Industrial Requirements on Component Technologies for Vehicular Control-Systems

Anders Möller[†][⋆] Mikael Åkerholm[†][⋆] Joakim Fröberg[†][‡]
Johan Fredriksson[†] Mikael Nolin[†][⋆]

[†]MRTC, Mälardalen University, Sweden
[⋆]CC Systems, Uppsala, Sweden
[‡]Volvo Construction Equipment, Eskilstuna, Sweden

February 10, 2006

Abstract

Software component technologies for automotive applications are desired due to the envisioned benefits in reuse, variant handling, and porting; thus, facilitating both efficient development and increased quality of software products. Component based software development has had success in the PC application domain, but requirements are different in the embedded domain and existing technologies does not match. Hence, software component technologies have not yet been generally accepted by embedded-systems industries.

In order to better understand why this is the case, we present two separate case-studies together with an evaluation of the existing component technologies suitable for embedded control systems.

The first case-study presents a set of requirements based on industrial needs, which are deemed decisive for introducing a component technology. Furthermore, in the second study, we asked the companies involved to grade these requirements.

Then, we use these requirements to compare existing component technologies suitable for embedded 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 can be used to guide modifications and/or extensions to existing component technologies in order to make them better suited for industrial deployment in the automotive domain. The results can also serve to guide other software engineering research by showing the most desired areas within component-based software engineering.

1 Introduction

During the last decade, Component-Based Software Engineering (CBSE) for embedded systems has received a large amount of attention. For office/Internet applications, CBSE has had tremendous impact [1, 2, 3], and today components are downloaded and on the fly integrated into, e.g., word processors and web browsers. However, in the embedded systems industry CBSE is still to a large extent envisioned as a promising future technology to meet specific demands on improved quality and lowered cost, by facilitating software reuse, efficient software development, enhanced system maintainability, and more reliable software systems [4].

CBSE has not yet been generally accepted by embedded-system developers. They are in fact, to a large extent, still using monolithic and platform dependent software development techniques, in spite of the fact that this make software systems difficult to maintain, upgrade, and modify. A major reason to not change to more modern techniques is to avoid the additional overhead with respect to, e.g., memory consumption and processor demands that new commercial technologies seem to introduce. A second reason is to not renounce relaibility and robustness aspects using, e.g., polymorphism and dynamic linking. Finally, there are also significant risks and costs associated with the adoption of a new development technique, that these companies may not be willing to take without guarantees.

The contributions of this article are threefold. First, it straightens out some question-marks regarding actual industrial requirements placed on a component technology. Second, we have asked industry to rank these requirements in order be able to focus on the most important aspects of component based development. This grading can be used to guide the research community when focusing on areas with the highest potential industrial impact. Third, we have used the ranked requirements to evaluate a set of available component technologies (from academia as well as from industry) that can be used to minimise the risk when introducing a new development process. Thus, this study can help companies to take the step into tomorrow's technology today. The list can also be used to guide modifications and/or extensions to existing component technologies, in order to make them better suited for industrial deployment. Our list of requirements also illustrates how industrial requirements on ! products and product development impact requirements on a component technology.

This article summarises our work on industrial requirements [5, 6, 7], and extends previous work, studying the requirements for component technologies, in that the results are not only based on our experience, or experience from a single company [8, 9]. We base most of our results on interviews with senior technical staff at the two companies involved in this article, but we have also conducted interviews with technical staff at other companies. Furthermore, since the embedded systems market is so diversified, we have limited our study to applications for distributed embedded real-time control in safety-critical environments, specifically studying companies within the heavy vehicles market segment [10, 11]. This gives our results higher validity, for this class of applications, than do more general studies of requirements in the embedded systems market [12].

2 Introducing CBSE in the Vehicular Industry

Component-based software engineering arouses interest and curiosity in industry. This is mainly due to the enhanced development process and the improved ability to reuse software offered. Also, the increased possibility to predict the time needed to complete a software development project, due to the fact that the assignments can be divided into smaller and more easily defined tasks, is seen as a driver for CBSE.

CBSE can be approached from two, conceptually different, points of view; distinguished by whether the components are (1) used as a design philosophy independent from any concern for reusing existing components, or (2) seen as reusable off-the-shelf building blocks used to design and implement a component-based system [13]. When talking to industrial software developers with experience from using a CBSE development process [14], such as Volvo Construction Equipment[1], the first part, (1), is often seen as the most important advantage. Their experience is that the design philosophy of CBSE gives rise to good software architecture and significantly enhanced ability to divide the software development in small, clearly-defined, sub-projects. This, in turn, gives predictable development times and shortens the time-to-market. The second part, (2), are by these companies often seen as less important, and the main reason for this is that experience shows that most approaches to large scale software reuse is associated with major risks and high initial costs. Rather few companies are willing to take these initial costs and risks since it is difficult to guarantee that money is saved in the end.

On the other hand, when talking to companies with less, or no, experience from component-based technologies, (2) is seen as the most important motivation to consider CBSE. This discrepancy between companies with and without CBSE experience is striking.

However, changing the software development process to using CBSE does not only have advantages. Especially in the short term perspective, introducing CBSE represents significant costs and risks. For instance, designing software to allow reuse requires (sometimes significantly) higher effort than does designing for a single application [15]. According to certain experience it takes even three times longer to develop a general reusable component than achieving the same functionailty targetting a specific case [16]. For resource constrained systems, design for reuse is even more challenging, since what are the most critical resources may vary from system to system (e.g. memory or CPU-load). Furthermore, a component designed for reuse may exhibit an overly rich interface and an associated overly complex and resource consuming implementation. Hence, designing for reuse in resource constrained environments requires significant knowledge not only about functional requirements, but also about extra-

[1]Volvo Construction Equipment, Home Page: http://www.volvo.com

functional requirements. These problems may limit the possibilities of reuse, even when using CBSE.

Within software engineering, having a clear and complete understanding of the software requirements is paramount. However, practice shows that a major source of software errors comes from erroneous, or incomplete, specifications [15]. Often incomplete specifications are compensated for by engineers having good domain knowledge, hence having knowledge of implicit requirements. However, when using a CBSE approach, one driving idea is that each component should be fully specified and understandable by its interface and associated documentation. Hence, the use of implicit domain knowledge not documented in the interface may hinder reuse of components. Also, division of labour into smaller projects focusing on single components, require good specifications of what interfaces to implement and any constraints on how that implementation is done, further disabling use of implicit domain knowledge. Hence, to fully utilise the benefits of CBSE, a software engineering process that do not rely on engineers' implicit domain knowledge need to be established.

Also, when introducing reuse of components across multiple products and/or product families, issues about component management arise. In essence, each component has its own product life-cycle that needs to be managed. This includes version and variant management, keeping track of which versions and variants is used in what products, and how component modifications should be propagated to different version and variants. Components need to be maintained, as other products, during their life cycle. This maintenance needs to be done in a controlled fashion, in order not to interfere aversively with ongoing projects using the components. This can only be achieved using adequate tools and processes for version and variant management, to fully support a component-based strategy such tools should support version management for components instead of traditional files, and also allow the use of different versions of a component to the same client (e.g., to allow a single product to use a number of diffent versions of a component).

3 A Component Technology for Heavy Vehicles

Existing component technologies [1, 2, 3] are in general not applicable to embedded computer systems, since they do not consider aspects such as safety, timing, and memory consumption that are crucial for many embedded systems [1, 2]. Some attempts have been made to adapt component technologies to embedded systems, like, e.g., MinimumCORBA [17]. However, these adaptations have not been generally accepted in the embedded system segments. The reason for this is mainly due to the diversified nature of the embedded systems domain. Different market segments have different requirements on a component technology, and often, these requirements are not fulfilled simply by stripping down existing component technologies; e.g. MinimumCORBA requires less memory then does CORBA, however, the need to statically predict memory usage is not addressed.

It is important to keep in mind that the embedded systems market is extremely diversified in terms of requirements placed on the software. For instance, it is obvious that software requirements for consumer products, telecom switches, and avionics are quite different. Hence, we will focus on one single market segment: the segment of heavy vehicles, including, e.g., wheel loaders and forest harvesters. It is important to realise that the development and evaluation of a component technology is substantially simplified by focusing on a specific market segment. Within this market segment, the conditions for software development should be similar enough to allow a lightweight and efficient component technology to be established.

3.1 The Business Segment of Heavy Vehicles

Developers of heavy vehicles faces a situation of (1) high demands on reliability and performance, (2) requirements on low product cost, and (3) supporting many configurations, variants and suppliers. Computers offer the performance needed for the functions requested in a modern vehicle, but at the same time vehicle reliability must not suffer. Computers and software add new sources of failures and, unfortunately, computer engineering is less mature than many other fields in vehicle development and can cause lessened product reliability. This yields a strong focus on the ability to model, predict, and verify computer functionality.

At the same time, the product cost for volume products must be kept low. Thus, there is a need to include a minimum of hardware resources in a product (only as much resources as the software really needs). The stringent cost requirements also drive vehicle developers to integrate low cost components from suppliers rather than develop in-house. On top of these demands on reliability and low cost, vehicle manufacturers make frequent use of product variants to satisfy larger groups of customers and thereby increase market share and product volume.

In order to accommodate (1)-(3), as well as an increasing number of features and functions, the electronic system of a modern vehicle is a complex construction which comprise electronic and software components from many vendors and that exists in numerous configurations and variants.

The situation described cause challenges with respect to verification and maintenance of these variants, and integration of components into a system. Using software components, and a CBSE approach, is seen as a promising way to address challenges in product development, including integration, flexible configuration, as well as good reliability predictions, scalability, software reuse, and fast development. Further, the concept of components is widely used in the vehicular industry today. Using components in software would be an extension of the industry's current procedures, where the products today are associated with the components that constitute the particular vehicle configuration.

What distinguishes the segment of heavy vehicles in the automotive industry is that the product volumes are typically lower than that of, e.g., trucks or passenger cars [10]. Also the customers tend to be more demanding with respect to technical specifications such as engine torque, payload etc, and less demanding with respect to style. This causes a lower emphasis on product cost and optimisation of hardware than in the automotive industry in general. The lower volumes also make the manufacturers more willing to design variants to meet the requests of a small number of customers.

3.2 System Description

In order to describe the context for software components in the vehicular industry, we will first explore some central concepts in vehicle electronic systems. Here, we outline some common and typical solutions and principles used in the design of vehicle electronics. The purpose is to describe commonly used solutions, and outline the de facto context for application development and thereby also requirements for software component technologies.

The system architecture can be described as a set of computer nodes called Electronic Control Units (ECUs). These nodes are distributed throughout the vehicle to reduce cabling, and to provide local control over sensors and actuators. The nodes are interconnected by one or more communication busses forming the network architecture of the vehicle. When several different organisations are developing ECUs, the bus often acts as the interface between nodes, and hence also between the organisations. The communication bus is typically low cost and low bandwidth, such as the Controller Area Network (CAN) [18].

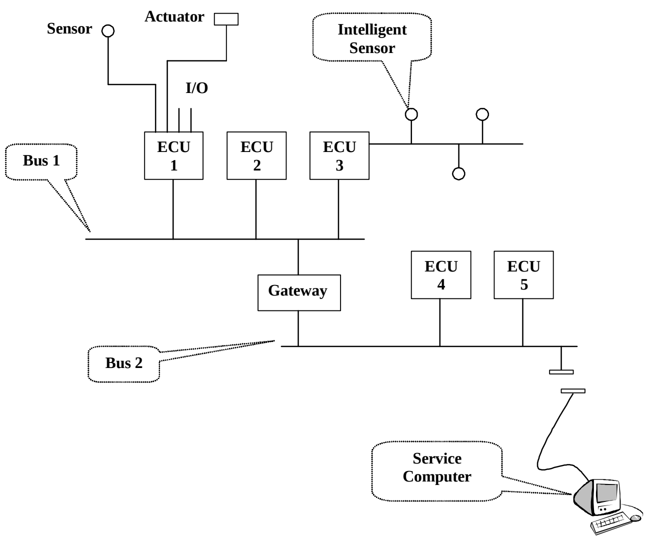
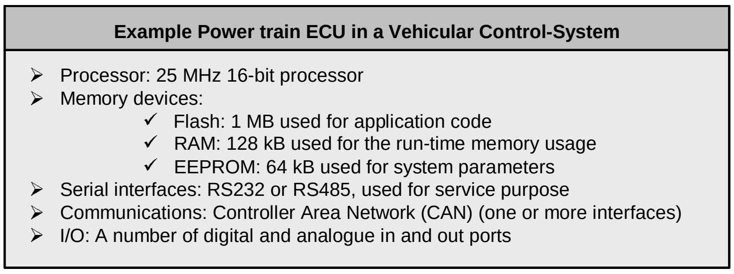


Figure 1: Example of a vehicle network architecture

In the example shown in Fig. 1, the two communication busses are separated using a gateway. This is a common architectural pattern that are used for several reasons, e.g., separation of criticality, increased total communication bandwidth, fault tolerance, compatibility with standard protocols [19, 20, 21], etc. Also, safety critical functions may require a high level of verification, which is usually very costly. Thus, non-safety related functions might be separated to reduce cost and effort of verification. In some systems the network is required to give synchronisation and provide fault tolerance mechanisms.

The hardware resources are typically scarce due to the requirements on low product cost. Addition of new hardware resources will always be defensive, even if customers are expected to embrace a certain new function. Because of the uncertainty of such expectations, manufacturers have difficulties in estimating the customer value of new functions and thus the general approach is to keep resources at a minimum.



Example Power train ECU in a Vehicular Control-System

- Processor: 25 MHz 16-bit processor
- Memory devices:
    - ✓ Flash: 1 MB used for application code
    - ✓ RAM: 128 kB used for the run-time memory usage
    - ✓ EEPROM: 64 kB used for system parameters
- Serial interfaces: RS232 or RS485, used for service purpose
- Communications: Controller Area Network (CAN) (one or more interfaces)
- I/O: A number of digital and analogue in and out ports

Figure 2: Specification of an embedded system ECU

In order to exemplify the settings in which software components are considered, we have studied our industrial partner's currently used nodes. In Figure 2 we list the hardware resources of a typical ECU with requirements on sensing and actuating, and with a relatively high computational capacity (this example is from a typical power train ECU).

Also, included in a vehicle's electronic system can be display computer(s) with varying amounts of resources depending on product requirements. There may also be PC-based ECU's for non-control applications such as telematics, and information systems. Furthermore, in contrast to these resource intense ECU's, there typically exists a number of small and lightweight nodes, such as, intelligent sensors (i.e. processor equipped, bus enabled, sensors).

Figure 3 on the following page depicts the typical software architecture of an ECU. Current practice typically builds on top of a reusable "software platform", which consists of a hardware abstraction layer with device drivers and other platform dependent code, a Real-Time Operating System (RTOS), one or more communication protocols, and possibly a software (component) framework that is typically company (or project) specific. This software platform is accessible to application programmers through an Application Programmers Interface (API). Different nodes, presenting the same API, can have different realisation of the different parts in the software platform (e.g. using different RTOSs).

Today it is common to treat parts of the software platform as components, e.g. the RTOS, device drivers, etc, in the same way as the ECU's bus connectors

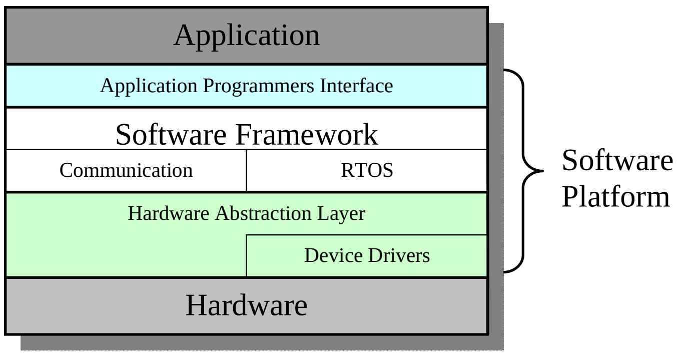


Figure 3: Internals of an ECU - A software platform

and other hardware modules. That is, some form of component management process exists; trying to keep track of which version, variant, and configuration of a component is used within a product. This component-based view of the software platform is however not to be confused with the concept of CBSE since the components does not conform to standard interfaces or component models.

4 Component Technology Requirements

There are many different aspects and methods to consider when looking into questions regarding how to capture the most important requirements on a component technology specially focusing on heavy vehicles. Our approach has been to cooperate with our industrial partners (CC Systems and Volvo Construction Equipment) very closely, both by performing interviews and by participating in software development projects. In doing so, we have extracted the most important requirements on a component-based technique from the developers of heavy vehicles point of view. The results from this study was first presented in [5].

The requirements are divided into two main groups, the technical requirements (Sect. 4.2) and the development process related requirements (Sect. 4.3). Also, in Sect. 4.4 we present some implied (or derived) requirements, i.e. requirements that we have synthesised from the requirements in sections 4.2 and 4.3, but that are not explicit requirements from industry.

4.1 Research Method

The goal of this study was to extract all challenges of relevance when introducing a component technology, and find the most important requirements. It seems natural to seek answers where the requirements are defined, i.e. at the automotive software developing organisations. Secondly, the answers are likely qualitative with a context full of details from development setting, products, organisation etc. These two facts led us to perform a case study [22] for the two cases represented by two developing organizations.

According to [22] a case study is an empirical inquiry that investigates a contemporary phenomenon in its real life context and copes with situations where there are more variables of interest than data points. In this study the phenomenon is the reluctance to adopt a component technology in automotive development and thereby the requirements put on such a technology. It is clearly a contemporary phenomenon and the situation in a development organisation comprises many variables with no hope of sampling enough data points to map relations.

The case-study was performed at Volvo Construction Equipment and at CC Systems. The respondents were senior technical staff from different parts of the organisation, like project managers, development process specialists, programmers, and testing specialists. The case-study protocol questions were open ended to encourage respondents to report on any issues they might attribute to component technologies.

4.2 Technical Requirements

The technical requirements describe the needs and desires that our industrial partners have regarding the technically related aspects and properties of a component technology.

4.2.1 Analysable

The vehicular industry strives for better analyses of computer system behaviour in general. This striving naturally affects requirements placed on a component model. System analysis, with respect to extra-functional properties, such as the timing behaviour and the memory consumption, of a system built up from well-tested components is considered attractive.

When analysing a system, built from well-tested and functionally correct components, the main issue is associated with composability. The composability problem must guarantee extra-functional properties, such as the communication, synchronisation, memory, and timing characteristics of the system [4].

When considering, e.g., timing analysability, it is important to be able to verify (1) that each component meet its timing requirements, (2) that each node (which is 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 distributed functions (e.g. distributed over several nodes in a distributed control system).

Because of the fact that the systems are resource constrained (Sect. 3), it is important to be able to analyse the memory consumption. To check the sufficiency of the application memory, as well as the paramater memory (typcially EEPROM), is important. This check should be done pre-runtime to avoid failures during runtime.

4.2.2 Testable and debuggable

Industry requires tools that support functional debugging, both at component level (e.g. a graphical debugging tool showing the components in- and out-port values) and at the traditional white-box source code level. The test and debug environment needs to be "component aware" in the sense that port-values can be monitored and traced and that breakpoints can be set on component level.

Testing and debugging is by far the most commonly used technique to verify software systems functionality. Testing is a very important complement to analysis, and it should not be compromised when introducing a component technology.

In fact, the ability to test embedded-system software can be improved when using CBSE. This is possible because the component functionality can be tested in isolation. This is a desired functionality asked for by our industrial partners. This test should be used before the system tests, and this approach can help finding functional errors and source code bugs at the earliest possible opportunity.

4.2.3 Portable

The components, and the infrastructure surrounding them, should be platform independent to the highest degree possible. Here, platform independent means hardware independent, RTOS independent and communication protocol independent.

Components are kept portable by minimising the number of dependencies to the supporting software platform. Such dependencies are off course, to some extent, necessary in order to construct an executable system. However, the dependencies should be kept to an absolute 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 may seem far fetched, since traditionally a component model has been tightly integrated with its component framework. However, support for migrating components between component frameworks is important for companies cooperating with different customers, using different hardware and operating systems.

4.2.4 Resource Constrained

The components should be small and light-weighted and the components infrastructure and framework should be minimised. Ideally, there should no run-time overhead compared to not using a component based approach.

Embedded vehicular systems are typically resource constrained in order to lower the production costs. When companies design new ECUs, future profit is the main concern. Therefore the hardware is dimensioned for anticipated use but not more.

One possibility, that can reduce resource consumption of components and the component framework significantly, is to limit the possible run-time dynamics.

This means that it is desirable to allow only 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.

4.2.5 Component Modelling

A component technology should be based on a standard modelling language like UML [23] or UML 2.0 [24]. The main reason for choosing UML is that it is a well known and thoroughly tested modelling technique with tools and formats supported by third-party developers.

The reason for our industrial partners to have specific demands in these details, is that it is belived that the business segment of heavy vehicles does not have the possibility do develop their own standards and practices. Instead they preferably relay on the use of simple and mature techniques supported by a welth of third party suppliers.

4.2.6 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, and to limit resource consumption. The computational model should be focused on a pipe-and-filter model [25]. 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.

4.3 Development Requirements

When discussing CBSE requirements, 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. However, this cannot be done without having an efficient development processes deployed. Hence – to obtain industrial reliance, the development requirements need to be considered and addressed by the component technology.

4.3.1 Introducible

It should be possible for companies to gradually migrate into a new development technology. It is important to make the change in technology as safe and inexpensive as possible.

Revolutionary changes in the development technique used at a company are associated with high risks and costs. Therefore a new technology should be possible to divide into smaller parts, which can be introduced separately. For instance, if the architecture described in Fig. 3 is used, the components can be used for application development only and independently of the real-time operating system. Or, the infrastructure can be developed using components, while the application is still monolithic.

One way of introducing a component technology in industry, is to start focusing on the development process related requirements. When the developers have accepted the CBSE way of thinking, i.e. thinking in terms of reusable software units, it is time to look at available component technologies. This approach should minimise the risk of spending too much money in an initial phase, when switching to a component technology without having the CBSE way of thinking.

4.3.2 Reusable

Components should be reusable, e.g., for use in new applications or environments than those for which they where originally designed [26]. The requirement of reusability can be considered both a technical and a development process related requirement. Development process related since it has to deal with aspects like version and variant management, initial risks and cost when building up a component repository, etc. Technical since it is related to aspects such as, how to design the components with respect to the RTOS and HW communication, etc.

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 simplified further by using input parameters to the components. Parameters that are fixed at compile-time, should allow automatic reduction of run-time overhead and complexity.

A clear, explicit, and well-defined component interface is crucial to enhance the software reusability. To be able to replace one component in the software system, a minimal amount of time should be spent trying to understand the component that should be interchanged.

It is, however, both complex and expensive to build reusable components for use in distributed embedded real-time systems [4]. The reason for this is that the components must work together to meet the temporal requirements, the components must be light-weighted since the systems are resource constrained, the functional errors and bugs must not lead to erroneous outputs that follow the signal flow and propagate to other components and in the end cause unsafe systems. Hence, reuse must be introduced gradually and with grate care.

4.3.3 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.

In essence, this requirement is a product of the previous requirement on reusability. The flip-side of reusability is that the ability to reuse and reconfigure the components using parameters leads to an abundance of different configurations used in different vehicles. The same type of vehicle may use different software settings and even different component or software versions. So, by introducing reuse we introduce more administrative work.

Reusing software components lead to a completely new level of software management. The components need to be stored in a repository where different versions and variants need to be managed in a sufficient way. Experiences from trying to reuse software components show that reuse is very hard and initially related with high risks and large overheads [4]. These types of costs are usually not very attractive in industry.

The maintainability requirement also includes sufficient tools supporting the service of the delivered vehicles. These tools need to be component aware and handle error diagnostics from components and support for updating software components.

4.3.4 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.

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. It is widely known that many software errors occur in code that deals with synchronisation, buffer management and communications. However, when using component technologies such code can, and should, be automatically generated; leaving application engineers to deal with application functionality.

4.4 Derived Requirements

Here, we present two implied requirements, i.e. requirements that we have synthesised from the requirements in sections 4.2 and 4.3, but that are not explicit requirements from the vehicular industry.

4.4.1 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 (Sect. 4.2.2). Since source code debugging is demanded, even if a component technology is used, black box components is undesirable.

Using black-box components would, regarding to our industrial partners, lead to a feeling of not having control over the system behaviour. However, the possibility to look into the components does not necessary mean that you are allowed to modify them. In that sense, a glass-box component model is sufficient.

Source code components also leaves room for compile-time optimisations of components, e.g., stripping away functionality of a component that is not used in a particular application. Hence, souce code components will contribute to lower resource consumption (Sect. 4.2.4).

4.4.2 Static Configuration

For a component model to better support the technical requirements of analysability (Sect. 4.2.1), testability (Sect. 4.2.2), and light-weightiness (Sect. 4.2.4), the component model should be configured pre-runtime, i.e. at compile time. Component technologies for use in the office/Internet domain usually focus on a dynamic behaviour [1, 2]. This is of course appropriate in this specific domain, where powerful computers are used. Embedded systems, however, face another reality - with resource constrained ECU's running complex, dependable, control applications. Static configuration should also improve the development process related requirement of understandability (Sect. 4.3.4), since there will be no complex run-time reconfigurations.

Another reason for the static configuration is that a typical control node, e.g. a power train node, does not interact directly with the user at any time. The node is started when the ignition key is turned on, and is running as a self-contained control unit until the vehicle is turned off. Hence, there is no need to reconfigure the system during runtime.

4.5 Discussion

Reusability is perhaps the most obvious reason to introduce a component technology for a company developing embedded real-time control systems. This matter has been the most thoroughly discussed subject during our interviews. However, it has also been the most separating one, since it is related to the question of deciding if money should be invested in building up a repository of reusable components.

Two important requirements that have emerged during the discussions with our industrial partners are safety and reliability. These two are, as we see it, not only associated with the component technology. Instead, the responsibility of designing safe and reliable system rests mainly on the system developer. The technology and the development process should, however, give good support for designing safe and reliable systems.

Another part that has emerged during our study is the need for a quality rating of the components depending on their success when used in target systems.

This requirement can, e.g., be satisfied using Execution Time Profiles (ETP's), discussed in [27]. By using ETPs to represent the timing behaviour of software components, tools for stochastic schedulability analysis can be used to make cost-reliability trade offs by dimensioning the resources in a cost efficient way to achieve the reliability goals. There are also emerging requirements regarding the possibilities to grade the components depending on their software quality, using for example different SIL (Safety Integrity Levels) [28] levels.

5 Requirements Grading

In order to better understand which of the requirements that is of most importance to industry we conducted a second study [7]. The motivation of grading requirements is that the results can be used to guide researchers and tool vendors to put focus on the most relevant industrial requirements, and to resolve conflicts between requirements.

5.1 Method

The first case study identified many areas of interests and many were closely related to the development process. Open ended discussions gave us the elicitation of the most important requirements but no notion of relative importance can be analysed based on these results. In order to grade requirements according to importance we performed a second study.

The requirement grading was performed in a workshop with a short presentation, definition of terms, questions and a numerical grading of requirements where the average sum was bounded. Thus, respondents could not grade all requirements high in order to get a sum average in the predefined range. The procedure was the following:

1. The workshop started with a short presentation of the study and of component technologies basics. A very brief background was presented with PC software benefits while automotive software engineers are still reluctant. Furthermore the development process of working with components in a component repository rather than developing in a normal V model was described. The terms; Tool, Components, Platform, Component Framework and Repository was explained. Finally the results from the earlier study were presented.

2. Secondly, the definitions of all the requirements that were to be graded were presented and respondents were given handouts with the definitions. Respondents were allowed to ask questions on the definitions.

3. The data collection was made by the respondents filling in a spreadsheet form on a laptop computer where all the twelve listed requirements were to be graded with a number 1-4 indicating from "interesting" to "absolutely decisive". The respondents were to make sure that the sum average of all

their grades was in the range 2.4 - 2.6. The sum, average of grades, was shown and recalculated throughout the grading.

5.2 Results

In this section we present the results (see Figure 1) from the second study, i.e. the industry grading of the requirements in section 4. We present the result by first discussing the requirements separately, and then in section 5.1 we draw same general conclusions from our work.

5.2.1 Analysable

Analysability is in general considered to be important, but the results from our case-study expose that it is not amongst the most important issues of component-based development. For example, it is worth noticing that our partners consider testability and the means to debug the application as much more important. Reasons for this might be that the business segment of heavy vehicles has low series (compared to, e.g., trucks or passenger cars) and that is cheaper to add extra processing power (faster CPU and more memory) in order to avoid timing or memory problems. It may be that a common view amongst industrial developers that analysability is complex and that it leads to a lot of manual information managing. Perhaps timing and memory consumption is not a problem in today's applications whereas testability gives direct feedback to the software developer and might hence be seen as more important. Yet another reason might be that analysability is not believed to be feasible or practical for distributed and complex industrial systems.

5.2.2 Test and Debug

Test and debug is the most important quality attribute seen in the requirement grades (see figure 1). This is most likely due to the fact that testing of embedded systems is extremely time consuming today. Hence, from a company perspective - there is a huge amount of time (and money) to save if a component technology could decrease the time it takes to verify software functionality.

Another important issue is the rising requirement from Original Equipment Manufacturers (OEMs) that sub-contractors deliver "error-free" software. Late or erroneous deliveries are typically punished by an OEM fine. This entail that testing of software (typically not complete systems but rather components) of the system gets more and more important.

It is also worth noticing that both CCS and VCE have spent huge amounts of money on developing test and debug equipment for their respective systems. Hence, the results might be a bit biased, i.e., that these companies consider it more important than the typical embedded software developer.

5.2.3 Portability

Portability is considered very important, mainly due to the fact that it is desired to keep hardware upgrading costs to an absolute minimum. But it is of course also important to be flexible in the choice of software platform.

For CCS, working with many different OEMs (and many different platforms), the requirements of portability is obvious - but it is striking to see that also VCE consider portability as being very important (see Figure 2). The reason for this is essentially that it is very important not to be too dependent on tool vendors and hardware platforms.

5.2.4 Resource Constrained

Surprisingly, and in quite contrary to what one could expect from developers of resource constrained embedded systems, this requirements is considered to be the least important in this study. The reason for this might be the fact that current state-of-practise development methods used by the vehicular industry are rather resource constrained. Hence, there is not much focus on this requirement in the daily work. It might be the case that developers take things they have for granted, and see things they do not have.

Another reason is Moore's law, it is cheaper to by more processing power than it is to spend money on analysing timing and memory consumption. This is also dependent on the product volumes, for low series products it might be worth spending some extra money on hardware in order to facilitate the use of more advanced development methods.

5.2.5 Component Modelling

This requirement is not considered to be very important; meaning that other aspects of modelling is more important than using business standards. For example, simplicity is more important than using a standard modelling language. However, it is interesting to notice that the requirement on using a standardised modelling language is more important relative to the requirement on resource usage.

5.2.6 Computational Model

The requirement on the computational model, meaning that the components should be passive (not having their own threads of execution) and that pipe-and-filter should be used as an architectural pattern, is the most deviating requirement (see Figure 2). This might be because VCE is currently using the Rubus Component Model [29] using a pipe-and-filter architecture, whilst CCS use different architectural patterns in different applications.

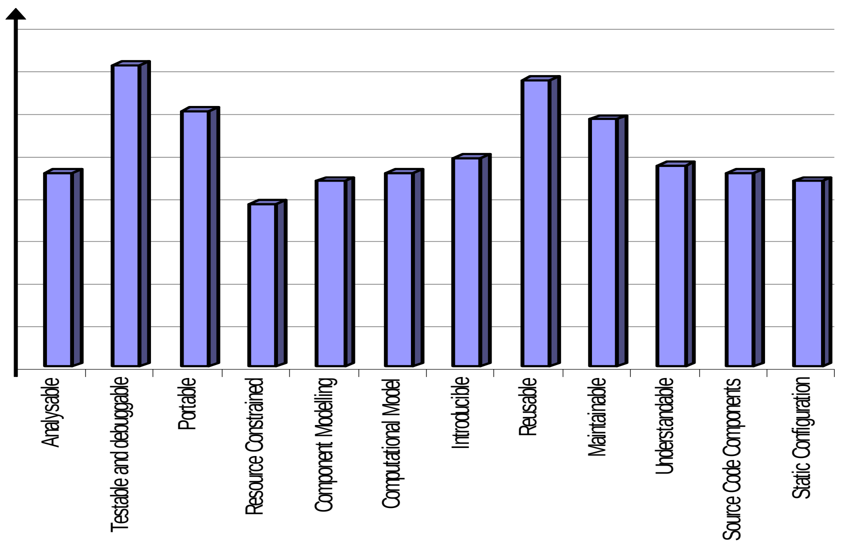


Figure 4: Requirements grades

5.2.7 Introducible

It is considered relatively important that the component technology is easy to introduce in new and existing projects/products. This requirement also includes the possibility to use parts of a component technology, e.g., together with various operating systems depending on customer needs.

One would expect to see a certain difference between a sub-contractor and an OEM - but as can be seen in Figure 2 both companies agree on the relative importance of this requirement.

5.2.8 Reusable

It is very interesting to see that reusability which is one of the fundamental reasons for moving towards CBSE is considered to be the second most important overall requirement. The reason for this is likely the large potential of software reuse in terms of development time and cost.

Reusability is typically considered to be very demanding, so it is worth noticing that the companies are willing to spend the extra money on more processing power (low emphasis on the requirement of resource usage) in order to facilitate reusability.

5.2.9 Maintainability

Maintainability is ranked as the third most important requirement. The reason for this is most likely the high costs that arise when upgrading or updating software. Support for software configuration management is considered a pre-requisite in order to facilitate cross platform and product reuse, and hence these requirements are tightly coupled. Also, updating existing software by replacing erroneous software components requires efficient tool support.

5.2.10 Understandable

Understandability is not a primary requirement. This means that the companies are willing to spend some money on training personnel in software development in order to reach primary goals like reusability, portability and testability.

5.2.11 Source Code and Static Configuration

Not much focus is spent on the derived requirements. These requirements should perhaps not be compared with the other requirement since they are tightly coupled to primary requirements. This is rather to be seen as means to reach other requirements. For example, it is not possible to debug the application source code if the software components are delivered in a binary format.

This might be considered a weakness of the study, but we include the results for consistency reasons.

5.3 Discussion

It is interesting to see that the basic properties of CBSE (e.g. reusability, maintainability, and portability) are highly valued by industry. This might be biased due to the fact that this case-study deals with component-based development. However, the relative importances between the listed requirements are obvious and should be seen as a driver for component-based software.

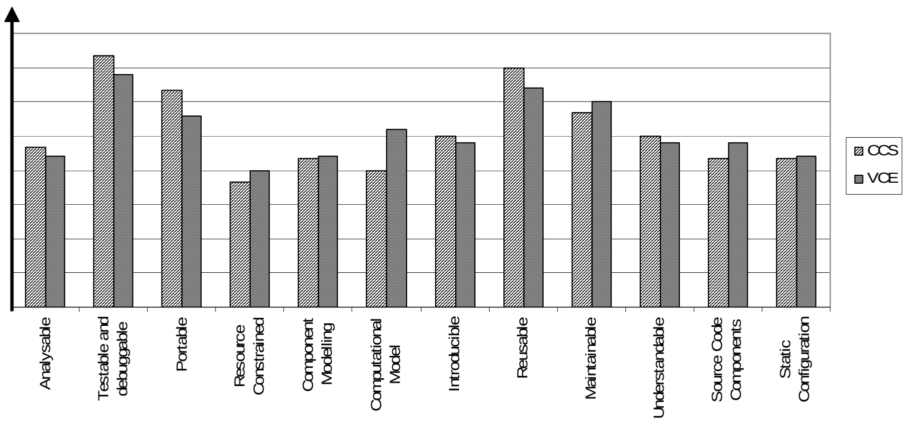


Figure 5: Requirements from the two companies

Also, it is interesting to see that the results from the two companies (see Figure 2) correspond with each other very well. Bearing in mind that the two companies represent two different types of control system developers, OEM and sub-contractor, these similarities are even more striking. Another interesting conclusion from this case-study is that the development process related requirements (i.e. introducible, reusable, maintainable, and understandable) is considered to be substantially more important then the technical requirements. Hence, the research community should not overlook these problems but rather spend more focus on issues like, e.g., support for software configuration management.

6 Component Technology Evaluation

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 [1] and Java Enterprise Beans [2]) since it is the only technology that explicitly address embedded and real-time issues. Also, the Windows CE version of .NET [1] 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.

6.1 Research Method

The research presented in this article started with a preliminary literature study, summarised in the state-of-the-art report [30]. The report is based on about 30 articles summarising the area of component-based software engineering for safety critical embedded applications. Understanding the state-of-the-art and state-of-practice component technologies was a prerequisite for the subsequent work. Based on the preliminary literature study – a qualitative case-study interview protocol (i.e. a case-study questionnaire) [22] was put together.

6.2 PECT

A Prediction-Enabled Component Technology (PECT) [12] 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

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

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) [12] 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.

6.3 Koala

The Koala component technology [9] 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 [31]. 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 [24]. 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

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

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. 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.

6.4 Rubus Component Model

The Rubus Component Model (Rubus CM) [29] 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) [32] 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 (Sect. 4.2.3), 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.

Extra-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 (Sect. 4.2.6). 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

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

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).

6.5 PBO

Port Based Objects (PBO) [33] combines object oriented design, with port automaton theory. PBO was developed as a part of the Chimera Operating System (Chimera OS) project [34], 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.

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

6.6 PECOS

PECOS[6] (PErvasive COmponent Systems) [8, 35] is a collaborative project between ABB Corporate Research Centre[7] and academia. The goal for the PECOS project was to enable a 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 extra-functional properties such as memory consumption and timeliness.

Extra-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.

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 [36] to model concurrent activities like component compositions, scheduling of components, and synchronisation of shared ports [37]. Debugging can be performed using COTS debugging and monitoring tools. However, the component technology does not support debugging on component level as described in Sect. 4.2.2.

The PECOS component technology uses a layered software architecture, which enhance portability (Sect. 4.2.3). There is a Run-Time Environment (RTE) that takes care of the communication between the application specific parts and the real-time operating system. PECOS use 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 (Sect. 4.3.1) 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 [38]. 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/

	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 6: Grading of component technologies with respect to the requirements

6.7 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 [17] for resource constrains systems, and RT-CORBA [39] 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. Minimum-CORBA defines a subset of the CORBA functionality that is more suitable for resource-constrained systems, where some of the dynamics is reduced.

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.

OMG has defined a CORBA Component Model (CCM) [3], which extends

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

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.

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 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 extra-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.

7 Summary of Component Technology Evaluation

In this section we assign numerical grades to each of the component technologies described in Sect. 6, grading how well they fulfil each of the requirements of Sect. 4. The grades are based on the discussion summarised in Sect. 6. 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 Fig. 6). For the CORBA-based technologies we have listed the best grade applicable to any of the CORBA flavours mentioned in Sect. 6.7.

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.

8 Conclusions

In this article 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.

We conclude that using software components and component-based development is seen as a promising to address challenges in product development, including integration, flexible configuration as well as support for software reuse.

One of the main contributions is that we show the relative importance of industrial requirements, in addition to the industrial requirements on a component technology for use in automotive applications. We describe and grade requirements on a component technology as elicited from two Swedish control-system developers. The requirements are divided into two main groups, the technical requirements and the development process related requirements. The reason for this is to clarify that the industrial actors are not only interested in technical solutions, but also in improvements regarding their development process.

The result can be used to guide modifications and/or extensions to existing component technologies in order to make them better suited for industrial deployment. The results can also serve as a platform for software engineering research, since researchers can be guided to put focus on the most desired areas within component-based software engineering.

Acknowledgements

We would like to thank CC Systems and Volvo Construction Equipment for their great support during our research. Especially we would likt to thank Jörgen Hansson (CCS), Nils-Erik Bånkestad (VCE) and Robert Larsson (VCE). Thank you!

References

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

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

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

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

[5] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proceedings of the* 7^{th} *International Symposium on Component-Based Software Engineering (CBSE7).* 2004 Proceedings Series: Lecture Notes in Computer Science, Vol. 3054, May 2004. Edinburgh, Scotland.

[6] Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin. Evaluation of Component Technologies with Respect to Industrial Requirements. In *Euromicro Conference, Component-Based Software Engineering Track*, August 2004.

[7] Anders M"oller, Mikael Åkerholm, Joakim Fr"oberg, and Mikael Nolin. Industrial grading of quality requirements for automotive software component technologies. In *Embedded Real-Time Systems Implementation Workshop in conjunction with the 26th IEEE International Real-Time Systems Symposium*, 2005 Miami, USA, December 2005.

[8] M. Winter, T. Genssler, et al. Components for Embedded Software – The PECOS Apporach. In *The 2^{nd} International Workshop on Composition Languages, in conjunction with the 16^{th} ECOOP*, June 2002. Malaga, Spain.

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

[10] J. Fröberg. *Engineering of Vehicle Electronic Systems: Requirements Reflected in Architecture.* Mälardalen University Technology Licentiate Thesis No.26, ISSN 1651-9256, ISBN 91-88834-41-7. Mälardalen Real-Time Research Centre, Mälardalen University, March, 2004, Västerås, Sweden.

[11] A. Möller. *Software Component Technologies for Heavy Vehicles.* Mälardalen University Technology Licentiate Thesis No.42, ISSN 1651-9256, ISBN 91-88834-88-3. Mälardalen Real-Time Research Centre, Mälardalen University, January, 2005, Västerås, Sweden.

[12] 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.

[13] A. Brown and K. Wallnau. The Current State of CBSE. *IEEE Software*, September/October 1998.

[14] C. Nordström, M. Gustafsson, et al. Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry. In *Eigth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, April 2001. Washington, USA.

[15] S. R. Schach. *Classical and Object-Oriented Software Engineering.* McGraw-Hill Science/Engineering/Math; 3rd edition, 1996. ISBN 0-256-18298-1.

[16] Ivica Crnkovic and Magnus Larsson. A case study: Demands on component-based development. In *Proceedings, 22th International Conference of Software Engineering*, Limerick, Ireland, May 2000. ACM, IEEE.

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

[18] International Standards Organisation (ISO). Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication, November 1993. vol. ISO Standard 11898.

[19] CiA. CANopen Communication Profile for Industrial Systems, Based on CAL, October 1996. CiA Draft Standard 301, rev 3.0, http://www.-canopen.org.

[20] SAE Standard. SAE J1939 Standards Collection. http://www.sae.org.

[21] SAE Standard. SAE J1587, Joint SAE/TMC Electronic Data Interchange Between Microcomputer Systems In Heavy-Duty Vehicle Applications. http://www.sae.org.

[22] R.K. Yin. *Case Study Research – Design and Methods.* Applied Social Research Methods Series, Volume 5, SAGE Publications, 2003. ISBN 0-7619-2553-8.

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

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

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

[26] 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 17th International Conference on Software Engineering*, April 1995. Seattle, USA.

[27] T. Nolte, A. Möller, and M. Nolin. Using Components to Facilitate Stochastic Schedulability. In *Proceedings of the 24th Real-Time System Symposium – Work-in-Progress Session.* IEEE Computer Society, December 2003. Cancun, Mexico.

[28] SIL. Safety Integrity Levels – Does Reality Meet Theory?, 2002. Report f. seminar held at the IEE, London, on 9 April 2002.

[29] K.L. Lundbäck, J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. In *Real-Time in Sweden – Presentation of Component-Based Software Development Based on the Rubus concept, Arcticus Systems: http://www.arcticus.se*, August 2003. Västerås, Sweden.

[30] M. Nolin et al. Component-Based Software for Embedded Systems - A Literature Survey. Technical report, MRTC Report No 104, ISSN 1404-3041, ISRN MDH-MRTC-104/203-1-SE, Mälardalen Real-Time Reseach Centre, Mälardalen University, June 2003. Västerås, Sweden.

[31] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001. ISBN 0-201-70332-7.

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

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

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

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

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

[37] 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.

[38] 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.

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