Improving Quality Attributes of a Complex System Through Architectural Analysis – A Case Study

Rikard Land
Mälardalen University
Department of Computer Engineering
Box 883
721 23 Västerås
+46 (0)21 10 70 35
rikard.land@mdh.se

ABSTRACT

The Software Architecture Analysis Method (SAAM) is a method for analyzing architectural designs, providing support in the design process by comparing different architectures and drawing attention to how a system's quality attributes are affected by its architecture. We used SAAM to analyze the architecture of a nuclear simulation system, and found the method to be of great help when selecting the architecture alternative to use, and to draw attention to the importance of software architecture in large.

It has been recognized that the quality properties of a system is to a large extent determined by its architecture; there are, however, other important issues to consider that belong to "lower" design levels. We describe how detailed technical knowledge affected the design of the architecture, and show how the development process in large, and the end product can benefit from taking these issues into consideration already during the architectural design phase.

Keywords

Software architecture, architectural analysis, SAAM.

1. INTRODUCTION

A nuclear power plant must be safe for humans and the environment; it must, moreover, be economical. To optimize plant maintenance in these respects, a number of computer simulations are performed. Governmental regulations state how and when safety analyses are to be carried out, to ensure that the plant is safe [13].

In the nuclear business domain, there are already a number of simulation programs [11], many with a development history of decades, already validated and approved by the authorities. Typically, the simulation programs are installed on powerful Unix servers. The input data to an execution is edited and stored in input files. The simulation program is started via a command line with arguments specifying e.g. the input files to use and simulation time. There may be some means of monitoring the progress of the simulation, and when it is finished, the output is available in output files. It is common that several input files are required, and the simulation produces several output files representing different kind of output data. Both input and output files may be either binary or text files.

However, this type of system is somewhat out of date. Files are stored in a central directory tree structure where users must know the naming conventions and the directory structure must be maintained. The users have to perform many tasks manually that could beneficially be done automatically. Since there are files, paper documents, and databases in different formats, one has to rely on methodologies to ensure that input data is consistent. The problem is made worse by the vast increase in size of data, both input and output, over the years. The user interface is inhomogeneous and hard to master – the users have to collect data from different sources, edit text files describing input, and analyze the results found in the output files. A number of tailor-suited tools, e.g. graphical plot programs, have been written and are used during the analysis of the output, but a more integrated system would improve the efficiency and quality of the work.

To address these problems, Westinghouse Atom developed the PAM system (Plant, Analysis, Methodology). In PAM, data is stored in a relational database, many tasks are done automatically, data consistency is ensured to a much higher degree through program code and the use of one single database, and all of these features are reached from an integrated graphical user interface, the *client*, executing locally. The question how to handle the simulation programs from within PAM was, however, not easily solved. One straightforward solution would be to port the simulation programs to the client's platform, but this is not practically possible for several reasons:

- Since there are a large number of simulation programs, it would require a huge amount of work.
- Many of the programs are commercial products.

- The programs would have to be re-verified and re-validated at very high costs.

The design of the simulation part therefore had to deal with existing programs, compiled, verified, and validated for a specific platform. We concluded that this requirement must be built into the highest design level, i.e. on the architectural level. Our goal was to find the best solution using different variants of architectural solutions. With "best", we intended the best tradeoff between certain quality properties; we wanted our system to be robust, maintainable, have acceptable performance and be as cheap as possible. We will see how these properties were included in the analysis and how they are affected by the architecture.

The remainder of the paper is organized as follows: the development of different architectures is described in section 2, the evaluation of the four alternatives in section 3, and section 4 contains other related observations.

2. THE ARCHITECTURAL DESCRIPTION

At first, we designed one architecture. We soon found it useful to split it into four variants and compare these with each other. To evaluate the architectural proposals created during the investigation we used the Software Architectural Analysis Method (SAAM) [1,7], a general method for evaluating quality attributes. After the evaluation, it was found that there were a few issues that needed more scrutiny. This refinement procedure was done in much the same way as with the methodology Bosch suggests [2].

With this case study we have followed the pragmatic approach that has characterized the field of software architecture so far; much of the architectural research has included case studies [1,2,3,4,5,6,9,12]. Bass et al describe case studies where SAAM is used [1]. The relation between architecture and quality attributes is emphasized by Bass et al [1] and Bosch [2]. The Architecture Tradeoff Analysis Method (ATAM) is a relative of SAAM, which refines the analysis by making the tradeoff choices even more explicit [8,9].

SAAM is applied early in the development cycle, and gives the architect the possibility to choose an architecture with an acceptable tradeoff between quality attributes. With this method, architectures are informally compared through the use of *scenarios*. In our case we had use cases like "the user starts a simulation" and change scenarios such as "PAM is extended with functionality to compare binary output files". For the outcome of the analysis to be reliable it is crucial that the selected scenarios are indeed representative for actual future scenarios. How could we be sure that we used enough scenarios – or the "right" ones? Every type of stakeholder of the system (users, developers, managers) had representatives participating during several discussion meetings in the development of the scenarios. Everybody was instructed about SAAM and scenarios in advance. Thus, the chance of any major scenario being missed was decreased. A dozen is a fair number of scenarios to use [1]; we gathered 19.

2.1 The basic features of the system

The first design decisions were quite straightforward: there is a client running in the PC environment, and a central database. To handle requests of executions from the clients, it was decided that some sort of PAM-specific software was needed on the server computers [10].

The PAM system thus contains three types of nodes with different tasks: the local PCs where the users work, a database server, and the Unix servers were the actual simulations are executed. In the following, when referring to the "servers" we intend the Unix servers. The database is in most cases discarded from the discussion for simplicity.

The users collect and edit input data in the client; the data is then stored in a central database. The functionality we focus on in this paper concerns what happens when this data is to be used in a simulation. Data from the database is supposed to be formatted and written to the input files, and the simulation program should be started. During and after execution, the client shall be able to present the output files to the user. In some cases the data should be filtered, such as when only one variable among many in the same file is plotted.

To handle the simulation programs, we designed a basic architecture; all four alternatives share the basic features described in this section. In the following, we will use the word "process" with the meaning "separate thread of execution with a specific task"; whether we should implement the components as operating system processes or threads will be discussed in section 4.1.

On each calculation server there is a very central process running, the "Service Broker" (SB). It simply provides the service of starting calculations to the clients. The idea with this process is to make the system robust: since it implements such a simple task, it should be possible to make it robust enough to always be running.

On request from a client, the SB starts a "Calculation Server" (CS), which maintains one simulation. It is a separate process without any direct connection to either the SB or the starting client. Any client can monitor and control the progress of the simulation through the means of sending messages to the CS.

One PAM-simulation consists of a user-written script, with loops etc., starting a number of *tasks*, which are the actual simulation program executions. The CS spawns one "Task Calculation Server" (TCS) process per task, not necessarily on the same node.

Figure 1 shows a snapshot of some of the processes in the system, describing how the processes interact when a simulation is started in the system. (The database is omitted from this and the following figures. It resides on a separate node, and all processes connect to it during startup and remain connected during their whole lifetime.) The client process requests an execution from the SB process on one server, which starts a CS process (the directed lines); after this, there are no dependencies between these processes. The CS starts three TCS processes, each responsible for the execution of one task; the CS and TCS processes are dependent on each other during the whole simulation (the lines without arrowheads). In this particular case two tasks need to be run on the same node as the CS, and one on another (this could be due to where particular simulation programs are installed or to utilize the system's resources better).

The key features are that there is always exactly one SB per server computer, exactly one CS per executing simulation, exactly one TCS per executing task, and any number of clients on each PC.

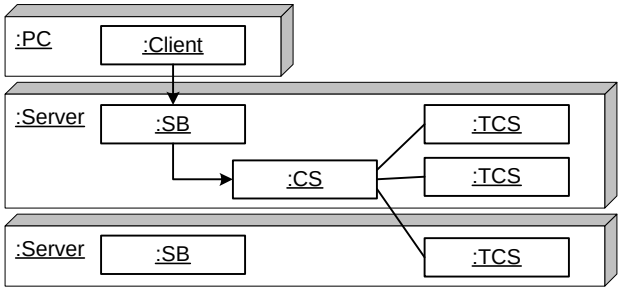

Figure 1. Process interaction when a simulation is started.

2.2 The four variants of the basic architecture

During simulation, the input and output files reside in a working directory, typically with as high performance as possible. When the simulation eventually is being "approved", data is to be filtered (to decrease size) and moved into the database for long-term storage. However, in the meantime, we would like the files to be stored on an intermediate storage area, typically an ordinary disk with backup mechanisms. It can be discussed on which node the files should be stored during this period of time. We can discern two strategies: either the files are stored on the node where the simulation took place (the "distributed" approach which we will call "1"), or on one node acting as "file server" for PAM (the "centralized" approach, "2"). The former approach would probably give higher performance on the expense of system complexity, while the latter would be easier to understand but includes more overhead.

The other issue concerns the presentation of these files in the client. The files are processed and filtered before they are presented to the user, and the question is where this filtering should take place – in the client or on the server (which implies an extra component on the server). Intuitively, if the files are filtered on the server, performance would be improved because a smaller amount of data is sent across the network, but the system would be more complex and the server more loaded. We name the strategies of processing files in the client or on the server "A" and "B", respectively. Figure 2 shows the different strategies.

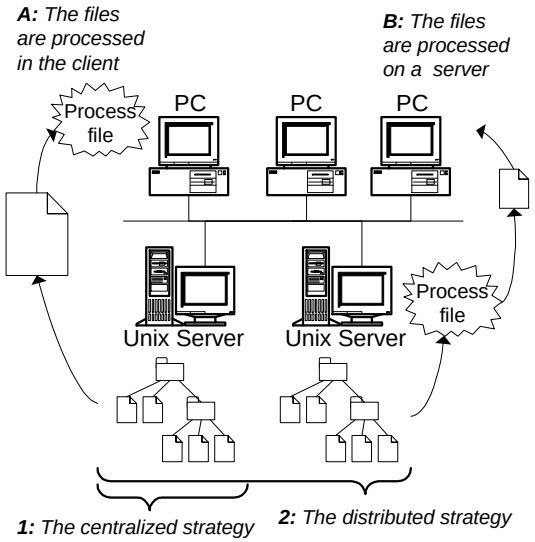

Figure 2. The different approaches for file handling.

What makes these issues important is that the size of the files described above can be very large. 10 MB for one simulation is not uncommon. Each of these two problem dimensions (where to store files and where to process files) has two solutions. All solutions seemed to have advantages and disadvantages, and it was by no means obvious which solution, and combination of solutions, would include the "best" strategy. It is an axiom in software architecture that after quality attributes have been assessed, a tradeoff decision is required [1,2,6]. Thus it was decided that all four combinations should be treated as separate architectures and compared using SAAM. Figure 3 describes how the architectures fit into the two problem dimensions and how they accordingly are named – A1, A2, B1, and B2.

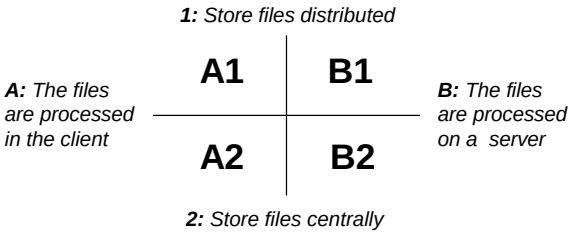

Figure 3. The processes in a small PAM system according to design B1 and B2.

Figure 4 shows a snapshot of the processes in a small system according to alternatives A1 and A2 (the difference between them is not discernible in this view). There are no simulations executing, and the SB is idle. Files are handled in the client components, so there are no extra components. There are an arbitrary number of clients on any number of PCs, but only one SB per server computer.

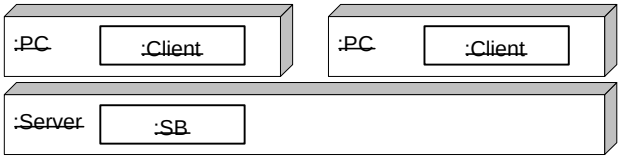

Figure 4. The processes in a small PAM system according to design A1 and A2.

In architectures B1 and B2, an extra process was introduced, called "Service Dispatch Server" (SDS). The task of the SDS is to process the input and output files associated with the simulations on the server before transferring them to the client. Figure 5 shows the processes graphically, according to alternatives B1 and B2 (there is no difference between them in this view); the system is of the same size and state as in Figure 4. There are an arbitrary number of clients, one SB per server computer, and one SDS per client.

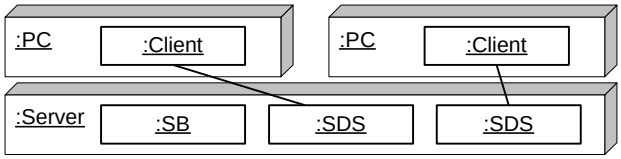

Figure 5. The processes in a small PAM system according to design B1 and B2.

We can clearly see that there are more components and dependencies in B1 and B2 than in architectures A1 and A2. The question to be analyzed is whether the expense in complexity pays off with other advantages, such as increased performance.

3. THE ANALYSIS OF THE ARCHITECTURES

We will now describe how performance, system load, and maintainability were estimated from the architectural description.

3.1 Performance analysis

As is described above, large files are sometimes transferred over the network, affecting performance negatively. In the performance analysis, the number of large data transfers over the network were measured or estimated. To be able to do this, we used five user scenarios including network transfers of large pieces of data, such as "a simulation is executed" and "two files are compared".

The number of actual transfers during each scenario was estimated, and the result of this analysis can be seen in Figure 6. As an example, the figure describes that for scenario 1 ("a text file from a server computer is viewed in a client") architectures A1 and A2 include one transfer, not necessarily of the whole file (depending on the circumstances), for alternative B1 always exactly one whole file, and for B2 between one and two whole transfers of a file.

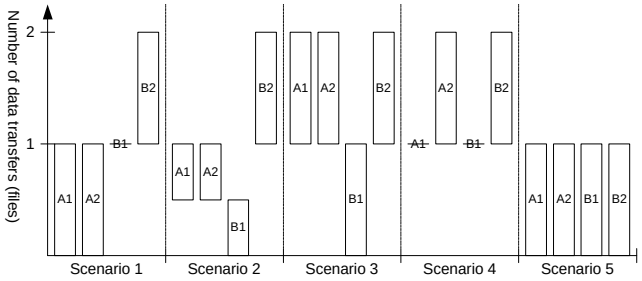

Figure 6. The number of large data transfers across the network in five different scenarios.

Architecture B2 clearly performed worst in all scenarios, A1 and A2 was equal in all but one scenario, and it can be argued which of A1 and B1 performed absolutely best. A more detailed analysis would include determining an average size of the data and weighting the scenarios. For our purpose, however, this analysis was considered enough – we found that one architecture (B2) was worst, and the others comparable when considering network load due to large file transfers.

3.2 System load analysis

In the system load analysis, the number of processes in a running system was calculated. This was thus not a SAAM analysis, but rather a simple addition of processes, based on the number of server computers and an estimated average number of clients and simulations ("small", "medium", or "large" systems). As is shown in Figure 7, the number of processes is consistently lowest for architectures A1 and A2, while the number of processes may be almost doubled in architecture B1.

Before performing this analysis we had no clear notion of what the outcome would be, but when looking at these results in retrospect, we found them to be very intuitive. The extra processes in systems according to B1 and B2 are due to the inclusion of the SDS component. The great difference between B1 and B2 are due to the strategy on which servers there must be SDSs, which in its turn depends on whether the simulation files are stored using a central or a distributed approach.

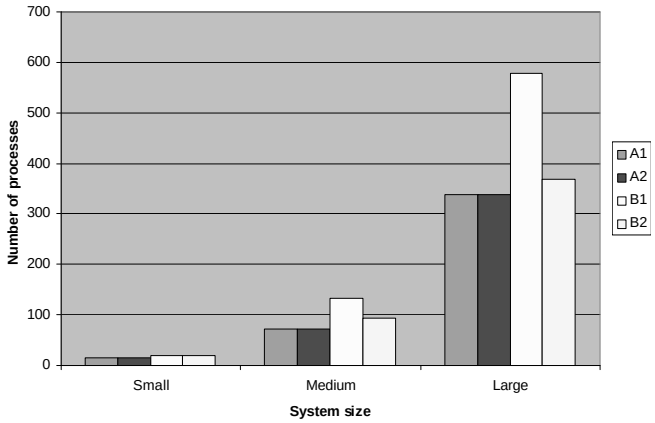

Figure 7. The number of processes in running systems of different sizes.

3.3 Maintainability analysis

The stakeholders formulated 16 change scenarios, containing the addition of functionality. An example of a change scenario is to "include functionality to compare two binary output data files". The results from the execution of these scenarios are presented in Table 1 – it turned out that architectures A1 and A2 were undistinguishable, as was B1 and B2. Architectures A1 and A2 in general score better than B1 and B2, which of course is because architectures B1 and B2 include more components (the SDS). However, two scenarios affect the SB in architectures A1 and A2, which is undesirable because of its central position and the robustness requirements.

Table 1. Statistics from the scenario executions.

	A1/A2	B1/B2
Number of components affected: total (average)	23 (1.4)	28 (1.8)
Number of scenarios affecting at least 2 components	6	10
Number of scenarios affecting the SB	2	0

Scenarios are said to *interact* on a component if both affect it [1]; if several unrelated scenarios affect the same component, this is an indication that the separation of concerns between components may be insufficient. The more interactions, the more complex it is to maintain the system. However, as always there are no absolute numbers on how many interactions are considered "too many"; these numbers should rather be used for comparing architectures (which is done here), or to focus attention on particular components with many scenario interactions. See Table 2.

With only five or six components and 16 scenarios, this analysis gives a rather decent distribution, apart from the client. However, the client is equally unstable in all architectures.

How should we interpret these results? The figures clearly say that architectures A1 and A2 are more maintainable than B1 and B2 and seem to leave no room for alternative interpretations. However, this result is at least partly a consequence of the fact that some functionality is added to an extra component, the SDS. If this means that architectures B1 and B2 are more fine-grained than A1 and A2 it is not fair to compare the architectural descriptions – they describe the system on different levels of abstraction. We considered this objection seriously, and among other things tried to estimate the size of the code in the components. We arrived at the conclusion that the client code would be slightly smaller in architecture B1 and B2 than in A1 and A2, but that the program code of the SDS would be substantially larger than the decrease in client code size. We finally decided that the architectures were indeed comparable, and the figures of the analysis fair. In addition to this, we had gained the insight that B1 and B2 would require more coding, which speaks to their disfavor.

Table 2. Scenario interaction on each component. (The variations depend on how certain scenarios are implemented.)

	A1/A2	B1/B2
Database	Affected by 4 scenarios	
SB	Affected by 2 scenarios	Affected by 0 scenarios
CS	Affected by 3 scenarios	
TCS	Affected by 3 or 4 scenarios	
Client	Affected by 10 or 11 scenarios	Affected by 9 to 11 scenarios
SDS	N/A	Affected by 7 or 8 scenarios

3.4 Other analyses

Besides analyzing performance, system load, and maintainability, we informally evaluated testability, reusability, and portability. However, these analyses did not reveal any differences between the architectures – we therefore omit the details of the analyses.

However, the conclusion that several architectures are indistinguishable is also a valuable result, from which it is possible to draw conclusions. Firstly, since these properties are not affected by the choice of architecture, any of the alternatives can be chosen (as far as these properties are concerned). Secondly, if these properties were considered crucial for the system's success, we might have devised more alternatives, to explore whether these properties could be improved at the architectural level. SAAM can only compare different alternatives, not give absolute measures of the quality properties, so the outcome could mean "they are equally good" as well as "they are equally bad". We did not pursue this track further because we were confident that our understanding of how these properties were affected

ensured that these properties would not pose any major problem. So, thirdly, through the analysis process itself, we had gained insight enough into the problem to make the decision that further analyses were not needed.

At this stage, we summarized the analysis and found performance, system load, and maintainability to be the properties distinguishing the alternatives.

3.5 Discussion

It was not easy to without aid predict which architecture would be the most fit for our requirements. The analyses clearly helped serving as a basis for a choice. We found that architecture B2 was inferior with respect to the number of large data transfers, while B1 was inferior with respect to the number of processes in the system. So far, if performance is important, either architecture A1 or A2 should be chosen; A1 was estimated to have slightly better performance than A2.

When evaluating maintainability, we see that A1 and A2 are superior, the only problem being that two change scenarios affect the SB, whereas in B1 and B2 the SB is unaffected by all scenarios. In other analyses the architectures were found to be equal.

It is quite clear, then, that architecture A1 or A2 should be chosen.

4. GENERAL OBSERVATIONS AND LESSONS LEARNED

4.1 Processes or threads

So far, we have described the runtime components as "processes", but the architectural description does not require the CS and TCS components to be implemented as separate operating system processes. They could very well be implemented as threads executing in a designated "CS and TCS host" process, or why not in the SB. The choice between processes and threads can be considered a lower-level design issue. This does not mean that the choice does not affect the properties of the system, but rather that this tradeoff does not need to be solved on the architectural level. There might indeed be a tradeoff between system load and robustness – processes load the system more, while a failure in one thread is likely to affect other threads. In the actual implementation of PAM, it was decided that the system would be more robust if processes are used in the case of a component failure, and that a threaded solution would be considered if there were any system load problems.

We can draw the general conclusion that *an architectural description does not need to distinguish between processes and threads*, but can simply describe the runtime components as "separate threads of execution". The choice of whether these are implemented as processes, operating system threads, or language-level threads can be postponed to later design stages.

4.2 Detailed knowledge useful

One possible source of instability in the system would be that the system is spammed with CS and TCS components having lost contact with each other. However, instead of being a potential source of instability, the communications channel is used to increase robustness. Sockets proved to fulfill our expectations well. Indeed, the knowledge of the socket mechanism was an important input to the creation of the architecture. Let us view the connection between a CS and one of its TCSs. Both sides will be noticed whenever the socket is unexpectedly closed, and immediately terminate themselves. The socket could be closed due to several reasons – the network might be lost, or the other component can have failed or terminated unexpectedly (due to e.g. a bug). The components show a consistent behavior in all such cases, provided that the sockets mechanism is reliable enough to always notice these cases (which we believe it is). Thus, the architecture builds robustness partly on the sockets mechanism.

Since we wanted to reuse legacy code written in the Tcl programming language [14], we knew in advance that Tcl was a strong candidate of implementation language. Our experience of the socket functionality being very robust and easy to use in Tcl strongly influenced the development of the architecture as described above. We also knew that the use of Tcl would support portability since there are Tcl interpreters available on the platforms of interest; as a consequence we found it superfluous to support portability in our architectural description.

The general conclusion to be drawn is that *detailed technical knowledge is an important input to the architectural design process*.

4.3 Simplicity implies robustness

Our next observation concerns another way the system is made robust. In earlier prototypes of the system, there were problems with robustness. There were many scenarios where a failure in one process made other processes fail too. Attempts were made to handle every possible faulty state, but this proved to rather introduce new errors and make the code incomprehensible. One of the governing ideas behind the new architectures has been to make the runtime components as independent of each other as possible, in the sense that the system as a whole is in a sound state even if many individual components and communication channels fail. It should be noted that this feature is not implemented through any advanced fault-tolerance techniques, but rather by creating a relatively simple architecture. Of course, the robustness, as well as any quality attribute supported by the architectural

description, is ultimately dependent on how well the system is actually implemented.

Our experience supports the idea that *one should build important properties directly into a system's architecture*, rather than try to add them afterwards [1].

4.4 An unexpected solution of a tradeoff

When discussing the outcome of the evaluation, there were a few minor issues that needed more consideration, of which we will describe one. As we saw, the results of the analysis indicate that we should choose between A1 (with a distributed file structure) and A2 (with a central file structure). On the one hand, the project group intuitively felt uncomfortable with the idea of having files distributed over a large number of computers when tracking errors, while on the other hand this implies slightly higher performance. When considering this problem, it seemed as we had to decide on a tradeoff. This proved to be both true and false. We found that the choice of strategy where to store files did not need to be decided upon until installing a PAM system, thus making a system administrator responsible for solving this tradeoff (for it is indeed a tradeoff). We decided that viewed this way, the resulting architecture could be described as a synthesis of A1 and A2, or in other words that there was no difference between A1 and A2.

In the general case, instead of making tradeoff decisions during the design phase, it might be possible to give the system manager the freedom to choose the tradeoff considered optimal in his particular situation. We believe that *one should consider whether a tradeoff can be postponed to the configuration and maintenance phases*. However, we are aware that such an approach may introduce new tradeoffs: a highly configurable system may be harder to understand and maintain, and harder to test, than a less configurable system.

5. CONCLUSION

We devised one architecture, but created four variants of it and compared these at the architectural level to be able to assess the quality attributes of the final system. SAAM provided a useful way of evaluating our four suggestions, revealing drawbacks not obvious at first sight. The analysis provided a basis for taking conscious decisions on which architecture to choose, given an estimate on what quality attributes the four variants would have. The use of SAAM proved to bring more benefits: the stakeholders of the system became more conscious of quality attributes and the architecture's impact on these; moreover, a fruitful interaction between analysis and design took place thanks to SAAM.

Besides supporting the usefulness of SAAM, we were able to draw a number of general conclusions. We learned that the creation of an architecture cannot be performed in an "ideal" world, rather the knowledge about the availability of implementation issues are both necessary and advantageous. In our case, the architecture was colored by the knowledge of specific Tcl and sockets features, and this knowledge was taken advantage of to create a robust architecture. We achieved a certain degree of robustness due to inherent features of the architecture, which is preferable to writing error-handling code. During the design process, we found it useful to discuss the runtime components in terms of "processes", although it was not decided whether these should actually be implemented as processes or threads. We have also described that it was possible and useful to postpone one tradeoff decision to the system configuration and maintenance phases. With further research we hope that these issues will mature from mere observations to more formal models incorporated into the theory and tools of software architecture.

During the analysis, the important question was raised whether the architectural descriptions, containing different numbers of components, actually were comparable. We were able to give what we believe to be a satisfactory answer by estimating the size of each component. With further research it might be possible to more formally decide when architectural descriptions differ too much and when they indeed are comparable – a prerequisite for any analysis.

Finally – what is our study worth for the stakeholders of PAM? Are our estimates of performance, system load and maintainability accurate? Is the system robust and portable enough? We will not be able to answer these questions until PAM has been in production use for some time. We hope that we will then be able to gather measures of the quality attributes of interest and compare it to our analysis. This will provide useful feedback to our research.

REFERENCES

[1] Len Bass, Paul Clements, Rick Kazman, *Software architecture in practice*, ISBN 0201199300, Addison-Wesley, Reading, Massachusetts 1998

[2] Jan Bosch, *Design & Use of Software Architectures*, ISBN 0-201-67494-7, Addison-Wesley, Edinburgh 2000

[3] Ivan T Bowman, Richard C Holt, Neil V Brewster, *Linux as a Case Study: Its Extracted Software Architecture*, Proceedings 21st International Conference on Software Engineering (ICSE), 1999

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern-oriented Software Architecture – A System of Patterns*, ISBN 0 471 95869 7, John Wiley & Sons, Chichester, West Sussex 1996

[5] Henrik Hermansson, Mattias Johansson, Lars Lundberg, *A Distributed Component Architecture for a*

Large Telecommunication Application, Proceedings of the Asia-Pacific Software Engineering Conference (APSEC), Singapore, December 2000, 188-195

[6] Christine Hofmeister, Robert Nord, Dilip Soni, *Applied Software Architecture*, ISBN 0-201-32571-3, Addison-Wesley, Reading Massachusetts 2000

[7] Rick Kazman, Len Bass, Gregory Abowd, Mike Webb, *SAAM: A Method for Analyzing the Properties of Software Architectures*, Proceedings of the 16th International Conference on Software Engineering, 1994

[8] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, Jeromy Carriere, *The Architecture Tradeoff Analysis Method*, Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), (Monterey, CA), August 1998, 68-78

[9] Rick Kazman, Mario Barbacci, Mark Klein, S. Jeromy Carrière, *Experience with Performing Architecture Tradeoff Analysis Method*, Proceedings of the International Conference on Software Engineering, New York, 1999, 54-63.

[10] Rikard Land, *Architectural Solutions in PAM*, M.Sc. Thesis, Department of Computer Engineering, Mälardalen University, 2001

[11] Frank Schliephacke, *Computer Codes Description*, Westinghouse Atom Report BTU 01-049, 2001

[12] Mary Shaw, David Garlan, *Software architecture – Perspectives on an emerging discipline*, ISBN 0-13-182957-2, Prentice Hall, Upper Saddle River, New Jersey 1996

[13] SKIFS 1998:1, *Statens kärnkraftinspektions författningssamling - Swedish Nuclear Power Inspectorate Regulatory Code*, ISSN 1400-1187, 1998

[14] Tcl Developer Site, http://www.scriptics.com