Evaluation of Component Technologies with Respect to Industrial Requirements*

Anders Möller[1,2], Mikael Åkerholm[1], Johan Fredriksson[1], Mikael Nolin[1]
[1]Mälardalen Real-Time Research Centre (MRTC), Västerås
[2]CC Systems, www.cc-systems.com
Sweden

E-mail: anders.moller@mdh.se

Abstract

We compare existing component technologies for embedded systems with respect to industrial requirements. The requirements are collected from the vehicular industry, but our findings are applicable to similar industries developing resource constrained safety critical embedded distributed real-time computer systems.

One of our conclusions is that none of the studied technologies is a perfect match for the industrial requirements. Furthermore, no single technology stands out as being a significantly better choice than the others; each technology has its own pros and cons.

The results of our evaluation can be used to guide modifications or extensions to existing technologies, making them better suited for industrial deployment. Companies that want to make use of component-based software engineering as available today can use this evaluation to select a suitable technology.

1 Introduction

Component-Based Software Engineering (CBSE) has received much attention during the last couple of years. However, in the embedded-system domain, use of component technologies has had a hard time gaining acceptance; software-developers are still, to a large extent, using monolithic and platform-dependent software technologies.

We try to find out why embedded-software developers have not embraced CBSE as an attractive tool for software development. We do this by evaluating a set of component technologies with respect to industrial requirements. The requirements have been collected from industrial actors within the business segment of heavy vehicles, and have been presented in our previous work [10]. Examples of heavy vehicles include wheel loaders, excavators, forest harvesters, and combat vehicles. The software systems developed within this market segment can be characterised as resource constrained, safety critical, embedded, distributed, real-time, control systems. Our findings in this study should be applicable to other domains with similar characteristics.

Our evaluation, between requirements and existing technologies, does not only help to answer why component-based development has not yet been embraced by the embedded-systems community. It also helps us to identify what parts of existing technologies could be enhanced, to make them more appropriate for embedded-system developers. Specifically, it will allow us to select a component technology that is a close match to the requirements, and if needed, guide modifications to that technology.

The reason for studying component-based development in the first place, is that software developers can achieve considerable business benefits in terms of reduced costs, shortened time-to-market and increased software quality by applying a suitable component technology. The component technology should rely on powerful design and compile-time mechanisms and simple and predictable run-time behaviour.

There is however significant risks and costs associated with the adoption of a new development technique (such as component-based development). These must be carefully evaluated before introduced in the development process. One of the apparent risks is that the selected component technology turns out to be inappropriate for its purpose; hence, the need to evaluate component technologies with respect to requirements expressed by software developers.

2 Requirements

The requirements discussed and described in this section are based on a previously conducted investigation [10]. The requirements found in that investigation are divided into two main groups, the technical requirements (section 2.1) and the development process related requirements (section 2.2).

*This work is supported by the Knowledge Foundation (KKS) and SSF within the projects HEAVE and SAVE.

In *Euromicro Conference, Component-Based Software Engineering Track*, Rennes, France, August, 2004

In addition, section 2.3 contains derived requirements, i.e. requirements that we have synthesised from the requirements in sections 2.1 and 2.2 but that are not explicitly stated requirements from the vehicular industry [10].

2.1 Technical Requirements

The technical requirements describe industrial needs and desires regarding technical aspects and properties of a component technology.

2.1.1 Analysable

System analysis, with respect to non-functional properties, such as timing behaviour and memory consumption is considered highly attractive. In fact, it is one of the single most distinguished requirements found in our investigation.

When analysing a system built from well-tested, functionally correct, components, the main issue is associated with composability. The composition process must ensure that non-functional properties, such as the communication, synchronisation, memory, and timing characteristics of the system, are predictabe [3].

2.1.2 Testable and debugable

It is required that tools exist that support debugging, both at component level (e.g., a graphical debugging tool), as well as on source code level.

Testing and debugging is one of the most commonly used techniques to verify software systems functionality. Testing is a very important complement to analysis, and testability should not be compromised when introducing a component technology. Ideally, the ability to test embedded-system software should be improved when using CBSE, since it adds the ability to test components in isolation.

2.1.3 Portable

The components, and the infrastructure surrounding them, should be platform independent to the highest degree possible. Here, platform independency means (1) hardware independent, (2) real-time operating system (RTOS) independent and (3) communications protocol independent. The components are kept portable by minimising the number of dependencies to the software platform. Eventually such dependencies are off course necessary to construct an executable system, however the dependencies should be kept to a minimum, and whenever possible dependencies should be generated automatically by configuration tools.

2.1.4 Resource Constrained

The components should be small and light-weighted and the components infrastructure and framework should be minimised. Ideally there should be no run-time overhead compared to not using a CBSE approach. Hardware used in embedded real-time systems is usually resource constrained, to lower production cost and thereby increase profit.

One possibility, that significantly can reduce resource consumption of components and the component framework, is to limit run-time dynamics. This means that it is desirable only to allow static, off-line, configured systems. Many existing component technologies have been design to support high run-time dynamics, where components are added, removed and reconfigured during run-time.

2.1.5 Component Modelling

The component modelling should be based on a standard modelling language like UML [15] or UML 2.0 [13]. The main reason to choose a standard like UML is that it is well known and thoroughly tested, with tools and formats supported by many third-party developers. The reason for the vehicular industry to have specific demands in this detail, is that this business segment does not have the knowledge, resources or desire to develop their own standards and practices.

2.1.6 Computational Model

Components should preferably be passive, i.e. they should not contain their own threads of execution. A view where components are allocated to threads during component assembly is preferred, since this is conceptually simple, and also believed to enhance reusability.

The computational model should be focused on a pipes-and-filters model [17]. This is partly due to the well known ability to schedule and analyse this model off-line. Also, the pipes-and-filters model is a good conceptual model for control applications.

2.2 Development Requirements

When discussing component-based development with idustry, development process requirements are at least as important as the technical requirements. To obtain industrial reliance, the development requirements need to be addressed by the component technology and its associated tools.

2.2.1 Introducible

Appropriate support to gradually migrate to a new technology should be provided by the component technology. It is

important to make the change in development process and techniques as safe and inexpensive as possible. Revolutionary changes in development techniques are associated with high risks and costs. Therefore a new technology should be possible to divide into smaller parts, which can be introduced incrementally. Another aspect, to make a technology introducible, is to allow legacy code within systems designed with the new technology.

2.2.2 Reusable

Components should be reusable, e.g., for use in new applications or environments than those for which they where originally designed [4]. Reusability can more easily be achieved if a loosely coupled component technology is used, i.e. the components are focusing on functionality and do not contain any direct operating system or hardware dependencies. Reusability is further enhanced by the possibility to use configuration parameters to components.

A clear, explicit, and well-defined component interface is crucial to enhance the software reusability. Also, specification of non-functional properties and requirements (such as execution time, memory usage, deadlines, etc.) simplify reuse of components since it makes (otherwise) implicit assumptions explicit. Behavioural descriptions (such as state diagrams or interaction diagrams) of components can be used to further enhance reusability.

2.2.3 Maintainable

The components should be easy to change and maintain, developers that are about to change a component need to understand the full impact of the proposed change. Thus, not only knowledge about component interfaces and their expected behaviour is needed. Also, information about current deployment contexts may be needed in order not to break existing systems. The components can be stored in a repository where different versions and variants need to be managed in a sufficient way. The maintainability requirement also includes sufficient tools supporting the service of deployed and delivered products. These tools need to be component aware and handle error diagnostics from components and support for updating software components.

2.2.4 Understandable

The component technology and the systems constructed using it should be easy to understand. This should also include making the technology easy and intuitive to use in a development project.

The reason for this requirement is to simplify evaluation and verification both on the system level and on the component level. Focusing on an understandable model makes the development process faster and it is likely that there will be fewer bugs. This requirement is also related to the introducible requirement (section 2.2.1) since an understandable technique is more introducible.

It is desirable to hide as much complexity as possible from system developers. Ideally, complex tasks (such as mapping signals to memory areas or bus messages, or producing schedules or timing analysis) should be performed by tools.

2.3 Derived Requirements

Here, we present requirements that we have synthesised from the requirements in sections 2.1 and 2.2, but that are not explicit requirements from industry.

2.3.1 Source Code Components

A component should be source code, i.e., no binaries. Companies are used to have access to the source code, to find functional errors, and enable support for white box testing (section 2.1.2). Since source code debugging is demanded, even if a component technology is used, black box components is undesirable. However, the desire to look into the components does not necessary imply a desire to be allowed to modify them.[1]

Using black-box components would lead to a fear of loosing control over the system behaviour (section 2.2.4). Provided that all components in the systems are well tested, and that the source code are checked, verified, and qualified for use in the specific surrounding, the companies might alleviate their source code availability.

Also with respect to the resource constrained requirement (section 2.1.4), source code components allow for unused parts of the component to be removed at compile time.

2.3.2 Static Configurations

Better support for the technical requirements of analysability (section 2.1.1), testability (section 2.1.2), and resource consumption (section 2.1.4), are achieved by using pre-runtime configuration. Here, configuration means both configuration of component behaviour and interconnections between components. Component technologies for use in the Office/Internet domain usually focus on dynamic configurations [9, 19]. This is of course appropriate in these specific domains, where one usually has access to ample resources. Embedded systems, however, face another reality; with resource constrained nodes running complex, dependable, control applications.

However, most vehicles can operate in different modes, hence the technology must support switches between a set

[1] This can be view as a "glass box" component model, where it possible to acquire a "use-only" license from a third party. This license model is today quite common in the embedded systems market.

of statically configured modes. Static configuration also improves the development process related requirement of understandability (section 2.2.4), since each possible configuration is known before run-time.

3 Component Technologies

In this section, existing component technologies for embedded systems are described and evaluated. The technologies originate both from academia and industry. The selection criterion for a component technology has firstly been that there is enough information available, secondly that the authors claim that the technology is suitable for embedded systems, and finally we have tried to achieve a combination of both academic and industrial technologies.

The technologies described and evaluated are PECT, Koala, Rubus Component Model, PBO, PECOS and CORBA-CCM. We have chosen CORBA-CCM to represent the set of technologies existing in the PC/Internet domain (other examples are COM, .NET [9] and Java Enterprise Beans [19]) since it is the only technology that explicitly address embedded and real-time issues. Also, the Windows CE version of .NET [9] is omitted, since it is targeted towards embedded display-devices, which only constitute a small subset of the devices in vehicular systems. The evaluation is based on existing, publically available, documentation.

3.1 PECT

A Prediction-Enabled Component Technology (PECT) [21] is a development infrastructure that incorporates development tools and analysis techniques. PECT is an ongoing research project at the Software Engineering Institute (SEI) at the Carnegie Mellon University.[2] The project focuses on analysis; however, the framework does not include any concrete theories - rather definitions of how analysis should be applied. To be able to analyse systems using PECT, proper analysis theories must be found and implemented and a suitable underlying component technology must be chosen.

A PECT include an abstract model of a component technology, consisting of a construction framework and a reasoning framework. To concretise a PECT, it is necessary to choose an underlying component technology, define restrictions on that technology (to allow predictions), and find and implement proper analysis theories. The PECT concept is highly portable, since it does not include any parts that are bound to a specific platform, but in practise the underlying technology may hinder portability. For modelling or describing a component-based system, the Construction and Composition Language (CCL) [21] is used. The CCL is not compliant to any standards. PECT is highly introducible, in principle it should be possible to analyse a part of an existing system using PECT. It should be possible to gradually model larger parts of a system using PECT. A system constructed using PECT can be difficult to understand; mainly because of the mapping from the abstract component model to the concrete component technology. It is likely that systems looking identical at the PECT-level behave differently when realised on different component technologies.

PECT is an abstract technology that requires an underlying component technology. For instance, how testable and debugable a system is depends on the technical solutions in the underlying run-time system. Resource consumption, computational model, reusability, maintainability, black- or white-box components, static- or dynamic-configuration are also not possible to determine without knowledge of the underlying component technology.

3.2 Koala

The Koala component technology [20] is designed and used by Philips[3] for development of software in consumer electronics. Typically, consumer electronics are resource constrained since they use cheap hardware to keep development costs low. Koala is a light weight component technology, tailored for Product Line Architectures [1]. The Koala components can interact with the environment, or other components, through explicit interfaces. The components source code is fully visible for the developers, i.e. there are no binaries or other intermediate formats. There are two types of interfaces in the Koala model, the provides- and the requires- interfaces, with the same meaning as in UML 2.0 [13]. The provides interface specify methods to access the component from the outside, while the required interface defines what is required by the component from its environment. The interfaces are statically connected at design time.

One of the primary advantages with Koala is that it is resource constrained. In fact, low resource consumption was one of the requirements considered when Koala was created. Koala use passive components allocated to active threads during compile-time; they interact through a pipes-and-filters model. Koala uses a construction called thread pumps to decrease the number of processes in the system. Components are stored in libraries, with support for version numbers and compatibility descriptions. Furthermore components can be parameterised to fit different environments.

Koala does not support analysis of run-time properties. Research has presented how properties like memory usage and timing can be predicted in general component-based systems, but the thread pumps used in Koala might cause some problems to apply existing timing analysis theories.

[2] Software Engineering Institute, CMU; http://www.sei.cmu.edu

[3] Phillips International, Inc; Home Page http://www.phillips.com

Koala has no explicit support for testing and debugging, but they use source code components, and a simple interaction model. Furthermore, Koala is implemented for a specific operating system. A specific compiler is used, which routes all inter-component and component to operating system interaction through Koala connectors. The modelling language is defined and developed in-house, and it is difficult to see an easy way to gradually introduce the Koala concept.

3.3 Rubus Component Model

The Rubus Component Model (Rubus CM) [8] is developed by Arcticus systems.[4] The component technology incorporates tools, e.g., a scheduler and a graphical tool for application design, and it is tailored for resource constrained systems with real-time requirements. The Rubus Operating System (Rubus OS) [7] has one time-triggered part (used for time-critical hard real-time activities) and one event-triggered part (used for less time-critical soft real-time activities). However, the Rubus CM is only supported by the time-triggered part.

The Rubus CM runs on top of the Rubus OS, and the component model requires the Rubus configuration compiler. There is support for different hardware platforms, but regarding to the requirement of portability (section 2.1.3), this is not enough since the Rubus CM is too tightly coupled to the Rubus OS. The Rubus OS is very small, and all component and port configuration is resolved off-line by the Rubus configuration compiler.

Non-functional properties can be analysed during desing-time since the component technology is statically configured, but timing analysis on component and node level (i.e. schedulability analysis) is the only analysable property implemented in the Rubus tools. Testability is facilitated by static scheduling (which gives predictable execution patterns). Testing the functional behaviour is simplified by the Rubus Windows simulator, enabling execution on a regular PC.

Applications are described in the Rubus Design Language, which is a non-standard modelling language. The fundamental building blocks are passive. The interaction model is the desired pipes-and-filters (section 2.1.6). The graphical representation of a system is quite intuitive, and the Rubus CM itself is also easy to understand. Complexities such as schedule generation and synchronisation are hidden in tools.

The components are source code and open for inspection. However, there is no support for debugging the application on the component level. The components are very simple, and they can be parameterised to improve the possibility to change the component behaviour without changing the component source code. This enhances the possibilities to reuse the components.

Smaller pieces of legacy code can, after minor modifications, be encapsulated in Rubus components. Larger systems of legacy code can be executed as background service (without using the component concept or timing guarantees).

3.4 PBO

Port Based Objects (PBO) [18] combines object oriented design, with port automaton theory. PBO was developed as a part of the Chimera Operating System (Chimera OS) project [6], at the Advanced Manipulators Laboratory at Carnegie Mellon University.[5] Together with Chimera, PBO forms a framework aimed for development of sensor-based control systems, with specialisation in reconfigurable robotics applications. One important goal of the work was to hide real-time programming and analysis details. Another explicit design goal for a system based on PBO was to minimise communication and synchronisation, thus facilitating reuse.

PBO implements analysis for timeliness and facilitates behavioural models to ensure predictable communication and behaviour. However, there are few additional analysis properties in the model. The communication and computation model is based on the pipes-and-filters model, to support distribution in multiprocessor systems the connections are implemented as global variables. Easy testing and debugging is not explicitly addressed. However, the technology relies on source code components and therefore testing on a source code level is achievable. The PBOs are modular and loosely coupled to each other, which admits easy unit testing. A single PBO-component is tightly coupled to the Chimera OS, and is an independent concurrent process.

Since the components are coupled to the Chimera OS, it can not be easily introduced in any legacy system. The Chimera OS is a large and dynamically configurable operating system supporting dynamic binding, it is not resource constrained.

PBO is a simple and intuitive model that is highly understandable, both at system level and within the components themselves. The low coupling between the components makes it easy to modify or replace a single object. PBO is built with active and independent objects that are connected with the pipes-and-filters model. Due to the low coupling between components through simple communication and synchronisation the objects can be considered to be highly reusable. The maintainability is also affected in a good way due to the loose coupling between the components; it is easy to modify or replace a single component.

[4] Arcticus Systems; Home Page http://www.arcticus.se

[5] Carnegie Mellon University; Home Page http://www.cmu.edu

3.5 PECOS

PECOS[6] (PErvasive COmponent Systems) [22, 5] is a collaborative project between ABB Corporate Research Centre[7] and academia. The goal for the PECOS project was to enable component-based technology with appropriate tools to specify, compose, validate and compile software for embedded systems. The component technology is designed especially for field devices, i.e. reactive embedded systems that gathers and analyse data via sensors and react by controlling actuators, valves, motors etc. Furthermore, PECOS is analysable, since much focus has been put on non-functional properties such as memory consumption and timeliness.

Non-functional properties like memory consumption and worst-case execution-times are associated with the components. These are used by different PECOS tools, such as the composition rule checker and the schedule generating and verification tool. The schedule is generated using the information from the components and information from the composition. The schedule can be constructed off-line, i.e. a static pre-calculated schedule, or dynamically during runtime.

PECOS has an execution model that describes the behaviour of a field device. The execution model deals with synchronisation and timing related issues, and it uses Petri-Nets [16] to model concurrent activities like component compositions, scheduling of components, and synchronisation of shared ports [11]. Debugging can be performed using COTS debugging and monitoring tools. However, the component technology does not support debugging on component level as described in section 2.1.2.

The PECOS component technology uses a layered software architecture, which enhance portability (section 2.1.3). There is a Run-Time Environment (RTE) that takes care of the communication between the application specific parts and the real-time operating system. The PECOS component technology uses a modelling language that is easy to understand, however no standard language is used. The components communicate using a data-flow-oriented interaction, it is a pipes-and-filters concept, but the component technology uses a shared memory, contained in a blackboard-like structure.

Since the software infrastructure does not depend on any specific hardware or operating system, the requirement of introducability (section 2.2.1) is to some extent fulfilled. There are two types of components, leaf components (black-box components) and composite components. These components can be passive, active, and event triggered. The requirement of openness is not considered fulfilled, due to the fact that PECOS uses black-box components. In later releases, the PECOS project is considering to use a more open component model [23]. The devices are statically configured.

[6]PECOS Project, Home Page: http://www.pecos-project.org/

[7]ABB Corporate Research Centre in Ladenburg, Home Page: http://www.abb.com/

3.6 CORBA Based Technologies

The Common Object Request Broker Architecture (CORBA) is a middleware architecture that defines communication between nodes. CORBA provides a communication standard that can be used to write platform independent applications. The standard is developed by the Object Management Group[8] (OMG). There are different versions of CORBA available, e.g., MinimumCORBA [12] for resource constrains systems, and RT-CORBA [14] for time-critical systems.

RT-CORBA is a set of extensions tailored to equip Object Request Brokers (ORBs) to be used for real-time systems. RT-CORBA supports explicit thread pools and queuing control, and controls the use of processor, memory and network resources. Since RT-CORBA adds complexity to the standard CORBA, it is not considered very useful for resource-constrained systems. MinimumCORBA defines a subset of the CORBA functionality that is more suitable for resource-constrained systems, where some of the dynamics is reduced.

OMG has defined a CORBA Component Model (CCM) [2], which extends the CORBA object model by defining features and services that enables application developers to implement, mange, configure and deploy components. In addition the CCM allows better software reuse for server-applications and provides a greater flexibility for dynamic configuration of CORBA applications.

CORBA is a middleware architecture that defines communication between nodes, independent of computer architecture, operating system or programming language. Because of the platform and language independence CORBA becomes highly portable. To support the platform and language independence, CORBA implements an Object Request Broker (ORB) that during run-time acts as a virtual bus over which objects transparently interact with other objects located locally or remote. The ORB is responsible for finding a requested objects implementation, make the method calls and carry the response back to the requester, all in a transparent way. Since CORBA run on virtually any platform, legacy code can exist together with the CORBA technology. This makes CORBA highly introducible.

While CORBA is portable, and powerful, it is very run-time demanding, since bindings are performed during run-time. Because of the run-time decisions, CORBA is not very deterministic and not analysable with respect to timing

[8]Object Management Group. CORBA Home Page. http://www.omg.-org/corba/

	Analysable	Testable and debugable	Portable	Resource Constrained	Component Modelling	Computational Model	Introducible	Reusable	Maintainable	Understandable	Source Code Components	Static Configuration	Average	Number of 2's	Number of 0's
PECT	2	NA	2	NA	0	NA	2	NA	NA	0	NA	NA	1.2	3	2
Koala	0	1	1	2	0	2	0	2	2	2	2	2	1.3	7	3
Rubus Component Model	1	1	0	2	0	2	1	1	1	2	2	2	1.3	5	2
PBO	2	1	0	0	0	1	1	1	1	2	2	0	0.9	3	4
PECOS	2	1	2	2	0	2	1	2	1	2	0	2	1.4	7	2
CORBA Based Technologies	0	1	2	0	0	0	2	0	0	1	0	0	0.5	2	8
Average	1.2	1.0	1.2	1.2	0.0	1.4	1.4	1.2	1.0	1.5	1.2	1.2	1.1	4.3	3.5

Figure 1. Grading of component technologies with respect to the requirements

and memory consumption. There is no explicit modelling language for CORBA. CORBA uses a client server model for communication, where each object is active. There are no non-functional properties or any specification of interface behaviour. All these things together make reuse harder. The maintainability is also suffering from the lack of clearly specified interfaces.

4 Summary of Evaluation

In this section we assign numerical grades to each of the component technologies described in section 3, grading how well they fulfil each of the requirements of section 2. The grades are based on the discussion summarised in section 3. We use a simple 3 level grade, where 0 means that the requirement is not addressed by the technology and is hence not fulfilled, 1 means that the requirement is addressed by the technology and/or that is partially fulfilled, and 2 means that the requirement is addressed and is satisfactory fulfilled. For PECT, which is not a complete technology, several requirements depended on the underlying technology. For these requirements we do not assign a grade (indicated with NA, Not Applicable, in figure 1). For the CORBA-based technologies we have listed the best grade applicable to any of the CORBA flavours mentioned in section 3.6.

For each requirement we have also calculated an average grade. This grade should be taken with a grain of salt, and is only interesting if it is extremely high or extremely low. In the case that the average grade for a requirement is extremely low, it could either indicate that the requirement is very difficult to satisfy, or that component-technology designers have paid it very little attention.

In the table we see that only two requirements have average grades below 1.0. The requirement "Component Modelling" has the grade 0 (!), and "Testing and debugging" has 1.0. We also note that no requirements have a very high grade (above 1.5). This indicate that none of the requirement we have listed are general (or important) enough to have been considered by all component-technology designers. However, if ignoring CORBA (which is not designed for embedded systems) and PECT (which is not a complete component technology) we see that there are a handful of our requirements that are addressed and at least partially fulfilled by all technologies.

We have also calculated an average grade for each component technology. Again, the average cannot be directly used to rank technologies amongst each other. However, the two technologies PBO and CORBA stand out as having significantly lower average values than the other technologies. They are also distinguished by having many 0's and few 2's in their grades, indicating that they are not very attractive choices. Among the complete technologies with an average grade above 1.0 we notice Rubus and PECOS as being the most complete technologies (with respect to this set of requirements) since they have the fewest 0's. Also, Koala and PECOS can be recognised as the technologies with the broadest range of good support for our requirements, since they have the most number of 2's.

However, we also notice that there is no technology that fulfils (not even partially) all requirements, and that no single technology stands out as being the preferred choice.

5 Conclusion

In this paper we have compared some existing component technologies for embedded systems with respect to industrial requirements. The requirements have been collected from industrial actors within the business segment of heavy vehicles. The software systems developed in this segment can be characterised as resource constrained, safety critical, embedded, distributed, real-time, control systems.

Our findings should be applicable to software developers whose systems have similar characteristics.

We have noticed that, for a component technology to be fully accepted by industry, the whole systems development context needs to be considered. It is not only the technical properties, such as modelling, computation model, and openness, that needs to be addressed, but also development requirements like maintainability, reusability, and to which extent it is possible to gradually introduce the technology. It is important to keep in mind that a component technology alone cannot be expected to solve all these issues; however a technology can have more or less support for handing the issues.

The result of the investigation is that there is no component technology available that fulfil all the requirements. Further, no single component technology stands out as being the obvious best match for the requirements. Each technology has its own pros and cons. It is interesting to see that most requirements are fulfilled by one or more techniques, which implies that good solutions to these requirements exist.

The question, however, is whether it is possible to combine solutions from different technologies in order to achieve a technology that fulfils all listed requirements? Our next step is to assess to what extent existing technologies can be adapted in order to fulfil the requirements, or whether selected parts of existing technologies can be reused if a new component technology needs to be developed. Examples of parts that could be reused are file and message formats, interface description languages, or middleware specifications/implementations. Further, for a new/modified technology to be accepted it is likely that it have to be compliant to one (or even more than one) existing technology. Hence, we will select one of the technologies and try to make as small changes as possible to that technology.

References

[1] P. Clements and L. Northrop. *Software Product LInes*. Addison-Wesley, 2001. ISBN 0-201-70332-7.

[2] CORBA Component Model 3.0. Object Management Group, June 2002. http://www.omg.org/technology/documents/formal/components.htm.

[3] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[4] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth International Conference on Software Engineering*, April 1995. Seattle, USA.

[5] T. Genssler, A. Christoph, B. Schuls, M. Winter, et al. PECOS in a Nutshell. PECOS project http://www.pecos-project.org.

[6] P. Khosla et al. The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications. *IEEE Transactions on Systems*, 1992. Man and Cybernetics.

[7] K.L. Lundbäck. Rubus OS Reference Manual – General Concepts. Arcticus Systems: http://www.arcticus.se.

[8] K.L. Lundbäck and J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. Arcticus Systems: http://www.arcticus.se.

[9] Microsoft Component Technologies. COM/DCOM/.NET. http://www.microsoft.com.

[10] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proc. of the 7^{th} International Symposium on Component-Based Software Engineering*. 2004 Proceedings Series: Lecture Notes in Computer Science, Vol. 3054, May 2004. Edinburgh, Scotland.

[11] O. Nierstrass, G. Arevalo, S. Ducasse, et al. A Component Model for Field Devices. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, June 2002. Germany.

[12] Object Management Group. MinimumCORBA 1.0, August 2002. http://www.omg.org/technology/documents/formal/-minimum_CORBA.htm.

[13] Object Management Group. UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, 2003. http://www.omg.com/uml/.

[14] D. Schmidt, D. Levine, and S. Mungee. The Design of the tao real-time object request broker. *Computer Communications Journal*, Summer 1997.

[15] B. Selic and J. Rumbaugh. Using UML for modelling complex real-time systems, 1998. Rational Software Corporation.

[16] M. Sgroi. Quasi-Static Scheduling of Embedded Software Using Free-Choice Petri Nets. Technical report, University of California at Berkely, May 1998. Berkely, USA.

[17] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall; 1 edition, 1996. ISBN 0-131-82957-2.

[18] D. Stewart, R. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, pages pages 759 – 776, December 1997.

[19] Sun Microsystems. Enterprise Java Beans Technology. http://java.sun.com/products/ejb/.

[20] R. van Ommering et al. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000.

[21] K. C. Wallnau. Volume III: A Component Technology for Predictable Assembly from Certifiable Components. Technical report, Software Engineering Institute, Carnegie Mellon University, April 2003. Pittsburg, USA.

[22] M. Winter, T. Genssler, et al. Components for Embedded Software – The PECOS Apporach. In *The Second International Workshop on Composition Languages, in conjunction with the 16th ECOOP*, June 2002. Malaga, Spain.

[23] R. Wuyts and S. Ducasse. Non-functional requirements in a component model for embedded systems. In *International Workshop on Specification and Verification of Component-Based Systems*, 2001. OPPSLA.