Evaluating Software Quality Attributes of Communication Components in an Automated Guided Vehicle System

Frans Mårtensson, Håkan Grahn, and Michael Mattsson
Department of Systems and Software Engineering
School of Engineering, Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden
{Frans.Martensson, Hakan.Grahn, Michael.Mattsson}@bth.se

Abstract

The architecture of a large complex software system, i.e., the division of the system into components and modules, is crucial since it often affects and limits the quality attributes of the system, e.g., performance and maintainability. In this paper we evaluate three software components for intra- and inter-process communication in a distributed real-time system, i.e., an automated guided vehicle system. We evaluate three quality attributes: performance, maintainability, and portability. The performance and maintainability are evaluated quantitatively using prototype-based evaluation, while the portability is evaluated qualitatively. Our findings indicate that it might be possible to use one third-party component for both intra- and inter-process communication, thus replacing two inhouse developed components.

1. Introduction

The size and complexity of software systems are constantly increasing. It has been identified that the quality properties of software systems, e.g., performance and maintenance, often are constrained by their software architecture [3]. The software architecture is a way to manage the complexity of a software system and describes the different parts of the software system, i.e., the components, their responsibilities, and how they interact with each other. The software architecture is created early in the development of a software system and has to be kept alive throughout the system life cycle. One part of the process of creating a software architecture is the decision of possible use of existing software components in the system.

The system we study in this paper is an Automated Guided Vehicle system (AGV system) [7], which is a complex distributed real-time system. AGV systems are used in industry mainly for supply and materials handling, e.g., moving raw materials, and finished products to and from production machines. Important aspects to handle in such systems are, e.g., the ability to automatically drive a vehicle along a predefined path, keeping track of the vehicles' positions, routing and guiding the vehicles, and collision avoidance. The software in an AGV system has to be adaptable to quite different operating environments, e.g., iron works, pharmacy factories, and amusement parks. More importantly, the system may under no circumstances inflict harm on a person or object. The safety and flexibility requirements together with other quality- and functional requirements of the system make it a complex software system to create and maintain. In the system in our case study, the software in the vehicle can be divided into three main parts that continuously interact in order to control the vehicle. These parts communicate both within processes as well as between processes located on different computers.

In this paper we evaluate two communication components used in an existing AGV system and compare them to an alternative COTS (commercial-off-the-shelf) component for communication [19] that is considered for a new version of the AGV system. We evaluate three quality attributes for each of the components: performance, maintainability, and portability. We use three prototypes built using a prototype framework to measure the performance of each component. Both intra-process as well as inter-processes communication are evaluated. The communicating processes reside both on the same computer and on two different computers connected by a network. We measure the maintainability of the three components using the Maintainability Index metric [17]. We also discuss qualitative data for the portability aspects of the components.

The evaluation is performed in an industrial context in cooperation with Danaher Motion Särö [6]. The usage scenarios and architecture description that are used during the evaluations have been developed in cooperation with them.

Our results indicate that the performance of the COTS component is approximately half the performance of the in-house developed communication components. On the other hand, using a third party COTS component significantly reduce the maintenance effort as well as increase the functionality. Finally, all three components turned out to be portable from Windows XP to Linux with very little effort.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

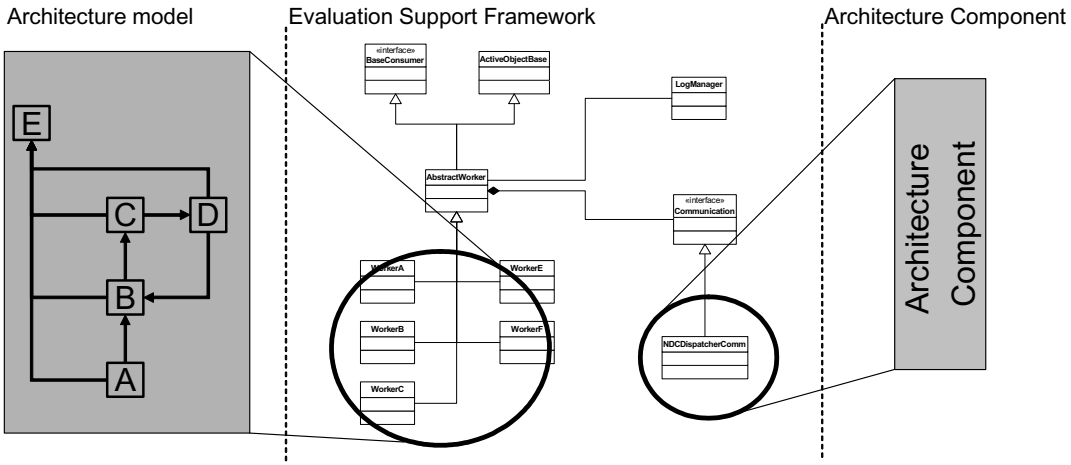

Figure 1. The prototype consists of three main parts: the architecture model, the evaluation support framework, and the architecture components.

The rest of the paper is organized as follows. Section 2 presents some background to software architecture, architecture evaluation, and automated guided vehicle systems. In Section 3 we introduce the components and the quality attributes that we evaluate. We present our evaluation results in Section 4. In Section 5 we discuss related work and, finally, in Section 6 we conclude our study.

2. Background

In this section, we give some background about software architectures, how to evaluate them, different quality attributes, and the application domain, i.e., automated guided vehicle systems.

2.1. Software Architecture

Software systems are developed based on a requirement specification. The requirements can be categorized into *functional* requirements and *non-functional* requirements, also called *quality requirements*. Functional requirements are often easiest to test (the software either has the required functionality or not) but the non-functional requirements are harder to test (quality is hard to define and quantify).

In the recent years, the domain of software architecture [3, 5, 18, 20] has emerged as an important area of research in software engineering. This is in response to the recognition that the architecture of a software system often constrains the quality attributes. Software architecture is defined in [3] as follows:

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."

Software architectures have theoretical and practical limits for quality attributes that may cause the quality requirements not to be fulfilled. If no analysis is done during architectural design, the design may be implemented with the intention to measure the quality attributes and optimize the system. However, the architecture of a software system is fundamental to its structure and cannot easily be changed without affecting virtually all components and, consequently, considerable effort. It has also been shown that several quality attributes can be in conflict with each other, e.g., maintainability and performance [9]. Therefore, it is important to evaluate all (or at least the most) relevant quality attributes at the software architecture level.

2.2. Evaluation Methodology

In order to make sure that a software architecture fulfills its quality requirements, it has to be evaluated. Four main approaches to architecture evaluation can be identified, i.e., scenarios, simulation, mathematical modelling, and experience-based reasoning [5]. In this paper we use a prototype-based architecture evaluation method which is part of the simulation-based approach and relies on the construction of an executable prototype of the architecture [5, 15, 21]. Prototype-based evaluation enables us to evaluate software components in an execution environment. It also lets the developer compare all components in a fair way, since all components get the same input from a simplified architecture model. An overview of the parts that go into a prototype is shown in Figure 1. A strength of this evaluation approach is that it is possible to make accurate measurements on the intended target platform for the system early on in the development cycle.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

The prototype-based evaluation is performed in seven steps plus reiteration. We will describe the steps shortly in the following paragraphs.

Define the evaluation goal. In this first step two things are done. First, the environment that the simulated architecture is going to interact with is defined. Second, the abstraction level that the simulation environment is to be implemented at is defined (high abstraction gives less detailed data, low abstraction gives accurate data but increases model complexity).

Implement an evaluation support framework. The evaluation support framework's main task is to gather data that is relevant to fulfilling the evaluation goal. Depending on the goal of the evaluation, the support framework has to be designed accordingly, but the main task of the support framework is to simplify the gathering of data. The support framework can also be used to provide common functions such as base and utility classes for the architecture models.

Integrate architectural components. The component of the architecture that we want to evaluate has to be adapted so that the evaluation support framework can interact with it. The easiest way of achieving this is to create a proxy object that translates calls between the framework and the component.

Implement architecture model. Implement a model of the architecture with the help of the evaluation support framework. The model should approximate the behavior of the completed system as far as necessary. The model together with the evaluation framework and the component that is evaluated is compiled to an executable prototype.

Execute prototype. Execute the prototype and gather the data for analysis in the next step. Try to make sure that the execution environment matches the target environment as close as possible.

Analyse logs. Analyse the gathered logs and extract information regarding the quality attributes that are under evaluation. Automated analysis support is preferable since the amount of data easily becomes overwhelming.

Predict quality attribute. Predict the quality attributes that are to be evaluated based on the information from the analysed logs.

Reiteration. This goes for all the steps in the evaluation approach. As the different steps are completed it is easy to see things that were overlooked during the previous step or steps. Once all the steps has been completed and results from the analysis are available, you should review them and use the feedback for deciding if adjustments have to be done to the prototype. These adjustments can be necessary in both the architecture model and the evaluation support framework. It is also possible to make a test run to validate that the analysis tools are working correctly and that the data that is gathered really is useful for addressing the goals of the evaluation.

2.3. Automated Guided Vehicle Systems

As an industrial case we use an Automated Guided Vehicle system (AGV system) [7]. AGV systems are used in industry mainly for supply and materials handling, e.g., moving raw materials, and finished products to and from production machines.

Central to an AGV system is the ability to automatically drive a vehicle along a predefined path, the path is typically stored in a path database in a central server and distributed to the vehicles in the system when they are started. The central server is responsible for many things in the system, it keeps track of the vehicles positions and uses the information for routing and guiding the vehicles from one point in the map to another. It also manages collision avoidance so that vehicles do not run into each other by accident and it detects and resolves deadlocks when several vehicles want to pass the same part of the path at the same time. The central server is also responsible for the handling of orders from operators. When an order is submitted to the system, e.g., "go to location A and load cargo", the server selects the closest free vehicle and begins to guide it towards the pickup point.

In order for the central server to be able to perform its functions, it has to know the exact location of all vehicles under its control on the premise. Therefore every vehicle sends its location to the server several times every second. The vehicles can use one or several methods to keep track of its location. The three most common methods are induction wires, magnetic spots, and laser range finders.

The first method, and also the simplest, is to use induction wires that are placed in the floor of the premises. The vehicles are then able to follow the electric field that the wire emits and from the modulation of the field determine where it is. A second navigation method is to place small magnetic spots at known locations along the track that the truck is to follow. The truck can then predict where it is based on a combination of dead reckoning and anticipation of coming magnetic spots. A third method is to use a laser located on the vehicle, that measures distances and angles from the vehicle to a set of reflectors that has been placed at known locations throughout the premises. The control system in the vehicle is then able to calculate its position in a room based on the data returned from the laser.

Regardless of the way that a vehicle acquires the information of where it is, it must be able to communicate its location to the central control computer. Depending on the available infrastructure and environment in the premises of the system, it can for example use radio modems or a wireless LAN.

The software in the vehicle can be roughly divided into three main components that continuously interact in order to control the vehicle. These components require commu-

nication both within processes and between processes located on different computers. We will perform an evaluation of the communication components used in an existing AGV system and compare them to an alternative COTS component for communication that is considered for a new version of the AGV system.

3. Component Quality Attribute Evaluation

In this section we describe the components that we evaluate, as well as the evaluation methods used. The goal is to assess three quality attributes, i.e., performance, portability and maintainability for each component. The prototypes simulate the software that is controlling the vehicles in the AGV system. The central server is not part of the simulation.

3.1. Evaluated Communication Components

The components we evaluate are all communication components. They all distribute events or messages between threads within a process and/or between different processes over a network connection. Two of the components are developed by the company we are working with. The third component is an open source implementation of the CORBA standard [16].

NDC Dispatcher. The first component is an implementation of the dispatcher pattern which provides publisher-subscriber functionality and adds a layer of indirection between the senders and receivers of messages. It is used for communication between threads within one process and can not pass messages between processes. The NDC Dispatcher is implemented with active objects using one thread for dispatching messages and managing subscriptions. It is able to handle distribution of messages from one sender to many receivers. The implementation uses the ACE framework for portability between operating systems. This component is developed by the company and is implemented in C++.

Network Communication Channel. Network Communication Channel (NCC) is a component is developed by the company as well. It is designed to provide point to point communication between processes over a network. It only provides one to one communication and has no facilities for managing subscriptions to events or message types. NCC can provide communication with legacy protocols from previous versions of the control system and can also provide communication over serial ports. This component is developed by the company and is implemented in C.

TAO Real-time Event Channel. The third component, The ACE Orb Real-time Event Channel (TAO RTEC) [19], can be used for communication between both threads within a process, and between processes both on the same computer and over a network. It provides communication from one to many through the publisher-subscriber pattern. The event channel is part of the TAO CORBA implementation and is open source. This component can be seen as a commercial of-the-shelf (COTS) component to the system. We use TAO Real-time Event Channel to distribute messages in the same way that the NDC Dispatcher does.

3.2. Software Quality Attributes to Evaluate

In our study we are interested in several quality attributes. The first is performance because we are interested in comparing how fast messages can be delivered by the three components. We assess the performance at the system level and look at the performance of the communication subsystem as a whole.

The second attribute is maintainability which was selected since the system will continue to be developed and maintained under a long period. The selected communication component will be an integral part of the system, and must therefore be easy to maintain.

The third attribute is portability, i.e., how much effort is needed in order to move a component from one environment/platform to another. This attribute is interesting as the system is developed and to some extent tested on computers running Windows, but the target platform is based on Linux.

Performance. We define performance as the time it takes for a communication component to transfer a message from one thread or process to another. In order to measure this we created one prototype for each communication component. The prototypes were constructed using a framework that separates the communication components from the model of the interaction, i.e., the architecture. As a result, we can use the same interaction model for the prototypes and minimize the risk of treating the communication components unequally in the test scenarios. The framework from a previous study [15] was reused, and functionality was added for measuring the difference in time for computers that were connected via a network. This information was used to adjust the timestamps in the logs when prototypes were running on separate computers, and to synchronize the start time for when the prototypes should start their models.

We created two different models of the interaction: one for communication between threads in a process and one for communication between processes on separate computers that communicate via a network. The NDC Dispatcher

was tested with the model for communication between threads in a process and NCC was tested with the model for communication over the network. TAO RTEC was tested with both models since it can handle both cases.

An example of a model can be seen in Figure 2 which shows the interaction between threads in a process for the NDC Dispatcher prototype. Here, thread A sends a message to thread B every 50 ms. Thread B sends the message on to thread C. Thread C sends it to thread D which in turn send it back to thread B. Each message is marked with a timestamp and stored to a logfile for offline analysis.

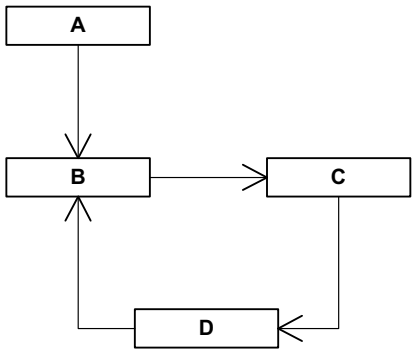

Figure 2. Interaction within a prototype.

The prototypes were executed three times on a test platform similar to the target environment and we calculated the average response time of the three runs. The test environment consisted of two computers with a 233Mhz Pentium 2 processor and 128 MB RAM each. Both computers were running the Linux 2.4 kernel and they were connected with a dedicated 10Mbps network.

Maintainability. We use a tool called CCCC [13, 14] to collect a number of measures (e.g., number of modules, lines of code, and cyclomatic complexity) on the source code of the components. The objective is to use these measures to calculate a maintainability index metric [17] for the components. The maintainability index (MI) is a combination of the average halstead volume per module (aveVol), the average cyclomatic complexity per module (aveV(g')), average lines of code per module (aveLoc), and average percentage of lines of comments per module (aveCM), as shown in Figure 3. The maintainability index calculation results in a value that should be as high as possible. Values above 85 are considered to indicate good maintainability, between 85 and 65 is medium maintainability, and finally, values below 65 are indicating low maintainability [17]. Based on the maintainability index together with our qualitative experiences from developing the prototypes, we evaluate and compare the maintainability of the components. We do not see the maintainability index as a definite judgement of the maintainability of the components but more as a tool to indicate the properties of the components and to make them comparable.

Portability. We define portability as the effort needed to move the prototypes and communication components from a Windows XP based platform to a Linux 2.4 based platform. This is a simple way of assessing the attribute but it verifies that the prototypes actually works on the different platforms and it gives us some experience from making the port. Based on this experience we can make a qualitative comparison of the three components.

4. Evaluation Results

During the evaluation, the largest effort was devoted to implementing the three prototypes and running the performance benchmarks. The data from the performance benchmarks gave us quantitative performance figures which together with the experience from the implementations were used to assess the maintainability and portability of the components.

4.1. Performance Results

After implementing the prototypes and performing the test runs, the gathered logs were processed by an analysis tool that merged the log entries, compensated for the differences in time on the different machines and calculated the time it took to transfer each message.

4.2. Intra Process Communication

The intra process communication results in Table 1 show that the average time it takes for the NDC Dispatcher to deliver a message is 0,3 milliseconds. The same value for TAO RTEC is 0,6 milliseconds. The extra time that it takes for TAO RTEC is mainly due to the differences in size between TAO RTEC and the NDC Dispatcher. TAO RTEC makes use of a CORBA ORB for dispatching the events between the threads in the prototype. This makes TAO RTEC very flexible but it impacts its performance

$$MI = 171 - 5.2 \times \ln(aveVol) - 0.23 \times aveV(g') - 16.2 \times \ln(aveLoc) + 50 \times \sin(\sqrt{2.46 \times aveCM})$$

Figure 3. Formula for calculating the maintainability index (MI) [17].

when both publisher and subscriber are threads within the same process; the overhead in a longer code path for each message becomes a limiting factor. The NDC Dispatcher on the other hand is considerably smaller in its implementation than TAO RTEC, resulting in a shorter code path and faster message delivery.

Table 1. Intra process communication times.

	NDC Dispatcher	TAO RTEC
Intra process	0,3 ms	0,6 ms

During the test runs of the NDC Dispatcher and the TAO RTEC based prototypes we saw that the time it took to deliver a message was not the same for all messages. Figure 4 and Figure 5 show a moving average of the measured delivery times in order to illustrate the difference in behavior between the components. In both the NDC Dispatcher and the TAO RTEC prototypes this time depends on how many subscribers there are to the message, and the order the subscribers subscribed to a particular message. We also saw that there is a large variation in delivery time from message to message when using TAO RTEC. It is not possible to guarantee that the time it takes to deliver a message will be constant when using neither the NDC Dispatcher nor TAO RTEC, but the NDC Dispatcher's behavior is more predictable.

Figure 4. NDC Dispatcher message delivery time (moving average).

4.3. Inter Process Communication

The inter process communication is evaluated in two different environments; when the communicating processes are on the same computer and when they reside on two different computer connected by a network.

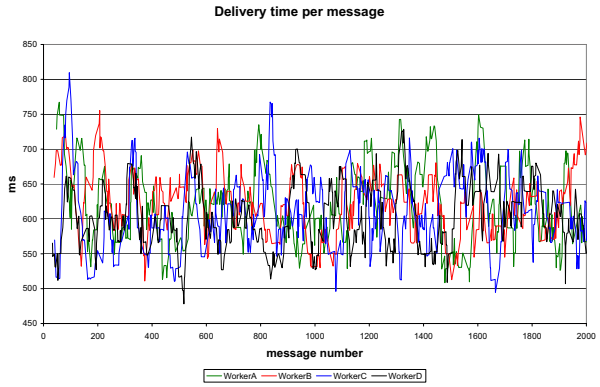

Figure 5. TAO RTEC message delivery time (moving average).

In Table 2 we present the message delivery times when the processes reside on the same computer. We find that TAO RTEC takes 2,0 milliseconds on average to deliver a message, while NCC only takes 0,8 milliseconds to deliver a message. Much of the difference comes from the fact that TAO RTEC offers much more flexibility, e.g., communication one-to-many, while NCC only provides one-to-one communication. Another contributing factor is that TAO RTEC runs the event channel in a separate process from the prototypes. This results in an added delay as messages are sent to the event channel process before they are delivered to the recipient process. NCC on the other hand, delivers the messages directly to the recipient process.

The inter process communication in Table 3 shows that TAO RTEC takes on average 2 milliseconds to deliver a message from one computer to another in our test environment. NCC takes on average 1 millisecond. The extra time needed for TAO RTEC to deliver a message is, as discussed earlier, a result of the longer code path involved due to the use of CORBA, and the need of an intermediary process for distributing the messages to the subscribers. The gain of using this component is added flexibility in how messages can be distributed between subscribers on different computers. In comparison, the NCC component is only able to pass messages from one point to another, making it less complex in its implementation.

Table 2. Communication times between processes running on the same computer.

	TAO RTEC	NCC
Inter process	2,0 ms	0,8 ms

In Table 4 we present the amount of data that is transmitted (and in how many TCP/IP packages) over the net-

Table 3. Communication times between processes running on different computers.

	TAO RTEC	NCC
Inter process over network	2,0 ms	1,0 ms

work by prototypes using TAO RTEC and NCC, respectively. In the architecture model, both prototypes perform the same work and send the same number of messages over the network. In the table we see that both components send about the same number of TCP/IP packages (TAO RTEC sends 37 more than NCC). The difference is located to the initialization of the prototypes where a number of packages are sent during ORB initialization, name resolution, and subscriptions to the event channel etc. When we look at the amount of data sent in the packages we see that TAO RTEC sends about 55% more data than NCC does. This indicates that NCC has less overhead per message than TAO RTEC has. Both components do however add considerably to the amount of data that is generated by the model, which generated 6 kb of data in 300 messages.

Table 4. Network traffic generated by TAO RTEC and NCC.

	TAO RTEC	NCC
TCP/IP packages	800 packages	763 packages
Data over network	137 kb	88 kb

In summary, we find that TAO RTEC has half the performance of both the NDC Dispatcher and NCC for both intra- and inter-process communication. However, TAO RTEC has the advantage that it can handle both intra- and inter-process communication using the same communication component, while the NDC Dispatcher and NCC can handle only one type of communication (either intra-process or inter-process).

4.4. Maintainability Results

The measures that we gathered using CCCC are listed in Table 5, and the metrics are defined as follows. Modules (MOD) is the number of classes and modules with identified member functions. Lines of code (LOC) and Lines of comments (COM) are measures for the source code, and a combination of them gives an indication of how well documented a program is (LOC/COM and LOC/MOD). The COM measure can also be combined with the cyclomatic complexity (CYC) to give an indication of how well documented the code is in relation to the code complexity. The cyclomatic complexity is also used in combination with the module count in order to give an indication of the program complexity per module (CYC/MOD). When analyzing the results we put the most weight on compound measures such as the maintainability index, cyclomatic complexity per comment, and cyclomatic complexity per module.

Table 5. Metrics from the CCCC tool [14].

Metric	NDC Dispatcher	NCC	TAO RTEC
Modules (MOD)	23	57	3098
Lines of code (LOC)	533	23982	312043
Lines of comments (COM)	128	19827	78968
LOC/COM	4,164	1,210	3,952
LOC/MOD	23,174	420,737	100,724
Cyclomatic complexity (CYC)	69	3653	34927
CYC/COM	0,539	0,184	0,442
CYC/MOD	3,0	64,088	11,274
Maintainability Index	128,67	50,88	78,91

The NDC Dispatcher is the smallest of the three components with 533 lines of code in 23 modules (see Table 5). The complexity per module is the lowest but the complexity per comment is the highest of all the components. While working with this component we found it easy to use and easy to get an overview of. The component also has the highest maintainability index (128,67) of the three components, indicating a high maintainability.

NCC is 23982 lines of code in 57 modules. It is also the most commented component of the three, which is shown in the low cyclomatic complexity per comment value. However, there are indications in the LOC/MOD and CYC/MOD measures that the component has very large modules. This can make NCC difficult to overview, thus lowering its maintainability. The maintainability index supports this assessment, since NCC is the component with the lowest maintainability index (50,88) indicating poor maintainability.

TAO RTEC is 312043 lines of code in 3098 modules. This is by far the largest component of the three. Although the parts that are used for the real-time communication channel are smaller (we gathered metrics for all the parts of TAO) it is still difficult to get an overview of the source

code. The maintainability index for TAO RTEC (78,91) puts it in the medium maintainability category. We do, however, think that the size of the component makes it difficult to maintain within the company. The question of maintainability is relevant only if one version of TAO is selected for continued use in the company. If newer versions of TAO are used as they are released then the maintenance is continuously done by the developer community around TAO. On the other hand, there is a risk that API:s in TAO are changed during development, breaking applications. But since the application developers are with the company, this problem is probably easier to deal with than defects in TAO itself.

4.5. Portability Results

Based on our experiences from building the prototypes we found that moving the prototypes from the Windows-based to the Linux-based platform was generally not a problem and did not take very long time (less than a day per prototype). Most of the time was spent on writing new makefiles and not on changing the code for the prototypes.

Both the NDC Dispatcher and TAO RTEC are developed on top of the ADAPTIVE Communications Environment (ACE) [19]. ACE provides a programming API that has been designed to be portable to many platforms. Once ACE was built on the Linux platform it was easy to build the prototypes that used it.

NCC was originally written for the Win32 API and uses a number of portability libraries built to emulate the necessary Win32 API:s on platforms other than windows. Building the prototype using NCC was not more complicated than those using the NDC Dispatcher or TAO RTEC.

5. Related Work

Prototypes are commonly used in interface design, where different alternatives to graphical user interfaces can be constructed and tested by users and developers [4]. The use of prototypes for architecture simulation and evaluation has been described and discussed in [2, 15]. The goal is to evaluate architectural alternatives before the detailed design documents have been developed, making it possible to obtain performance characteristics for architecture alternatives and hardware platform working together. Other commonly used performance models are queueing networks, stochastic petri nets, stochastic process algebra, and simulation models [1]. Software Performance Engineering based and architectural-pattern based approaches both use information obtained from UML design documents (Use Case, System Sequence, and Class diagrams) for the evaluation of the software architecture. This makes it possible to make performance evaluations as soon as the design of the system begins to take shape. A weakness of these performance evaluation models is that it is difficult to capture the dynamic properties of the executing code when it interacts with the operating system.

Several methods for evaluating one aspect of a components quality attributes have been described [22]. Most of the methods focus on the evaluation of different performance aspects of components. However, when selecting components it is likely that more than the performance attribute is of interest for the developers, this result in a need to perform evaluations for several quality attributes for the components. Qualities such as maintainability can be quantified and compared using for example the maintainability index [17]. Using tools for static analysis of the source code of the components makes it possible to extract complexity and maintainability metrics for components.

Methods for assessing several quality attributes during an evaluation exist in several architecture level evaluation methods. Methods such as the scenario-based Software Architecture Analysis Method (SAAM) [10] and Architecture Tradeoff Analysis Method (ATAM) [11], as well as the attribute-based ABAS [12] method can be used to assess a number of quality attributes using scenario-based evaluation. Especially ATAM tries to handle several quality attributes and their impact on the software architecture simultaneously. The evaluation methods that we used in this paper can be used to supply input for both SAAM and ATAM. In addition, the method that we have used in this paper can also complement the results from SAAM and ABAS, i.e., they focus around qualitative reasoning while our method provides quantitative data. Together, the methods can address a broader spectrum of quality attributes.

6. Conclusions

In this paper we have used a prototype-based evaluation methods for assessing three quality attributes of three different communication components. We have shown that it is possible to compare the three evaluated components in a fair way using a common framework for building the prototypes and analyzing the resulting data. The components were one COTS component, i.e., The ACE Orb Real-Time Event Channel (TAO RTEC), and two inhouse developed components, the NDC Dispatcher and NCC. For each of the components we have evaluate three quality attributes: performance, maintainability, and portability. The performance and maintainability are evaluated quantitatively, while portability is evaluated qualitatively.

The performance measurements show that TAO RTEC has half the performance of the NDC Dispatcher in communication between threads within a process. Our results also show that TAO RTEC has approximately half the performance of NCC in communication between processes.

On the other hand, TAO RTEC provides functionality for both intra- and inter-process communication, while the NDC Dispatcher and NCC only support one type of communication.

As for the maintainability, the NDC Dispatcher has the highest maintainability index of the three components (it indicated a high maintainability for the component). NCC turned out to have the lowest maintainability index (so low that it indicated a low maintainability for the component). However, even though NCC has the lowest maintainability index of all the components, we think that it is rather easy for the company to maintain since it has been developed within the company and is well documented. TAO RTEC is the largest of the three components, with a medium high maintainability index, and the knowledge of how it is constructed is not within the company. Therefore, we think that TAO RTEC is less maintainable for the company. On the other hand, the company can take advantage of future development of TAO RTEC with little effort as long as the API:s remain the same.

Finally, considering the portability aspects. All three evaluated components fulfill the portability requirement in this study. We had no problems moving the prototypes from a Windows-based to a Linux-based platform.

Acknowledgments

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" http://www.bth.se/besq. We would like to thank Danaher Motion Särö AB [6] for providing us with a case for our case study and many interesting discussions and ideas.

References

[1] S. Balsamo, A. Di Marco, P Inverardi, and M Simeoni, "Model-based Performance Prediction in Software Development: A Survey," *IEEE Transactions on Software Engineering*, 30(5):295-310, May 2004.

[2] J. E. Bardram, H. B. Christensen, K. M. Hansen, "Architectural Prototyping: An Approach for Grounding Architectural Design and Learning," *Proc. 4th Working IEEE/IFIP Conf. on Software Architecture (WICSA '04)*, pp. 15-24, 2004.

[3] L. Bass, P. Clements, and R. Kazman, *"Software Architecture in Practice,"* Addison-Wesley, 1998.

[4] D. Baumer, W. Bischofberger, H. Lichter, and H. Zullighoven, "User Interface Prototyping-Concepts, Tools, and Experience," *Proc. 18th International Conference on Software Engineering*, pp. 532-541, 1996.

[5] J. Bosch: *"Design & Use of Software Architectures,"* Pearson Education Limited, ISBN 0-201-67494-7.

[6] Danaher Motion Särö AB, http://www.danahermotion.se

[7] D.A. Davis, "Modeling AGV Systems," *Proc. 1986 Winter Simulation Conference*, pp. 568-573, Dec. 1986.

[8] L. Dobrica, E. Niemela, "A Survey On Architecture Analysis Methods," *IEEE Transactions on Software Engineering,* 28(7):638 - 653, July 2002.

[9] D. Häggander, L. Lundberg, and J. Matton, "Quality Attribute Conflicts - Experiences from a Large Telecommunication Application," *Proc. 7th IEEE International Conference of Engineering of Complex Computer Systems*, pp. 96-105, June 2001.

[10] R. Kazman, L. Bass, G. Abowd, and M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proc. 16th International Conference of Software Engineering*, pp. 81-90, 1994.

[11] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and S.J. Carrière, "The Architecture Tradeoff Analysis Method," *Proc. 4th IEEE International Conference of Engineering of Complex Computer Systems*, pp. 68-78, August 1998, Monterey, CA.

[12] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson, "Attribute-Based Architecture Styles," *Proc. First Working IFIP Conference on Software Architecture (WICSA-1)*, pp. 225-243, February 1999, San Antonio, TX.

[13] T. Littlefair, *"An Investigation Into the Use of Software Code Metrics in the Industrial Software Development Environment,"* Ph.D. thesis, Edith Cowan University, Australia, June 2001.

[14] T. Littlefair, "CCCC," available at http://cccc.sourceforge.net/ last checked November 2004.

[15] F. Mårtensson, H. Grahn, and M. Mattsson, "An Approach for Performance Evaluation of Software Architectures using Prototyping," *Proc. Int'l Conference on Software Engineering and Applications (SEA 2003)*, pp. 605-612, Nov. 2003.

[16] Object Management Group, *"CORBA™/IIOP™ Specification, 3.0,"* Mar. 2004, available at www.omg.org.

[17] T. Pearse and P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities," *Proc. International Conference on Software Maintenance*, pp. 295-303, 1995.

[18] D.E. Perry and A.L.Wolf, "Foundations for the Study of Software Architecture," *Software Engineering Notes*, 17(4):40-52, October 1992.

[19] D. Schmidt et al., "The ACE ORB," available at http://www.cs.wustl.edu/~schmidt/TAO.html. Checked Sept. 2004.

[20] M. Shaw and D. Garlan, *"Software Architecture - Perspectives on an Emerging Discipline,"* Prentice Hall, ISBN 0-13-182957-2.

[21] C. Smith and L. Williams, *"Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software,"* Addison-Wesley, 2001.

[22] S. Yacoub, "Performance analysis of component-based applications," *Proc. Second International Conference on Software Product Lines, SPLC 2*, pp. 299-315, Aug. 2002.