MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-155/2004-1-SE

An Industrial Evaluation of Component Technologies for Embedded-Systems[1]

Anders Möller[*†], Mikael Åkerholm[*], Johan Fredriksson[*], Mikael Nolin[*]

[*] MRTC, Mälardalen University, Box 883, SE-721 23 Västerås, Sweden

[†] CC Systems, Home Page: http://www.cc-systems.com

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; and 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 the evaluation to select a suitable technology.

1 Introduction

During the last few years, Component-Based Software Engineering (CBSE) for embedded real-time systems has received much attention in the research community. However, industrial software-developers are still, to a large extent, using monolithic and platform-dependent software technologies. Especially in the embedded-systems domain, use of component technologies has had a hard time gaining acceptance.

In this paper 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 [10]. Example 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 should be applicable to other domains with similar characteristics.

[1] This work is supported KKS (The Knowledge Foundation) and SSF (Stiftelsen för Strategisk Forskning), within the projects HEAVE and SAVE/AutoComp.

This 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 see what modifications can be done in existing technologies to make them more appropriate for embedded-systems developers. Specifically, it will allow us to select a component technology that is a close match to the requirements and guide modifications to that technology to make it better suited for industrial use in the embedded system domain.

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. It is also considered very attractive by industry to allow engineers to practise CBSE without involving heavy run-time mechanisms. 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. These must be carefully evaluated before introduced in the development process. One of the apparent risks is that the selected component technology is not appropriate for its purpose; hence, the need to evaluate component technologies with respect to the requirements expressed by the software developers.

Paper outline: Section 2 describes the requirements which are used for evaluating the component technologies. Section 3 presents the selected component technologies, and a discussion of how well the technologies fit the requirements. Section 4 provides a summary of our evaluation, including a graded table showing how well each of the technologies matches the listed requirements. Finally, in section 5 we draw conclusions and present our future plans.

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). The reason for this classification is mainly to clarify that industrial actors are not only interested in technical solutions, but also in improvements regarding their development process. In addition, section 2.3 contains implied (or 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.

2.1 Technical Requirements

The technical requirements describe industrial needs and desires regarding the technically related 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, of systems build up from well-tested components is considered highly attractive. In fact, it is one of the single most distinguished requirements found in our investigation. Vehicle industry strives for better analyses of computer system behaviour in general. This striving naturally affects requirements placed on a component technology.

When analysing a system built from well-tested, functionally correct, components, the main issue is associated with composability. The composition process must guarantee non-functional properties, such as the communication, synchronisation, memory, and timing characteristics of the system [1].

When considering timing analysability, it is important to be able to verify (1) that each component meet its timing requirements, (2) that each node, built up from several components, meet its deadlines (i.e. schedulability analysis), and (3) to be able to analyse the end-to-end timing behaviour of functions in a distributed system. Since

the systems are resource constrained, it is important to be able to analyse the memory consumption. These checks should be done pre-runtime to avoid failures during runtime.

2.1.2 Testable and debugable

It is required that there exist tools that support debugging both at component level, e.g. a graphical debugging tool showing the component's in- and out-port values (if supported by the component model), as well as on the traditional source code debugging 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. In fact, the ability to test embedded-system software can be improved when using CBSE, since it adds the ability to test components in isolation. This is a desired functionality asked for by the industry.

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) RTOS independent and (3) communications protocol independent. The components are kept portable by minimising the number of dependencies to the software platform. 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.

Ideally, components should also be independent of the component framework used during run-time. This whish may be difficult to satisfy since traditionally a component model has been tightly integrated with its component framework. However, this kind of optimisation is important for companies cooperating with different customers, using different hardware and operating systems. Such an approach also enhances the ability to upgrade or update the hardware or the operating system.

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 can significantly 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 at run-time. However, this dynamic behaviour comes at the price of increased resource consumption.

2.1.5 Component Modelling

The component modelling should be based on a standard modelling language like UML [4] or UML 2.0 [8]. The main reason to choosing UML is that it is a well known and tested modelling technique 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 it is believed that the business segment does not have the possibility do develop their own standards and practices. Instead they preferably relay on the use of readily available and mature techniques.

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 believed to enhance reusability but most of all, simplicity, but without introducing any unnecessary resource consumption.

The computational model should be focused on a pipe-and-filter model [5]. 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 requirements for CBSE technologies, the research community often overlooks requirements related to the development process. For software developing companies however, these requirements are at least as important as the technical requirements. When talking to industry, earning money is the main focus. This cannot be done without having an efficient development processes deployed. To obtain industrial reliance, the development requirements need to be considered by the component technology and the tools associated with the technology. A change in development process is associated with major risks and costs. This fact implies that the development requirements are very essential and cannot be neglected.

2.2.1 Introducible

Appropriate support for companies to gradually migrate into a new development technology should be considered 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, it to allow legacy code within systems designed with the new technology. Often systems are developed in an incremental way, where new versions build on previous versions. To redevelop the whole system from scratch, just to adapt it to a new development technique is far too expensive.

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 [8]. 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 using configuration parameters to the components. The parameters, which are fixed at compile-time, should allow both configuration of behaviour and automatic reduction of run-time overhead and complexity.

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, meaning that 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 where the component is used. 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. Also, 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) in that an understandable technique is also 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 two implied requirements, i.e. 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. The reasons for this include that 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[2].

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 requirements (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

For a component model to better support the technical requirements of analysability (section 2.1.1), testability (section 2.1.2), and resource consumption (section 2.1.4), the component model should be configured pre-runtime, i.e. at compile time. Here a 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 [2][3]. This is of course appropriate in this specific domain where one usually has access to ample resources. Embedded systems, however, face another reality; that is, with resource constrained nodes running complex, dependable, control applications.

A motivation for the static configuration is that a typical embedded system does not interact directly with the user. A node is started, e.g., when the ignition key is turned on, and is running as a self-contained control unit until the key is turned off. Hence, there is no need to reconfigure the system during runtime. However, most vehicles

[2] 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.

can operate in different modes, hence the technology must support switches between a set 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 considered 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 [2] and Java Enterprise Beans [3]) since it is the only technology that explicitly address embedded and real-time issues. Also, the Windows CE version of .NET [2] is omitted, since it is targeted towards embedded display-devices, which only constitute a small subset of the devices in vehicular systems.

3.1 PECT

Prediction-Enabled Component Technology (PECT) [11] is a development infrastructure that incorporates development tools and analysis techniques. PECT is an ongoing research project at the Software Engineering Institute (SEI) at Carnegie Mellon University[3]. PECT focuses on analysis; in principle any analysis could be incorporated. However, the framework does not include any theories on how to analyse different properties, just definitions of how analysis shall be applied in a so called reasoning framework. To be able to analyse using PECT, proper analysis theories must be found and implemented and a suitable underlying component technology must be available. A PECT is an abstract model of a component technology, consisting of a construction framework and a reasoning framework. When concretizating 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. Although to move from one platform to another, the choice of the underlying technology may become important in practice. For modelling or describing a component based system the Construction and Composition Language (CCL) [11]; which is not UML-compliant, is used. 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 with PECT can be difficult to understand; mainly because of the mapping from the abstract component model to the concrete component technology. It is probably the case that systems that look identical at the PECT-level behave differently when realised on different component technologies.

PECT is an abstract technology that requires an existing underlying component technology. For instance, how testable and debugable a system is depends highly 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. Worth to notice is that although reusability and maintainability is not directly addressed by PECT, the analysability increases these abilities.

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

3.2 Koala

The Koala component technology [12] is designed and used by Philips[4] for development of software in consumer electronics. Typically, consumer electronics are resource constrained systems since they are using cheap hardware components to keep development costs low. Koala is a light weight component model, tailored for Product Line Architectures [24]. The Koala components can interact with the environment or other components through explicit interfaces only. The source code of Koala the components are fully visible for the developers, i.e. they are not binaries or any 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 [8]. 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 has passive components that interact through a pipes-and-filters model, which is allocated to active threads. 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 also has rules telling when a component can be changed to another component based on the provided interface and the produced result set.

Koala does not support analysis of run-time properties. Research has presented how properties like memory usage can be analysed, but the thread pumps used in Koala might cause some problems to apply existing timing analysis theories. 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 interaction to the operating system through Koala connectors. The modelling language is defined and developed in-house. Although it seams to be rather straight forward to map UML 2.0 component diagrams to Koala, using provided and required ports. It is difficult to see an easy way to gradually introduce the Koala concept, it is all or nothing.

3.3 Rubus Component Model

The Rubus Component Model [22] is developed by Arcticus systems[6]. The component model 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. Rubus has one time-triggered part, used for time-critical hard real-time systems, and one event-triggered part, used for less time-critical soft real-time systems. However, the Rubus Component Model is only supported by the time-triggered part.

The Rubus Component Model runs on top of the Rubus Operating System [23], 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 component model is too tightly coupled to the Rubus operating system. 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 since the component technology is statically configured, but timing analysis on component and node level (i.e. schedulability analysis) is the only analysable properties implemented in the Rubus tools. Testability is facilitated by static scheduling (which gives predictable execution patterns). Testing the functional behaviour of the source code can also be achieved using the Rubus Windows simulator, which can be used to run the systems on a regular PC.

---

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

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

Applications are described in the Rubus Design Language, which is an in-house non-standard language. The fundamental building blocks are passive. Components may or may not include execution threads. The interaction model is the required pipes-and-filters (section 2.1.6). The graphical representation of a system is quite intuitive, and the Rubus Component Model 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 a component level, but COTS debuggers can be used to debug the source code. The components are very simple, but the component technology does not include any supporting tools to enhance reusability. The components can be parameterised, which improves the possibility to change the component behaviour without changing the component source code. This enhances the possibilities to reuse the components. Component simplicity makes the systems fairly simple to maintain, however, no explicit support for maintainability exists.

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) [13] combines object oriented design, with port automaton theory. PBO was developed as a part of the Chimera RTOS project [18], at the Advanced Manipulators Laboratory at Carnegie Mellon University[9]. 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, and that admits easy unit testing. A PBO is however tightly coupled to the Chimera operating system and system calls are widely used. A PBO is an independent concurrent process, i.e. an active object.

Since PBO is coupled to the Chimera OS, it can not be easily introduced in any legacy system. The Chimera operating system is a large and dynamically configurable operating system, and since the objects in PBO are active and run as tasks and support dynamic binding etc. hence it can not be considered 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 objects makes it easy to modify or replace a single component. However, there is no explicit way of modelling objects; also no modelling language is suggested to be suitable or compatible. Although, PBO use port-automaton theory that provides an algebraic model of the system that is suitable for control systems. PBO is built with active and independent objects that are connected with the pipes-and-filters model. Due to the low coupling between objects 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.

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

3.5 PECOS

PECOS[11] (PErvasive COmponent Systems) [20][21] is a collaborative project between ABB Corporate Research Centre[12] and academia. The original motivation for the PECOS project was to expand the field-device market and the needs to develop systems more quickly and at lower costs. 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 a project where much focus has been put on non-functional properties such as memory consumption, timeliness etc. which makes PECOS analysable.

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 run-time.

The PECOS model 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 [25] to model concurrent activities like component compositions, scheduling of components, and synchronisation of shared ports [14]. 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 layered software architecture. The chosen approach is rather similar to the requirements of 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 systems are implemented using the cheapest possible hardware (e.g. 16-bit microprocessor, 256 kB ROM, 40 kB RAM [21]). These systems are considered small compared to the systems used in vehicular industries [10].

The PECOS component technology uses a modelling language that is easy to understand, however no standard language is used. Each component is responsible for a single piece of functionality, executed repeatedly in a cyclic behaviour. The components communicate using a data-flow-oriented interaction, it is a pipe-and-filter 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. One of the motivating factors for starting the PECOS project was to enhance maintainability of the software systems. The PECOS component technologies support for maintainability is enhanced by using a development and testing toolkit including editors, code browsers, and code generators supports target development. Schedule computation and testing is also supported. Most of these tools are integrated in the PECOS component environment.

There are two types of components, leaf components (black-box components) and composite components. These components can be passive (explicitly scheduled by its nearest ancestor), active (have their own thread of execution), and event components (triggered by an event). 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 [19]. The devices are statically configured.

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

[12] 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 standard that provides a set of rules for writing platform independent applications. The CORBA standard is developed by the Object Management Group[15] (OMG). A major drawback with CORBA is that it requires a lot of functionality in order to connect diverse platforms within a heterogonous system. Because of this, variants of CORBA exist, two major are MinimumCORBA [15] for resource constrains systems, and RT-CORBA [17] for time-critical systems.

RT-CORBA is a set of extensions to CORBA tailored to equip Object Request Brokers (ORBs) to be used as a component of a real-time system. Features supported by RT-CORBA are explicit thread pools and queuing control. RT-CORBA builds on controlling the use of processor, memory and network resources. Since RT-CORBA adds complexity to the standard CORBA it is not considered very useful for vehicular systems. MinimumCORBA defines a subset of the CORBA functionality that is more suitable for resource-constrained systems, where e.g. some or the dynamics is reduced.

OMG has defined a CORBA Component Model (CCM) [16], 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 a 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 and e.g. bindings are performed during run-time. Because of the run-time decisions, CORBA is not very deterministic or analysable. There is no explicit modelling language for CORBA. CORBA uses a client server model for communication, where each object is active. There are no extra-functional properties or any specification of interface behaviour. A CCM component can have strong, undocumented, assumptions on its use and context. All these things together make reuse virtually impossible in embedded systems. 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 in section 3, grading how well they fulfil each of the requirements of section 2. The grade is based on the account for each technology given in section 3 and the papers references in the respective subsection. We use a simple 3 level grade, where 0 means that the requirement is not addressed by the technology and is 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 chosen, for these requirements we do not assign a grade (indicated with NA, Not

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

Applicable, in figure 1). For the CORBA based technologies we have listed the best grade applicable to any of the 3 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. Since these requirements should be quite easy to satisfy we can only draw the conclusion that component-technology designers have not paid any attention to these requirements. We also note that no requirements have a very high grade (above 1.5). This indicate that none of the requirement we have listed 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, this average grade should be taken with a grain of salt; e.g., the average cannot be directly used to rank technologies among 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. Comparing for example PECOS and CORBA is not fare, since they are aiming for different types of systems. But the evaluation of each of the component technologies based on the requirements in section 2 is appropriate.

	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	1	0	0	0	2	0	0	1	0	0	0.4	1	8
Average	1.2	1.0	1.0	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 industrial requirements*

5 Conclusion

In this paper we have compared some existing component technologies for embedded systems with respect to real industrial requirements. The requirements have been collected from industrial actors within the business segment of heavy vehicles [10]. 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 the conclusion is that there is no component technology available that fulfil all the requirements listed in section 2. 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 on of the technologies and try to make as small changes as possible to that technology.

6 References

[1] I. Crnkovic, M. Larsson, Building Reliable Component-Based Software Systems, 2002, ISBN 1-58053-327-2

[2] COM/DCOM/.NET by Microsoft; Home Page: http://www.microsoft.com

[3] Enterprise Java Beans by Sun; Home Page: http://www.java.sun.com

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

[5] M Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall 1996

[6] T. Nolte, A. Möller, M. Nolin; Using Components to Facilitate Stochastic Schedulability Analysis, In Proceedings of the WiP Session of the 24th IEEE Real-Time System Symposium, Cancun, Mexico, December, 2003

[7] Safety Integrity Levels - Does Reality Meet Theory?, Report of seminar held at the IEE, London, on 9 April 2002.

[8] UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, http://www.omg.com/uml; 2003

[9] D. Garlan, R. Allen, J Ockerbloom; Architectural Mismatch or Why it's hard to build systems out of existing parts; Proceedings of the Seventeenth International Conference on Software Engineering, Seattle WA, April 1995

[10] A. Möller, J. Fröberg, M. Nolin; Industrial Requirements on Component Technologies for Embedded Systems; Submitted for publication, available as Technical Report: MRTC report ISSN 1404-3041 ISRN MDH-MRTC-150/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, January, 2004

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

[12] R. van Ommering, F. van der Linden, and J. Kramer; The Koala component model for consumer electronics software. IEEE Computer, 33(3):78–85, March 2000.

[13] D. B. Stewart, R. A. Volpe, P. K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, IEEE Transactions on Software Engineering, December 1997, pages 759-776.

[14] O. Nierstrass, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Seidler, T. Genssler, R. van den Born, A Component Model for Field Devices; Proceedings of the First International IFIP/ACM Working Conference on Component Deployment, Germany, June 2002.

[15] Object Management Group. Minimum CORBA 1.0, http://www.omg.org/technology/documents/formal/-minimum_CORBA.htm OMG,

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

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

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

[19] R. Wuyts, S. Ducasse. Non-Functional Requirements in a Component Model for Embedded Systems, In International Workshop on Specification and Verification of Component-Based Systems, OOPSLA 2001

[20] M Winter, T Genssler, et al.; Components for Embedded Software – The Pecos Approach; Second International Workshop on Composition Languages, In conjunction with 16th European Conference on Object-Oriented Programming (ECOOP) Málaga, Spain, June 11, 2002

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

[22] K-L. Lundbäck, J. Lundbäck, M. Lindberg; Component based development of dependable real-time applications, Arcticus Systems, Home Page: http://www.arcticus.se

[23] Rubus OS Reference Manual, General Concepts, Arcticus Systems, Home Page: http://www.arcticus.se

[24] Clements, P., Northrop, L., Software Product Lines, Addison-Wesley, August 2001, ISBN 0-201-70332-7

[25] M. Sgroi. Quasi-Static Scheduling of Embedded Software Using Free-Choice Petri Nets (M.S. Dissertation), University of California at Berkeley, May 1998