synthesis. Care has to be taken of course for the limit where the $Sinc_M$ becomes $0/0$, in this case the limit goes to 1. M is always odd because an impulse train has one "harmonic" at DC (the 0th harmonic or the average offset of the signal), and an even number of non-zero harmonics, provided no harmonic is allowed at exactly half the sampling rate.

We can define the maximum number of harmonics M in terms of the period P as

$$M = 2\lfloor P/2 \rfloor + 1$$

in other words, M is the largest odd integer smaller than P in samples.

4.2   Sum of windowed sincs (BLIT-SWS)

This method is loosely based on the wavetable method mentioned earlier. It is basically a band limited periodic wavetable synthesis of an impulse train. Bandlimited interpolation is used to convert the sampling rate of a discrete-time unit sample impulse train with an integer period to the desired pitch. This conversion causes each impulse to be replaced by a windowed sinc function, again because of the anti-aliasing filter h_s and the simple properties of the impulse train.

The idea is to use, instead of a periodic wavetable, an overlapping of (precalculated) windowed sinc-functions, one sinc-pulse for every impulse in the train. Because these windowed sinc-functions are usually sampled at a varying (non-integer) phase, it is helpful to have an oversampling factor in these precalculated windowed sinc-tables. Because the windowing causes a finite fall-off in the sinc-function, some aliasing is inevitable. We can however, control this by our choice of window.

We can optimize this method by comparing the number of overlapping windowed sinc-functions with the number of harmonics of the BLIT. At high frequencies the number of overlaps becomes large, while the number of harmonics becomes small.

Then it is a good idea to switch to simple additive synthesis instead of summing windowed sincs. The tradeoff here occurs depending on the computational cost of summing sines versus overlapping sincs from wavetables. Also the amount of accuracy comes into play because it affects the choice of window length and therefore the number of overlapping sincs at a given frequency.

4.3   Efficiency and implementation details

The $sinc_M$ method computes a band limited classic waveform in constant time (per sample), however the drawback is that the computational cost is determined by two sines and a divide instruction per sample. Also care must be taken for the possible division by zero, which we need to test for. Depending on the hardware of the implementation, these drawbacks may or may not be a big problem.

The BLIT-SWS method can be very fast when memory-lookups are cheap and the calculations of a $sinc_M$ are too expensive. If the algorithm is optimized by switching to additive synthesis for high frequencies, these can also be implemented quite cheap. If an inverse FFT is used for the additive synthesis, this brings along new problems like the FFT windowsize. So, for a high number of harmonics, the complexity is linear in the frequency, and when switching to additive synthesis it becomes linear in the number of harmonics, which is equal to about half of the period, so inversely linear in the frequency. When inverse FFT is used, the "additive" synthesis can even become even logarithmic in the number of harmonics.

5   Open problems

The $sinc_M$ method can be greatly optimized by using differential sine-oscillators instead of actual sine-functions. These differential oscillators use their previous values to determine the value of the sine at the next sampling point. Care has to be taken with the stability of these oscillators, a trivial Euler-approximation would not do, for example. Some preliminary experiments with reasonably stable oscillators (computational cost: two multiplications and two additions per sine per sample) have been quite successful, however it remains to be seen what happens to the stability of these differential oscillators when, for example, the frequency

is slided.

6 Conclusion

The sound of the old analog synthesizers is hard to emulate. Many implementations are careless about band limiting the basic waves, because of real-time issues, and use heavy filtering to hide the effects of it. Using the method described in this paper, it is possible to get much clearer sounds, with little overhead. This makes it highly attractive for use in real-time analog synthesizer emulation.

7 References

[Brandt 2001] E. Brandt. "Hard Sync Without Aliasing"

[Mathews 1969] Mathews, M.V. 1969. "The Technology of Computer Music." Cambridge, MA: MIT Press

[Moorer 1975] Moorer, J.A. 1975. "The Synthesis of Complex Audio Spectra by Means of Discrete Summation Formulae." J. Audio Eng. Soc., 24(Dec.):717-727 (Also available as CCRMA Report No. STAN-M-5).

[Stilson 1996] T. Stilson and J. Smith 1996. "Alias-Free Digital Syntheses of Clasic Analog Waveforms". In "Proc. International Computer Music Conference". International Computer Music Association.

[Kraftwerk 1974] "Autobahn", Philips records.

[Wikipedia] http://en.wikipedia.org/wiki/Digital_signal_processing

A Comparison of Haskell and OCaml

Mark IJbema and Hilverd Reker

Abstract. This paper presents a brief overview of the differences and similarities between the programming languages Haskell (version 98) and OCaml (version 3.08). We provide the reader with a comparison according to a number of theoretical criteria. First we compare the major characteristics of both languages: their type system, evaluation strategy, and module system. Then we examine the available constructs for imperative and object-oriented programming, and discuss some less significant distinctions. Throughout this article, we present source code examples to illustrate various language features, and to help explain the criteria themselves.

Comparing (these) two languages — on more than just a syntactical level — helps one better understand their fundamental properties, and those of programming languages in general. Even though Haskell and OCaml are both functional programming languages and have a lot in common, there are still a number of important differences worth looking at.

1 Introduction

The Haskell programming language, named after the logician Haskell Curry, was created by a committee formed in 1987. The latest semi-official language standard is *Haskell 98* [1], which this paper refers to. Haskell is *purely functional*, which essentially means that it has no assignment statements and emphasizes the definition of functions. A distinguishing feature of Haskell is its use of *monads* (cf. §5.2) to simulate imperative constructs.

OCaml ("Objective Caml"), created in 1996, is a general-purpose programming language descended from the *ML* family. It is *impurely* functional, adding support for imperative programming using references and assignment statements. OCaml uses automatic garbage collection, as does Haskell (and most other modern languages).

This paper will study the main differences and similarities between Haskell and OCaml (release 3.08 [2]). We will *not* talk about practical issues such as available compilers and libraries, foreign function interfaces, documentation, and so on, but limit our discussion to a language-theoretical level.

2 Typing

Like most programming languages, both Haskell and OCaml are *typed*. Typed languages may have either *strong* or *weak* typing. In a strongly typed language, conversion between types requires the use of explicit conversion functions; in a weakly typed language, there may also be well-defined exceptions or an explicit type-violation mechanism such as *coercing*). Type *checking* may occur either at compile-time (*static* typing) or at run-time (*dynamic* typing). Both OCaml and Haskell are strongly and statically typed. Consider the following fragment of Haskell code:

```
add_one :: Integer -> Integer
add_one n = n + 1
```

We explicitly tell the compiler that `add_one` is a function from integers to integers, and then define the function itself. By contrast, here is a similar program in JavaScript, a weakly, dynamically typed language:

```
function add_one(arg) {
  return arg + 1;
}

a = add_one(0);   alert(a);
a = add_one("0"); alert(a);
```

When run, this program produces two alerts: the first contains the text "1" while the second one shows "01". Apparently, when its argument is a number, `add_one` returns the successor of that number, but when given a string, it returns that string with "1" appended. JavaScript converts these types *dynamically* at run-time. We

also see that the two subsequent assignments to `a` are perfectly legal, even though we first assign a number and then a string: JavaScript's *weak* typing allows this.

2.1 Pattern Matching and Algebraic Datatypes

As one would expect, both Haskell and OCaml are equipped with common built-in datatypes such as integers, characters, strings, and so on. There is also support for lists, arrays, tuples, and records. Much more interesting is their support for *algebraic* datatypes, illustrated by the following (Haskell) example:

```
data Tree = Leaf Char | Node Tree Tree
```

The above algebraic datatype declaration introduces a new *type constructor* `Tree` with two constituent *data constructors*: `Leaf` and `Node`. We now have a user-defined type for binary trees whose elements are either leaves containing a character, or internal nodes recursively containing two subtrees.

Suppose we are given a *value* of type `Tree` and we need a way to "decompose" it so we can access its parts. This is done by *pattern matching*; for instance, we could create a function `yield` that accepts a `Tree` and returns the left-to-right concatenation of its leaves:

```
yield :: Tree -> [Char]
yield (Leaf c) = [c]
yield (Node left right) =
  (yield left) ++ (yield right)
```

Incidentally, this example illustrates a few things about Haskell's syntax: the type `[Char]` means "a list of characters", and the value `[c]` denotes the list consisting of just element `c`. The binary operator `++` concatenates two lists.

To actually create a value of type `Tree`, we need to (recursively) apply either one of the data constructors we just defined. To illustrate,

```
print (yield t)
   where t = (Node (Node (Leaf 'a')
                         (Leaf 'b'))
                   (Leaf 'c'))
```

results in the value `"abc"` being printed. With only minor syntactic modifications, one can do exactly the same in OCaml.

Algebraic datatypes are also known as *sum of products* types. A product type is just the Cartesian product of some other types (e.g. `int * bool`), while a sum type corresponds to the set-theoretic idea of *disjoint union*. OCaml adds a special kind of sum type called a *polymorphic variant* type, which is explained in §5.2.

As a side note, a small advantage of Haskell over OCaml worth mentioning is its support for *list comprehensions*, which are best illustrated by an example. List comprehensions allow us to write, say, `[(x, y) | x <- xs, y <- ys]`, forming the Cartesian product of the lists `xs` and `ys`.

So far we have explicitly typed our functions, such as in the first example: `add_one :: Integer -> Integer`. That is not strictly necessary, even though many programmers still do so for clarity: Haskell's and OCaml's type systems can automatically *infer* the correct types for us. They employ a version of the so-called *Hindley-Milner* type inference algorithm, which essentially infers the most general type suitable for a given function.

2.2 Functions

Since Haskell and OCaml are functional languages, functions are *first class*, which means we can use them just like other values; we can pass them as arguments to other functions, return them from other functions, and so on. For example, Haskell's standard library contains a function `map` which applies a function to each element in a list. For instance,

```
map add_one [1, 2, 3]
```

yields `[2, 3, 4]`. We also see that function application is denoted by simply juxtaposing the function and its arguments (as it is in OCaml), so one writes things like `push item list` instead of `push(item, list)`. Both languages allow *currying* and *partial application*, demonstrated in the following Haskell code fragment:

```
mult a b = a * b
map (mult 2) [1, 2, 3]
  -- this results in [2, 4, 6]
```

Currying means we write `add a b` instead of `add (a, b)` in our function definition. We partially apply `add` to only its first argument, resulting in a "new" function that is equivalent to `mult b = 2 * b`.

In the preceding example we separately defined a function `mult` to be used by `map`, which

is not really necessary: OCaml and Haskell allow "anonymous" functions to be denoted via *lambda abstractions*. This means we could have written `map (\x -> x * 2) [1, 2, 3]` instead.

2.3 Polymorphism

Haskell and OCaml incorporate *polymorphic* types, which basically describe *families* of types. For example, while `[Char]` means "a list of characters", `[a]` denotes the type family "list of elements of type `a`", for every type `a`. Here the letter `a` is called a *type variable*, and this kind of polymorphism is known as *parametric* polymorphism. We can use it to improve upon our `Tree` datatype and the corresponding `yield` function:

```
data Tree a =
    Leaf a
  | Node (Tree a) (Tree a)

yield :: Tree -> [a]
yield (Leaf c) = [c]
yield (Node left right) =
  yield left ++ yield right
```

This polymorphic version of `Tree` is more general: leaves can hold any value, not just characters. Also, the `yield` function has been turned into a polymorphic one.

This brings us to an important difference between the two languages. There is another kind of polymorphism which Haskell supports, but OCaml does not: *ad hoc* polymorphism, better known as *overloading*. Overloading allows the same function to perform different operations on different types, such as a `+` operator that can add integers as well as real numbers. Haskell uses *type classes* as a structured way to control overloading. To illustrate, we can define a type class containing an equality operator:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Here we have stated that a type `a` is an instance of the type class `Eq` if there is an (overloaded) operation `==`, of the appropriate type, defined on it. We may then specify the actual behavior of `==` on a particular type. Assuming functions `integerEq` and `floatEq` are available, which compare two integers resp. floats, we can write:

```
instance Eq Integer where
  x == y  =  x `integerEq` y

instance Eq Float where
  x == y  =  x `floatEq` y
```

Having done this, we can now write expressions like `3 == 4` as well as `3.0 == 4.0`.

3 Evaluation Strategy

Haskell and OCaml differ fundamentally in their *evaluation strategy*, which is the set of rules defining how expressions are evaluated. Emphasis is typically placed on functions or operators: an evaluation strategy defines when and in what order the arguments to a function are evaluated, when they are substituted into the function, and what form that substitution takes. There are two kinds of (deterministic) evaluation strategies: *strict* and *nonstrict* evaluation.

3.1 Strict evaluation

In strict evaluation, the arguments to a function are always evaluated completely before the function is applied. Aside from a few technical details, this can also be called *eager* evaluation. The typical strict evaluation strategy is *applicative-order* evaluation, which we demonstrate by an example. Suppose we have a function *square* which returns the square of its argument, and we want to calculate the square of the square of three. An applicative-order evaluation of the expression `square (square 3)` proceeds as follows:

```
square (square 3)
square (3 * 3)
square 9
9 * 9
81
```

Applicative-order evaluation, which OCaml implements, evaluates the arguments before applying the function, as shown above.

Within strict evaluation, we can handle the argument expressions in different ways, the most well-known of which are *call by value* and *call by reference*. Call by value evaluates the argument expressions and assigns a local copy of the result to the corresponding parameter. This is what OCaml does (as a side note: it evaluates arguments right-to-left). OCaml also supports

call by reference, which means one can pass a *reference* value as an argument to a function. For this purpose the language offers a `ref` data constructor (cf. §5.1).

3.2 Nonstrict evaluation

Nonstrict or *lazy* evaluation implies that arguments to a function are not evaluated unless they are actually used in the evaluation of the function body. As an illustration of nonstrict evaluation, we evaluate `square (square 3)` using *normal-order* evaluation, which is the "opposite" of applicative-order evaluation:

```
square (square 3)
(square 3) * (square 3)
(3 * 3) * (3 * 3)
9 * 9
81
```

Although this type of evaluation is arguably more intuitive, it is only possible in a language which has no *side effects*, i.e. which is purely functional. (We will not go into details about this.) In the above example, we see that the function arguments are substituted directly into the function body, a process which is known as *call by name* evaluation. Call by name is rarely used in practice, since if an argument is used several times, it is re-evaluated each time. We see it happening in the example: the square of three is calculated twice, whereas we only needed to calculate it once under applicative-order evaluation. For this reason, Haskell employs *call by need*, a memoized version of call by name: if a function argument is evaluated, the result is stored for subsequent uses.

3.3 Strict functions

A *strict function* is a function which always evaluates all of its arguments. If we let $\perp$ denote the "value" of an expression that either produces an error or loops infinitely, then a function f is strict if $f \perp = \perp$; otherwise it is *nonstrict*. A strict programming language is one in which only strict functions may be defined by the user. Therefore, OCaml is a strict programming language, whereas Haskell is nonstrict because of its laziness. (As an example of the latter, define `first a b = a`. Then `first 1 1/0` evaluates to `1` under normal-order evaluation, while under applicative-order evaluation, we are forced to "evaluate" `1/0`, resulting in $\perp$.)

4 Module System

A *module* usually packages together related definitions (such as the definitions of a datatype and associated operations over that type) and places them in a separate namespace. Haskell and OCaml both have a module system, but OCaml's is more sophisticated. We will use OCaml's terminology and refer to the body (i.e. implementation) of a module as a *structure*. A module can specify which of its parts (declared names) should be exported — that is, made available to other modules. This allows the construction of *abstract datatypes* (ADTs). A datatype is made *abstract* by witholding its actual representation as a concrete type, a concept familiar from the OOP world. Modules can recursively define and import other modules, and in OCaml, they can be defined locally to an expression. However, they cannot be considered "first class" in either language.

In contrast to Haskell, OCaml supports *functors* (also known as *parameterized modules*), which are "functions" from structures to structures. In essence, functors implement parametric polymorphism for modules, which we clarify by the following example. OCaml's standard library contains the `Set` module, which can be used to store a collection of values as a set; one can then perform the usual operations such as insertion, union, intersection, etcetera. To make these operations efficient, a set is internally stored as a balanced binary tree. However, this means we cannot store values of just any type: we need at least an *ordering* function for the elements of the type.

To realize this, `Set` is provided as a functor which accepts a so-called `OrderedType` and produces a specialized kind of `Set` module for this ordered type. A module is considered an `OrderedType` when it (at least) defines a certain element type and a comparison function for this type. So, assuming the function `str_compare` compares strings, this is how we could create a `StringSet` module in OCaml:

```
module StringSet =
  Set.Make(struct
           type t = string
           let compare = str_compare
         end)
```

This code fragment uses the *let* keyword, which we have not discussed yet: `let` *name* = *expr* is

a (global) declaration, defining the binding between the name *name* and the value of the expression *expr*, which will be known to all subsequent expressions.

5 Imperative Programming

OCaml's facilities for imperative programming can be divided into three groups:

(1) input/output operations,
(2) control structures, and
(3) modifiable data structures.

There is not much to say about the first two of these, apart from *exceptions* (a kind of control structure), which we will treat in §5.2. OCaml offers so-called input and output *channels* for reading and writing, and control structures (if--then-else, while, etcetera) similar to those found in most imperative languages.

As mentioned, Haskell does not have any real imperative constructs; it uses *monads* instead, which are discussed in §5.2. First we will look at OCaml's modifiable data structures.

5.1 Modifiable Data Structures

OCaml offers a number of mutable data structures: vectors, character strings, records with mutable fields, and references. For example, we could define a record type for points in the plane, and a function for moving a point by *modifying* its components:

```
type point =
  { mutable xc : float ;
    mutable yc : float } ;;

let move_to p dx dy =
  p.xc <- p.xc +. dx ;
  p.yc <- p.yc +. dy
```

To create a sequence of statements, we separate them by a single semicolon (a double semicolon is used for "top-level" statements). The <- operator allows us to access or modify a particular element. Also, note that we use +. instead of +: this is a consequence of OCaml's lack of ad hoc polymorphism. There is a function + for adding integers, while +. adds floats.

As was briefly touched on earlier, OCaml provides a polymorphic type ref which can be seen as the type of a pointer to any value. Here is an example of its usage:

```
let x = ref 3 ;;
x := 4 ;;
print_string (string_of_int !x) ;;
```

We construct a reference to a value using the *function* ref. The referenced value can be reached using !, the (prefix) dereference operator. The function modifying the content of a reference is the infix function :=. Actually, references are not primitive in OCaml. They could be defined as follows:

```
type 'a my_ref =
  { mutable content : 'a } ;;
let my_ref x = { content = x } ;;
let deref r = r.content ;;
let assign r x =
  r.content <- x ;
  x
```

(A minor syntactic difference between Haskell and OCaml is that OCaml prefixes type variables by an apostrophe.) Now we can rewrite our example:

```
let x = my_ref 3 ;;
assign x 4 ;;
print_string (string_of_int (deref x))
```

Although useful, the introduction of mutable fields leads to complications in OCaml's type system. Suppose we were to declare let x = ref [] ;;. Then the variable x would have type 'a list ref, which is OCaml's way of writing "a reference to a list of values of type a". But now we can modify x in a way which would be inconsistent with the strong static typing of OCaml:

```
x := 1 :: !x ;;
x := true :: !x ;;
```

(The :: operator prepends an item to a list.) Thus the same variable x would have type int list at one moment and bool list at the next (compare our JavaScript example at the very beginning of this paper). To remedy this, OCaml adds a new category of type variables: *weak* type variables. After declaring let x = ref [] ;;, x gets the type '_a list ref. The type variable '_a is not a type parameter, but an unknown type awaiting instantiation; the first use of x after its declaration fixes the value that '_a will take in all types that depend on it, permanently.

5.2 Monads and Exceptions

In Haskell, monads are used (among other things) to implement the kind of imperative constructs discussed above. A monad is a rather abstract concept, and we will not go into detail about it. Besides playing a central role in the I/O system, monads can be used to create imperative-style computational structures which remain safely isolated from the main body of the functional program. This provides a way to incorporate *side effects* and *state* into a purely functional language like Haskell.

Since exceptions are "impure", in Haskell they are incorporated using monads. OCaml does have a separate exception mechanism, which we describe in this section.

All exceptions belong to a predefined type `exn`. Users can define their own exceptions by adding new constructors to this type. Here are two examples:

```
exception MyException ;;
exception Depth of int ;;
```

We can then construct exception values using, for instance, `MyException` or `Depth(4)`. Exceptions are ordinary values, but they must be *monomorphic*; a polymorphic exception such as

```
exception Wrong of 'a ;;
```

would namely permit the definition of functions with an arbitrary return type.

Actually, the `exn` datatype is (necessarily) special in that it is an *extensible* sum type, which basically means that the set of values of the type can be extended by declaring new data constructors. This feature of OCaml's type system, called *polymorphic variants*, is not present in Haskell.

The following code fragment briefly illustrates how exceptions can be used in practice:

```
let result =
  try
    string_of_int (1/0)
  with
      Division_by_zero -> "NaN"
in
  print_string result ;;
```

The `try ... with ...` construct allows us to do pattern matching over any exceptions that might have been raised. In this case, we are checking only for OCaml's predefined `Division_by_zero` exception.

6 Object-Oriented Programming

Unlike Haskell, a significant part of OCaml is dedicated specifically to providing support for the OOP paradigm. Much of this support is similar to that offered by languages such as C++ and Java, and we review it only briefly.

One can define classes and objects, and use (multiple) inheritance. We should mention that as in most OO languages, OCaml's subclasses give rise to subtypes, which in turn gives rise to *subtyping* polymorphism (also known as *inclusion* polymorphism). Basically, this means that a function could work on an object of a certain type T, but also on objects belonging to subtypes of T.

Methods can be public, private, and virtual, and one can define initializers. So-called *friend* methods can also be defined. Finally, one of OCaml's more interesting OOP features is its support for parameterized classes.

Does Haskell 98, with both its parametric and ad-hoc polymorphism, need any extensions to support conventional object-oriented programming (with encapsulation, mutable state, inheritance, and so on)? This question is a difficult one, and we will not attempt to answer it in this paper (cf. [3]).

7 Conclusion and Extensions

In this paper, we have only looked at "stable", standardized versions of both Haskell and OCaml. All kinds of extensions and variants of Haskell are currently being developed, which may eventually find their way into a new version of the language; the same goes for OCaml. We saw that in a number of areas, Haskell and OCaml hold (small) advantages over each other, by which some of these extensions are inspired. Specifically, Haskell could be equipped with a more powerful module system, such as proposed by Shields and Jones [4]. OCaml, in turn, might benefit from the addition of ad-hoc polymorphism: Furuse's *G'Caml* [5] extends the OCaml compiler to accomplish this.

Two other Haskell extensions are worth mentioning. *Generic Haskell* [6] is a superset of Haskell 98 designed specifically for *generic programming*. This is a subject in itself; suffice it to say that even though Haskell's type classes already support a generic kind of programming to some extent, they do not sup-

port generic programming. Then there is *Template Haskell* [7], offering support for type-safe compile-time meta-programming [8]. A similar effort for OCaml is being carried out in the form of *MetaOCaml* [9].

References

1. Peyton-Jones, S.L., Hughes, J., eds.: Haskell 98: A Non-strict, Purely Functional Language. http://www.haskell.org/onlinereport/ (1999)

2. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system, release 3.08.
http://caml.inria.fr/pub/docs/manual-ocaml-308/ (2004)

3. Kiselyov, O., Lämmel, R.: Haskell's overlooked object system.
http://homepages.cwi.nl/~ralf/OOHaskell/ (2005)

4. Shields, M.B., Peyton Jones, S.: First-class modules for Haskell. In: Ninth International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon. (2002) 28–40

5. Furuse, J.: Generic polymorphism in ML. In: Journées Francophones des Langages Applicatifs (JFLA 2001), Pontarlier, France. (2001) 75–96

6. Hinze, R., Jeuring, J.: Generic Haskell: Practice and theory. In: Summer School on Generic Programming. (2002)

7. Lynagh, I.: Template Haskell: A report from the field. http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/Template_Haskell-A_Report_From_The_Field.ps (2003)

8. Taha, W.: Multi-stage programming: Its theory and applications. Technical Report CSE-99-TH-002 (1999)

9. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++ (2004)

10. P. Hudak, J. Peterson, J. Fasel: A gentle introduction to Haskell 98.
http://www.haskell.org/tutorial/ (1999)

11. Bird, R.: Introduction to Functional Programming using Haskell. 2nd edn. Series in Computer Science. Prentice Hall (1998)

12. Rémy, D.: Using, Understanding, and Unraveling the OCaml Language. In Barthe, G., ed.: Applied Semantics. Advanced Lectures. LNCS 2395. Springer Verlag (2002) 413–537

13. Chailloux, E., P. Manoury, B. Pagano: Developing applications with Objective Caml.
http://caml.inria.fr/pub/docs/oreilly-book/html/index.html (2000)

14. Leucker, M., Noll, T., Stevens, P., Weber, M.: Functional programming languages for verification tools: Experiences with ML and Haskell (2001)

Software Architecture Document Management System

Anton Rademaker[1] and Marten Veldthuis[2]

[1] University of Groningen
anton@antonrademaker.com,
WWW home page: http://www.antonrademaker.com
[2] University of Groningen, marten@veldthuis.com,
WWW home page: http://www.standardbehaviour.com/

Abstract. Today there are many tools involved in the creation of a software architecture document. These tools all maintain different files formats and do not communicate which each other about the semantics of the data itself. Because of this, a lot of information is lost into the document, like design decisions. Later in the software project this causes problems, for example in the area of change management. In this paper, we give an introduction about what a Software Architecture Document Management System (SADMS) is in our opinion and how it helps improving the software engineering process and prevents the erosion of architectural knowledge. A SADMS will include all tools involved in the process of writing a software architecture document. Further, we will investigate possible features for SADMS and their benefits to the software engineering process.

1 Introduction

One of the de-facto standards for software engineering is the Rational Unified Process. In the Rational Unified Process (RUP) the software engineering process is split into three phases: the inception, the elaboration and the construction phase. Most of the work on the architecture is done in the inception and elaboration phases. To handle the problem we will focus on these two phases.

In the inception phase several important things are done in regards to the architecture: stakeholders are selected, an initial use case model is build, risks are assessed and a list of requirements is created. During the elaboration phase, some tasks like the use case model are finished and others like a list of non-functional requirements and a first version of the architecture is build.

Having a SADMS with a central information repository would not suffer from these kinds of problems. Such a system would be purposed in the iterative creation and maintenance of software architecture documents, aiding the users of the system in team collaboration and maintaining coherence throughout the documents and over the course of the project.

The system would be able to do so by allowing users to build and update the document in a versioned web-like structure, while being able to export the document to different standardized views. For example there would be views for architects, managers and for testers. Having multiple views on the architecture makes the data more accessible by only displaying the relevant information for one software engineering discipline. By using one data store for the architecture, it becomes easier to verify the architecture in an automated way. Automated checking of requirements, design decisions and other parts of the architecture becomes possible and reduces the chance of errors in

the architecture. Additionally, it helps make relevant processes less complicated: writing tests for example is much easier when all data necessary for testing can displayed together in a specialized view.

To constrain the part of the problem we will be focusing upon, we will not go into detail for each artifact and relation. We will only provide a list of example objects and relations which in no way we claim to be exhaustive or complete. We'll try to keep the list focussed upon what we believe to be the most important ones though.

In this paper we start with analyzing the elements of the architecture. Using this knowledge we subtract a number of important requirements. At the end of the paper we will state the benefits for the architecture team and the whole software engineering process of our approach.

1.1 Problems

Software Architecture can suffer from what we'll call erosion of architectural knowledge. That means, during the software architecture process, knowledge is lost. Other problems while working on the software architecture of a system is that precious time is lost on things like maintaining coherence.

We will be proposing a solution to these problems, and we'll present these by outlining different aspects of the solution; the general idea (section 3) and the major requirements (section 4). Ofcourse, we'll also be talking about why we think our proposal would solve the problem, and what benefits it would have for the architects. This we'll do in section 5.

2 Background and related work

In the software engineering process it's normal to use different tools like the Rational tools [Cor]) to create a software architecture document ([CBB$^+$02] and [OPF]). Software architecture documents are often a combination of text and UML diagrams (the "4+1" view model [Kru95]). But using different tools and different file formats gives trouble when changes have to be made to the document: changing it at one place is not enough. This has led to new tools like Architecture Rationale and Element Linkage ([TJHN]). Also a lot of architectural knowledge is lost during the process [TA].

Using different views at the same data (see [Cle05] and [OPF]) it's possible to serve the user the information he needs.

3 Analysis

Software architecture documents are often very large documents and grow by the year, because the complexity of software increases every year. To support these large documents with SADMS we need a scalable data model and a good graphical representation. The data model consists of two main parts: *entities* and their *relations* which can be divided over the 4+1 view model: logical view,
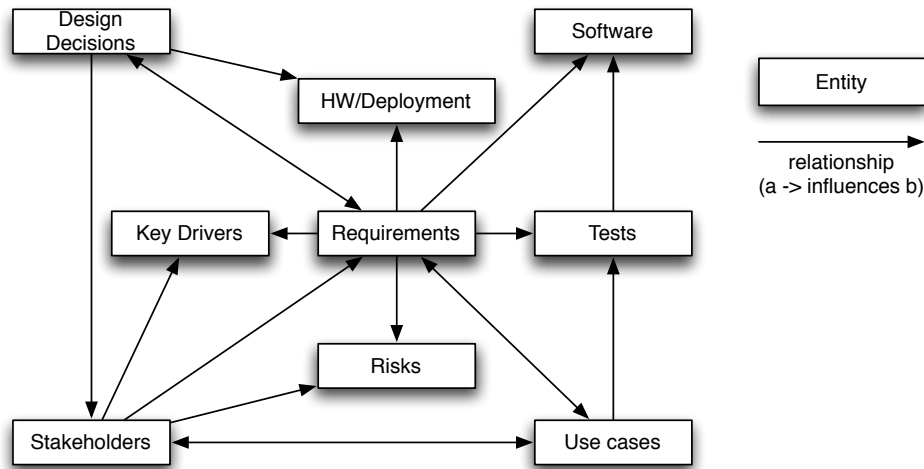
Fig. 1: Different entities in the SADMS and their relations

process view, development view, physical view and scenarios. Both parts have properties like name, management information and possible implementation constraints. In our view, the entities are for example the software's classes, tests, requirements, design decisions and source code. An important property in product families would be variability. Additionally the architectural background has to be retained: rationale, analysis results, and assumptions.

The power of SADMS can only be used if the display of all relations and entities is very powerful. This means we have to think about how everything is displayed: how are entities displayed and how do we present the relations? If the schema becomes too large, a person who is new to the architecture will have problems to understand it. So we have to provide a way to present the reader a view at his own knowledge level of the architecture.

On order to be able to support any current or future entities, we try to keep the amount of different entity types to a bare minimum, and allow the user to customize the model to his/her needs. On the basic level, all entities can be represented as a stream of characters. Classification of different types of streams could be done using a taxonomy; i.e. classification based on completely user-chosen tags.

We can split the entities in a number of groups, as shown in figure 1. In this figure, groups are shown in rectangles, and relations using arrows. We will focus upon some of the more interesting relations.

Requirements and Key Drivers By mapping requirements to key drivers, we are able to see clearly how well each key driver is covered in terms of requirements. In the verification phase, having maintained these mappings allows stakeholders to verify them.

Requirements and Software Obviously, the requirements have a huge impact on the software design. Being able to track dependencies back and forth between requirements and software will make sure change management becomes a much more enjoyable task.

Design Decisions and Requirements All design decisions should respect the requirements. When the latter are slated for change, conflicts may occur and assumptions made previously may no longer hold or will need to be revised.

Tests and Use Cases Use cases have a strong correlation to functional tests. After all, functional tests verify the workings of the system by checking the output for a given simulated user input. Use cases serve a similar purpose, but on another level. By linking these it's easy to see if the functional test is still in correspondence with the use case.

Descriptions of the entities are about the responsibilities and provided interfaces. The relations between the groups and within groups can divided in (also visualized in figure 2):

Depends on Many links exist between requirements, design decisions, tests and actual pieces of implementation. By defining these relations, change management becomes much easier.

Data flow Data flows from different objects within the logical view

Control flow A better understanding of control flow helps programmers track down the source of a bug.

Patterns By saving the related objects of the pattern, validation is easier and the architecture is better understandable.

Required interfaces Normally you can check these relation only at compile time in an automated fashion. An SADMS would enable the architecture team to perform this validation while constructing the document.
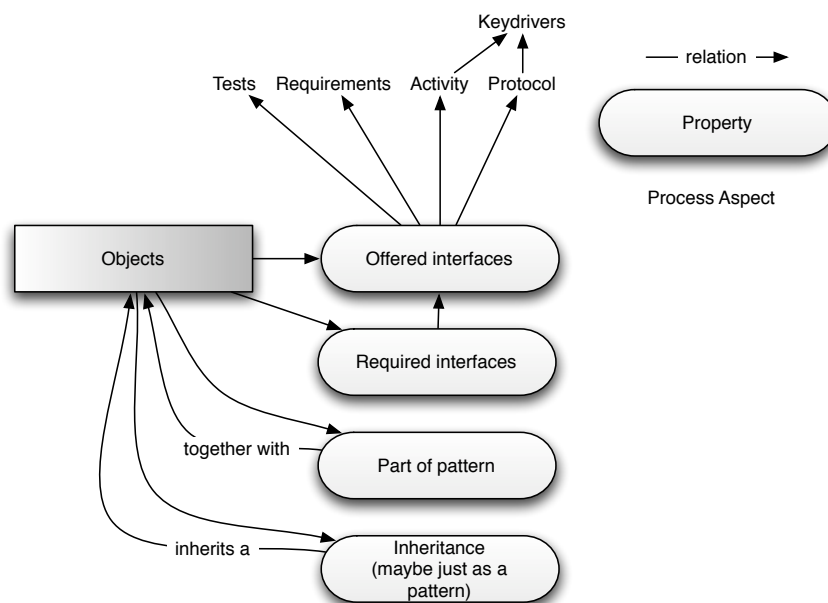

Fig. 2: Possible relations between entities

From the analysis we can now subtract a number of requirements that would be placed on a SADMS.

4   Important requirements

We identified a couple of the most important requirements which we'll divide them in general, display related and management related requirements. First of all, there are some generic requirements.

Rationale For example each entity and relation should have the possibility to add a rationale.

Tests and source code Source code should be linked to objects in the architecture, which can be done by adding a property to entities.

Language plug-ins To support source code linking it is important that SADMS supports a large number of programming languages (e.g. using a plug-in system).

The graphical interface and display of the document itself has requirements too:

View support The graphical interface must support different views at the data model and must contain the tools to edit and view the data model in a easy and understandable way. Typical views would be project manager view, developer view, tester and integrator view, maintainer view, designer of other systems view, product line application builder view, customer view, end-user view, analyst view, infrastructure support person. More information about these views can be found in [CBB+02].

UML Because architectures can be quite large with many entities and relations it's very important that the used figures in the document are easy to understand and conform to a standard (e.g. the UML standard notation).

Also there are a some management related requirements. These are important to have in order to improve the process in areas not directly related to the architecture itself.

Version control By saving what is changed and also by who the history of the document is easily tracked. It helps architects in case an error was introduced in the document to track down when it was introduced and what is maybe depending on it.

Time tracking Storing the amount of time every object and entity has cost a architect improves the project management process.

5   Process benefits

Now we have the relations between the several entities we can focus on the benefits of a SADMS, and a set of requirements we feel necessary for a successful SADMS. The benefits range from software architecting to software engineering.

Architectural knowledge and validation For the software architects the benefits are mostly related to architectural knowledge (stopping the erosion of architectural knowledge) and easier validation.

Rationale SADMS makes it easy to add rationale to each entity in the architecture and view it later.

Stating (design) decisions Architects are encouraged to add information about their decisions, so later in the project this information is not lost but can be found at the point where the decision was made and why.

Patterns Architectural patterns could be spotted more easily, and could be stated explicitly and later added to a global library.

Validating By saving standard information and benefits about patterns in SADMS helps in validating the architecture (automatically).

Change impact All relations are stated explicitly so it is easier to see what the impact at the architecture would be in case a change to an entity is made.

Project status and testing The management of the whole process becomes easier because the status of every artifact can checked in detail using for example the tests linked to the source code.

Requirements validating The source code links enables SADMS to verify automatically the status of requirements and thus the status of product developed.

Getting into the project The time that is needed to get into the project is reduced by using the multiple views and layers and reading the rationales. Using these tools it is easier to understand how things work and are done. In case a project is taking too long it is faster reliable to add extra workers to the team because they don't have to spend much time understanding the architecture.

Version management Using the version tracker it is easier to see who was responsible for something and if in case that a mistake was made the document can be reverted back. This also helps to prevent the lost information due to updates in de architecture.

Training Using the statistics that the system can generate, it is easier to see who needs which training, for example by determining the error rate of the architects.

6 Future work

A lot of work still remains to be done before we will see a working SADMS. This paper only describes which relations there are and which benefits derives from stating them explicitly. The next big thing would be to compile a complete list of requirements. For example, this list should contains a definition of which views are appreciated. Also work has to be done about how entities and relations are displayed.

After all the preparations are done the actual data model and visual libraries can be created. With these two components the main SADMS editor can be build. After that all kinds of analytic tools and plugins can be added to SADMS.

7 Summary

We can clearly see the benefits of having a single system in which all architectural knowledge is maintained. Immediately after putting such a tool to use, benefits would be most clearly visible in the communication of changes in the architecture. Because all relations are explicitly mapped, it's easy to assess the impact of any change and large time-savings could be achieved in that area.

In the medium term, we'll see that people join and leave projects and will be able to do so easier than before, since no knowledge is kept private.

Long term benefits would come from a different angle, the versioned nature of the system would enable heuristics and statistical analysis. It would become easier to see patterns emerging, and it might be possible to integrate a pattern repository into the system (perhaps even finally bringing an end to the scattered distribution of knowledge about patterns).

References

[CBB+02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Number ISBN 0201703726. Addison Wesley Professional, September 2002.

[Cle05] Paul Clements. Comparing the sei's views and beyond approach for documenting software architectures with ansi-ieee 1471-2000. July 2005.

[Cor] Rational Software Corporation. Rational unified process - best practices for software development teams.

[Kru95] Philippe Kruchten. Architectural blueprints - the 4+1 view model of software architecture. November 1995.

[OPF] Open process framework (opf) repository organization (opfro).

[TA] Jeff Tyree and Art Akerman. Architecture decisions: Demystifying architecture.

[TJHN] Antony Tang, Yan Jin, Jun Han, and Ann Nicholson. Predicting change impact in architecture design with bayesian belief networks.