Nikolay Grozev

A comparison of component-based software engineering and model-driven development from the ProCom perspective

Master thesis, Software Engineering

Mälardalen University, School of Innovation, Design and Engineering

University of Sofia, Faculty of Mathematics and Informatics

Supervisor: Juraj Feljan, Mälardalen University
Consultant: Sylvia Ilieva, University of Sofia
Examiner: Ivica Crnkovic, Mälardalen University

28th June 2011

Abstract

Component-based software engineering (CBSE) and model-driven development (MDD) are two approaches for handling software development complexity. In essence, while CBSE focuses on the construction of systems from existing software modules called components; MDD promotes the usage of system models which after a series of transformations result with an implementation of the desired system. Even though they are different, MDD and CBSE are not mutually exclusive. However, there has not been any substantial research about what their similarities and differences are and how they can be combined. In this respect, the main goal of this thesis is to summarize the theoretical background of MDD and CBSE, and to propose and apply a systematic method for their comparison. The method takes into account the different effects that these development paradigms have on a wide range of development aspects. The comparison results are then summarized and analyzed.

The thesis also enriches the theoretical discussion with a practical case study comparing CBSE and MDD with respect to ProCom, a component model designed for the development of component-based embedded systems in the vehicular-, automation- and telecommunication domains. The aforementioned comparison method is refined and applied for this purpose. The comparison results are again summarized, analyzed and proposals about future work on ProCom are made.

Keywords

Component-based software engineering, Model-driven development, ProCom, Comparison

Contents

List of Figures

List of Tables

Acronyms

CBSE	Component-Based Software Engineering
MDD	Model Driven Development
MDA	Model Driven Architecture
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
IDE	Integrated Development Environment
UML	Unified Modeling Language
MOF	Meta-Object Facility
CIM	Computation Independent Model
PIM	Platform Independent Model
PSM	Platform Specific Model
ISM	Implementation Specific Model
AMDD	Agile Model Driven Development
SoC	Separation of Concerns
RUP	Rational Unified Process
ER	Entity-Relationship model
ORM	Object Role Modeling
VEG	Vehicle Expert Group
BPMN	Business Process Management
CRCD	Cyclical Requirement-Component Dependency
ADL	Architecture Description Languages
JVM	Java Virtual Machine
DLR	Microsoft's Dynamic Language Runtime
SEI	Carnegie Mellon Software Engineering Institute

1 Introduction

The decades of experience in software development abound in examples of project failures in spite of the ever growing investments in the field. As a result software costs have become a major component of corporate expense. It is not uncommon for the software and computing budgets of a company to exceed 10% of the annual corporate expenditures. Despite the huge investments it is a fact that the software industry has the highest failure rate of any engineering field. This is supported by Jones' statistics [1] that more than 75% of software projects end up late and 35% of the large ones are canceled before they finish. Jones summarizes these statistics by stating that currently software engineering is not a true engineering discipline and emphasizes on the need for sound engineering principles incorporated in the field [1]. A plethora of books, research papers and practitioners' analyses over the years have made software development complexity proverbial and have emphasized on the need for systematic means to tackle it.

Two promising systematic approaches to coping with the development complexity are component-based software engineering (CBSE) and model-driven development (MDD). CBSE focuses on the construction of systems from existing software modules called components, and makes a clear distinction between developing a component and developing a system. From the perspective of CBSE the development of a component should result with a reusable software module implementing a cohesive set of functionalities called a component. System development is the selection and "binding together" of existing components. A key concept for this paradigm is reusability, since the developed components are meant to be relatively self-sustained and thus may be used in various systems.

MDD brings the notion of models as main development artifacts to software development. The main idea is to build models which are close in nature to the targeted application domain and then gradually refine/transform these models until a fully functional system is achieved. Models and model transformations are the two key concepts in MDD. A popular analogy to MDD is drawn from civil engineering where a building is created after a series of models, starting from an architectural one which is then elaborated multiple times until a detailed blueprint is created. As McConnell states "*... the power of models is that they're vivid and can be grasped as conceptual wholes*" [2]. It is this conceptual entirety and understandability that

makes models valuable, since it is much easier first to design roughly and then to refine the achieved models, rather than to think in the algorithmic/implementation level from the start.

CBSE and MDD have been used successfully multiple times to mitigate the complexity when developing general purpose and enterprise applications, but they have rarely been utilized for the development of embedded systems. This is not surprising, since most CBSE and MDD approaches do not address adequately the specifics of the embedded domain when it comes to extra-functional properties. Besides, most modern paradigms, programming languages and tools provide additional abstraction layers to the software developers, which very often hurt the overall performance. Thus embedded systems are usually manually written in low-level programming languages agnostically of the practices of paradigms like CBSE and MDD. However, the need to develop embedded software within time and budget constraints forces practitioners and researches to revise some of the advances from general purpose development so as to meet the specifics of embedded software development.

One notable such initiative is PROGRESS [3], which is a large research project bringing CBSE techniques to the development of embedded systems. More specifically PROGRESS aims to provide know-how, theoretical methods, and a set of tools and models that facilitate component based development of embedded real-time systems. Among them is the component model ProCom [4]. It also accommodates certain aspects of MDD by supporting different views of a system and its components through various models. Thus ProCom provides the embedded software developers with means from both CBSE and MDD so as to work at a higher level of abstraction and reduce development efforts and time through component reuse.

1.1 Purpose

This thesis has two main goals. The first one is to systematically compare CBSE and MDD. This comparison is meant to provide insight into their theoretical backgrounds, typical usages, practical aspects and key differences and similarities. Few similar research projects have been carried out before and thus the thesis presents some novel results. The thesis also aims to analyze the comparison results and to suggest how these two paradigms can be combined.

The second main goal is to enrich the results from the general comparison with a case study comparing CBSE and MDD with respect to ProCom. This comparison aims to add practical

aspects to the general one and to illustrate the key differences and similarities between CBSE and MDD with the concrete technological aspects of ProCom. Also, the thesis aims to analyze further the outcome of the practical case study and to suggest future improvements of ProCom so as to better accommodate features of the two paradigms.

1.2 Structure of the thesis

The rest of the thesis is organized as follows. Section 2 summarizes the research with related scope to the one of the thesis. Section 3 summarizes the background of the paradigms and then introduces and motivates a comparison method. Section 4 describes the general comparison of CBSE and MDD. The section also provides the results of the application of the previously introduced method and analyzes them. Section 5 introduces the ProCom component model and a revised version of the previous comparison method suitable for the comparison of CBSE and MDD from the viewpoint of ProCom. Section 6 presents the second comparison of CBSE and MDD from the perspective of ProCom. This section also summarizes and analyzes the results of this comparison. Section 7 concludes the results of the thesis.

2 Related Work

As mentioned just a few related to the thesis scope studies have been identified. The most related one among them is a paper by Törngren et al. [5] which defines and applies a framework for the comparison of CBSE and MDD in the context of the development of vehicular embedded systems. The framework codifies important issues in the development of embedded systems and the comparison section discusses the approaches of both paradigms when it comes to each of these issues. Though the scope of this paper is obviously very much related to the one of the thesis the paper does not compare CBSE and MDD in general and does not discuss the comparison with respect to some concrete technology like ProCom.

An interesting method facilitating model-driven and component-based development of embedded systems is MARMOT, which stands for *Method for Component-Based Real-Time Object-Oriented Development and Testing*. A few papers have been written to describe and validate the applicability of this method. Among these are two papers by Bunse et al. [6] [7] which describe exploratory studies about the possible practical combination of MDD and CBSE and the overall usefulness of MARMOT. Both publications conclude that combining CBSE and MDD has a positive impact on reuse, effort, and quality. However, these studies do not provide a systematic comparison of the two paradigms. Also the generalization of their results can be considered questionable since the subjects of analysis were small projects, developed in academic environment.

Several papers describe approaches and tools combining CBSE and MDD in the embedded domain. Among them is a publication by Schmidt et al. [8] showing how MDD tools and techniques can be used to specify, analyze, optimize, synthesize, validate, and deploy component middleware platforms which are customizable for the needs of distributed real-time and embedded (DRE) systems. The approach is exemplified with the CoSMIC MDD tool suite developed and designed especially for these purposes. Other papers discussing combinations of CBSE and MDD were created as a part of the DARPA PCES project [9], which provides a variety of capabilities for model-driven implementation and analysis of component middleware systems. Most notably the publications by Childs et al. [10] and Hatcliff [9] describe the approach adopted by the project and some practical observations about Cadena – a prototype developed as a part of the project. Carlson et al. [11] also suggests

a combination of the two paradigms, only this time with respect to ProCom. More specifically the paper describes the ProCom approach for deployment and synthesis of component based embedded systems which makes use of MDD techniques. Again none of these publications presents a systematic comparison of the two paradigms, though combining CBSE and MDD is still related to scope of the thesis.

Przybylek [12] introduces and applies a systematic comparison method for the comparison of two of the most popular post object-oriented paradigms – aspect-oriented programming and composition filters. The study investigates in details how the two paradigms deal with a predefined set of issues inherent for the post-OO approaches – e.g. implementing crosscutting concerns etc. Unfortunately these issues are too specific for the post-OO approaches and this comparison method cannot be directly reused for comparing CBSE and MDD, though the idea of defining a set of issues or aspects as a basis for comparison is viable for the purposes of the thesis.

3 Background of the general comparison

The first two subsections of this section present an overview of CBSE and MDD respectively. Their goal is to familiarize the reader with the essence of the two paradigms before diving into the detailed comparison. The third subsection introduces and motivates a general comparison method that can be employed to compare other development approaches as well.

3.1 Overview of component-based software engineering

The philosophy of CBSE is to build software systems from pre-existing components rather than to develop them from scratch. Ideally this should lead to reduced development time and efforts because of the component reuse. Jones summarizes well the engineering nature and the motivation for such an approach: "*So long as software applications are hand-coded on a line-by-line basis, "software engineering" will be a misnomer. Switching from custom development to construction from certified reusable components has the best prospect of making really significant improvements in both software engineering disciplines and in software cost structures.*" [1].

Admittedly, the paramount concept in CBSE is the ***component***. There have been many different definitions of what a component is. These definitions mostly differ in the nuances of the context in which the term is defined and are complementary to each other. All of them adhere to the intuitive idea that "*Components are things that can be plugged into a system*" [13]. Probably the most popular definition is the one from Szyperski: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party*" [14]. The requirement that a component should be independently deployed implies that a clear distinction from its environment and other components is made. A component does its communication with its surroundings only through its interfaces while its concrete implementation remains encapsulated and inaccessible from its environment and other components. This is what makes a component a subject to direct third-party composition and deployment. Usually components are in a ready to use state and it is not required to recompile or relink an application when adding a new component to it [15].

Another important concept in CBSE is the ***interface***. It is used in almost all definitions of component. As to Szyperski [14] an interface of a component can be defined as a specification of its access point. A component would typically provide a set of interfaces corresponding to different access points. Usually each access point provides a different service aimed at a distinct client need. A component is normally agnostic of its clients. A component's client has knowledge only about its interface and knows nothing about the concrete implementation. Thus Szyperski emphasizes on the contractual philosophy of interfaces [14]. It is important for a component to specify well its interfaces and for the clients of that component to obey the contract imposed by the interface they are using. Components should also obey "their part" of these contracts by implementing correctly the specifications of their interfaces. An interface does not provide implementations of the operations it defines - it just specifies them. This makes it possible to change the underlying implementation of a component without having to modify the rest of the application.

Interfaces can be classified as ***imported*** or ***exported***. An exported interface of a component is an interface provided and implemented by it. An imported interface of a component is one which is required for it to work correctly. Exported and imported interfaces are also called outgoing and incoming. For example let's consider a component **A** which exports/provides an interface $\mathbf{a_1}$ for logging programming events and imports/requires an interface for I/O manipulations $\mathbf{a_2}$. In order to use the logging functionality of $\mathbf{a_1}$ provided by **A,** a client must ensure that a component that provides/exports an I/O manipulation interface $\mathbf{a_2}$ is present. Interfaces can be versioned, so each component can specify which interface version(s) it requires or provides. Versioning is important since it facilitates interface evolution. In the sense of the previous example component **A** may require interface $\mathbf{a_2}$ with a version higher than 1.0, since in this version a new feature used by **A** has been introduced. Hence a client of **A** should ensure the presence of a component providing $\mathbf{a_2}$ with a version within this required interval.

A major topic in CBSE is the specification of components' functionality and behavior. Since components are "visible" only through interfaces their specification contains only the specifications of their interfaces. There are three kinds of interface specifications - ***syntax***, ***semantic*** and ***extra-functional***. A syntax specification is absolutely necessary for a component interface. It specifies what are the operations of an interface by their names, parameters etc. The semantic specification describes the functionalities of the operations of an interface. A semantic specification would typically define a set of preconditions and

postconditions for each operation similarly to the ones from the development technique Design by Contract [16]. Also a semantic specification may include invariants of the interface state model and inter-interface conditions, which are predicates over the state models of all interfaces of a component [15]. Extra-functional specifications define the so-called extra-functional properties of interface operations such as performance, reliability etc. Such specifications are especially important for embedded, real-time and mission-critical systems where it is vital to foresee the exact behavior of a system. However, it is laborious, difficult and sometimes even impossible to describe precisely the behavior of an operation in terms of a given extra-functional property. Thus there is a need for ways to describe approximately such properties. One approach which often works well enough for specifying performance is to give asymptotic estimations of the memory consumption and needed time for execution. A more general approach is to use the so-called credentials, which provide numerical estimations and descriptions of an interface property [15].

As discussed, a component communicates with its environment and other components exclusively through its interfaces. However, nothing has been mentioned so far about what this environment exactly is and how the binding of components actually happens. Two other notions need to be defined for these purposes – namely ***component framework*** and ***component model***. As the survey by Bachman et al. [17] emphasizes some authors do not distinguish between these concepts. In this thesis they will be considered different. By component model we shall mean a set of well defined standards and conventions for a component. More specifically, a component model defines what a component is and how it interacts with other components. In order for a group of components to work together it is necessary for them to be compliant to the same component model. On the other hand a component framework is a concrete technical solution. It allows components compliant to a given component model to work together. Bachman et al. liken a component framework to a mini-operating system, because it manages resources shared by components, and provides the mechanisms for their communication just like an operating system does with processes [17].

Figure 1 brings all the CBSE notions previously described together and shows how a typical component based system is organized.

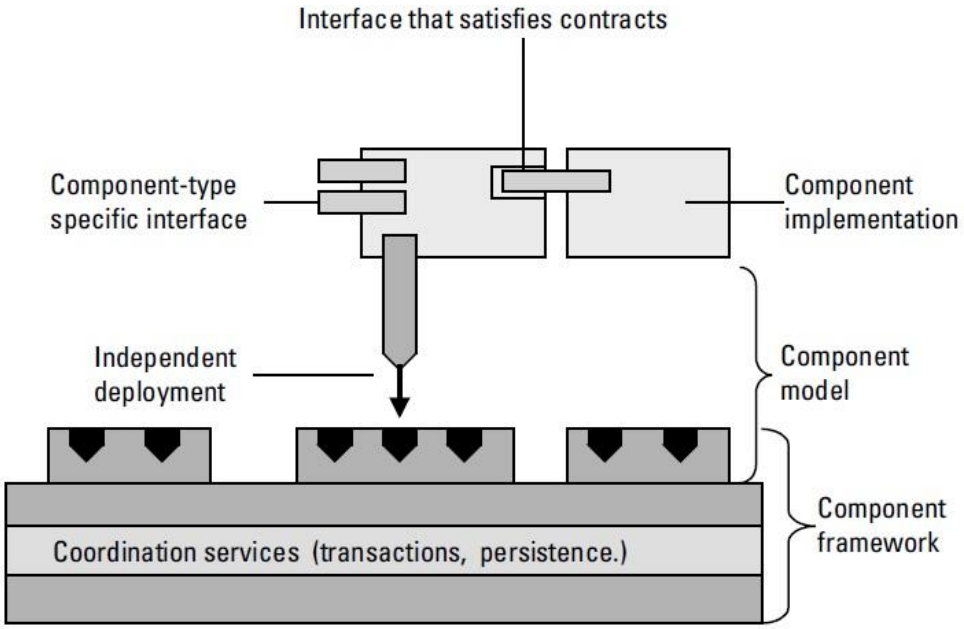

Figure 1 Design of a typical component based system [**15**].

As depicted in Figure 1 a component framework provides the technical means for components to work together. Crnkovic and Larsson liken a component framework to a circuit board in whose positions components are inserted [15]. The component model defines the requirements for a component which is to be deployed in the framework. Each component defines its incoming and outgoing interfaces. All incoming interfaces of a component must be resolved before it is successfully composed into a framework.

A subfield of CBSE that is being actively worked on is component system modeling. Its aim is to create simplified representations (models) of the components and their interactions. Different modeling formalisms have been defined for the purpose. Component system models usually describe the whole architecture of the component system under consideration, but often can be used to describe lower level designs as well. It can be argued that modeling is not an integral part of CBSE, since none of the popular definitions of CBSE mentions it and many popular component technologies provide no means for adequate and useful modeling. However, component modeling is an essential part of the component based development process. Practicing component based development without having at least coarse-grained models of the needed components and their interactions can be chaotic and poses a great risk for the whole project. Also, having such models allows for early evaluation of the system before it has been implemented. Hence, in the rest of the thesis component system modeling will be considered as an integral part of CBSE, though it is not yet a well established practice and there is little consensus about the modeling best practices in the context of CBSE.

3.2 Overview of model-driven development

As Selic emphasizes it is hard to imagine that a building or an automobile can be created without first creating a set of specialized system models [18]. Many software systems are much more complex than this and thus it seems logical that modeling techniques from other engineering disciplines are adopted in software development. That is what MDD tries to achieve. Models in MDD are used to reason about a problem domain and design a solution in the terms of that domain. In essence the philosophy of MDD is to start with models which are simple and include only the most essential parts of the designed system. Then typically these initial models undergo series of elaborating transformations until a final model is achieved. This final model actually is the target system itself and is the model with the lowest abstraction level.

Raising the level of abstraction in order to simplify software development has always been a primary goal in software engineering. An outstanding example for this is the transition from second-generation (assembly) languages to third-generation programming languages (3GL). 3GL languages are much more programmer-friendly and allow engineers to program agnostically of some non-essential, hardware specific details. MDD can be seen as another logical step in this direction. Usually when following a MDD approach the models are lacking any details unrelated to the system view being represented – not only the low-level hardware details as it is in 3GL. MDD promotes shifting the focus from code and programming constructs to abstract models throughout the whole development lifecycle, including documentation, requirements gathering and analysis, design, testing etc.

The term ***model*** has several different definitions and is a primary one in MDD. As to Beydeda et al. "*a model is a set of statements about some system under study*" [19]. Rahmani et al. augment this definition by defining a model as "*a description or specification of a system and its environment for some certain purposes*" [20]. The key idea that makes this definition more substantial than the first one is that a model is aimed at a few distinct purposes. Every model is meant to allow engineers to reason about and design the target system ignoring irrelevant details and focusing on the artifacts related to the purposes of the model. A variety of modeling concepts and notations may be used to emphasize on specific views and perspectives of the system, depending on what is considered relevant at some point. These concepts and notations are usually a part of the model ***formalism*** (or language), which defines the model's syntax and semantics [19]. Rothenberg captures well the essence of these

definitions and the concept of modeling in general in his self-explanatory definition: "*Modeling in its broadest sense is the cost-effective use of something in place of something else for some purpose. It allows us to use something that is simpler, safer, or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality*" [21].

When a complex system is being designed it is quite common to develop not just one, but a set of models. These models usually present different views to the system and might be at different levels of abstraction. It is often necessary to convert between the different models, or to create new ones based on the already existing models. This is achieved through transformations. A model ***transformation*** can be defined as an operation which takes as input a model of a system described in formalism $\mathbf{F_1}$ and produces another one described in formalism $\mathbf{F_2}$. Transformations are also a primary notion in MDD and have been a subject of many both theoretical and practical studies. Many different classifications of model transformations have been created over the years. In this section only a few of the most notable ones (also described by Beydeda et al. [19]) are presented.

A very simple distinction between transformations is based on the type of the input and output formalism. More specifically a transformation is called ***endogen*** if the formalism of its input model and the one of its output model are the same ($F_1 = F_2$). Otherwise it is called ***exogen*** ($F_1 \neq F_2$).

Another distinction can be based on the abstraction level of the input and output models. Transformations evolving the input model into an output model at the same level of abstraction as the input one are called ***horizontal***. Ones which transform the input model into a model that is closer to the run-time platform (e.g. source code generation) are called ***vertical***.

Another important distinction is based on the amount of manual work needed for a transformation. In this respect a transformation can be classified as ***fully automatic***, ***partially automated*** or ***manual***.

The last classification to mention is based on the technique used for the description and execution of a transformation. A ***declarative*** transformation is described by rules, which are

specified by pre- and post-conditions. These pre- and post-conditions describe statements about the states of the input and output models which hold true respectively before and after a successful execution of a transformation. Transformations which define series of actions which should be taken to create the output model from the input one are called ***operational*** (or imperative).

So far the general idea and the most important notions of MDD have been described. However, nothing has been mentioned about how modeling fits in software development, and more specifically what have been the most notable practices. Beydeda et al. suggest a classification of the current MDD practices with respect to the ways models are synchronized with the source code [19]. Note that for the sake of simplicity in this classification it is not considered that source code itself is a system model, though it can be viewed this way.

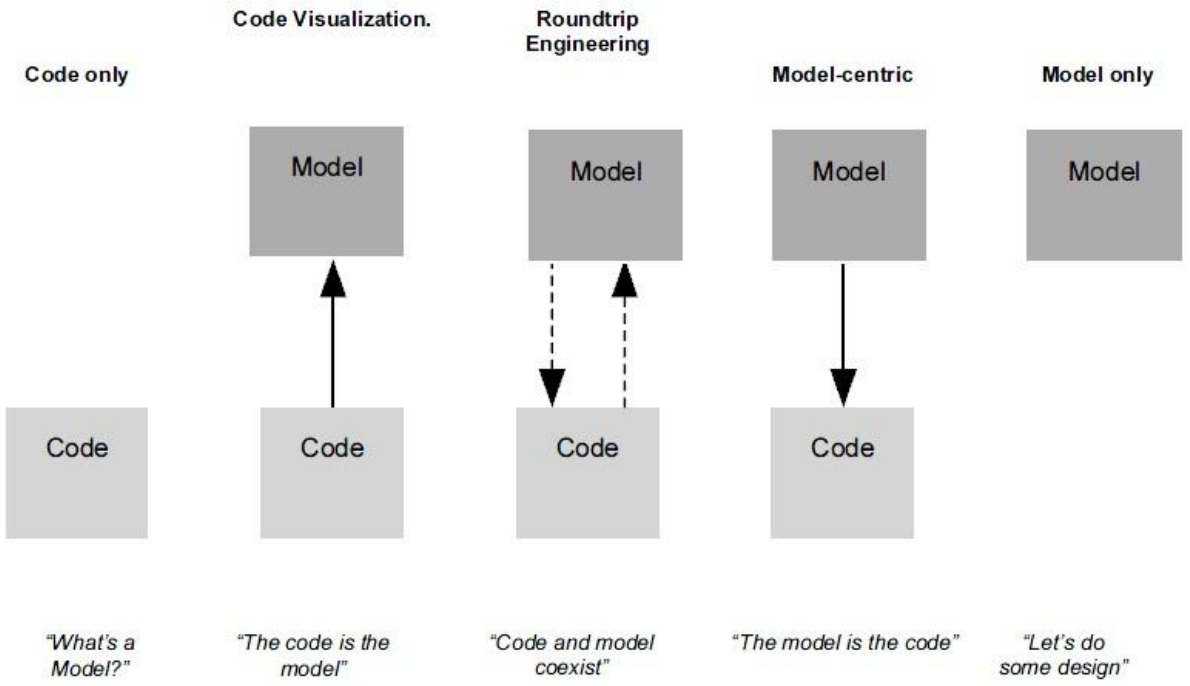

Figure 2 The modeling spectrum [19].

Figure 2 shows the spectrum of modeling approaches with respect to such a classification. The distinct groups of practices are:

- ***Code only*** - this approach considers the source code (usually written in a third-generation programming language) as the only representation of the system. If any separate modeling is done it is informal and intuitive. Obviously this approach is only adequate for small projects developed by very small teams or a single developer.

- ***Code Visualization*** - this approach uses the source code itself as an initial representation over which transformations are applied. These transformations result in different visual representations of the code. These are used by the software engineers to better understand the code. Such transformations are usually fully automated.

- ***Roundtrip Engineering*** takes the idea of Code Visualization one step further. Following this approach firstly a model is created, which is then transformed to source code or source code stubs, which are later manually implemented. Whenever a change is made to the code, the initial model is updated to reflect it. Thus it is guaranteed that the code and the model are always synchronized. Usually this approach is supported by a set of tools that automate the needed transformations between the code and the abstract model.

- ***Model-centric***. This approach requires that the models of the system are detailed enough so that the automatic generation of the full source code is possible.

- ***Model only***. In this approach models are used for design discussions and analysis, better understanding of business requirements, communication etc. Such models are usually not used in the actual implementation of the system.

The existence of so many MDD related practices shows that MDD defines a very general and abstract development approach, which allows different interpretations and unfortunately misinterpretations of its philosophy. In fact MDD defines no formal requirements as to how models and model transformations are described. Also it does not concisely define a development process. Thus there is a need for a set of development standards which augment the general idea of MDD. The most prominent such initiative is Model Driven Architecture (MDA) which is created and supported by OMG. MDA employs OMG's established modeling standards like Unified Modeling Language (UML) [22] and Meta-Object Facility (MOF) [23] to define an open, vendor-neutral MDD based approach [19]. MDA defines the following types (layers) of models:

- ***Computation Independent Models (CIM)*** - these are the initial type of models when following the MDA approach. The CIM models only represent the business context and business requirements.

- ***Platform Independent Models (PIM)*** - a refinement of the CIM models. Specifies services and interfaces that the target systems must provide to the business. PIM models do not include any platform specific information.
- ***Platform Specific Models (PSM)*** - a refinement of the PIM with respect to the chosen technological platform.
- ***Implementation Specific Models (ISM)*** - the final source code of the system.

The models in each of these layers are elaborated to construct the models of the successive layer. Figure 3 shows how the layers correlate to each other and what kinds of models are typically included in each of them.

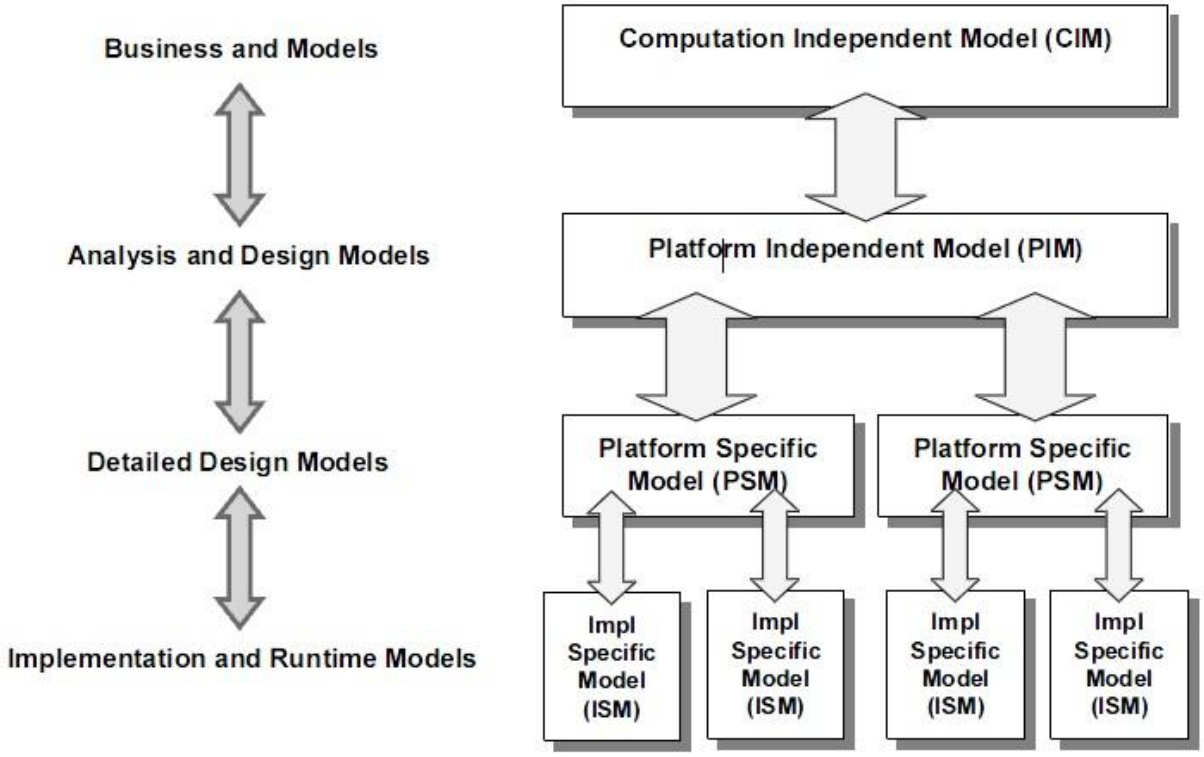

Figure 3 MDA layers and transformations [19].

MDA is the most popular MDD standard, but it has also been heavily criticized for being overly complex, too formal and requiring too much upfront investment in training. An alternative model-driven approach called Agile Model Driven Development (AMDD) has been introduced as a lightweight alternative to MDA. AMDD is aimed at incorporating better modeling into the agile software development, thus scaling it beyond its usual small and co-located team approach [24]. However, it can be argued that AMDD is not a true MDD approach since it discourages the constant usage of detailed models as main artifacts

throughout the whole development lifecycle. AMDD promotes the usage of models which are only good enough to drive the overall development efforts. AMDD is becoming more and more popular in the context of the constantly growing interest in agile methodologies.

3.3 Method of comparison

There are two general approaches for comparing development paradigms. The first is a non structured one. Following this approach a comparison would typically be at a high-level and provide mostly theoretical and philosophical reflections on the common and different features of the paradigms. On the contrary, a structured comparison would require a clearly defined comparison method. Such a method would typically select important aspects of software development which are very much influenced by the used development paradigms. The comparison then would comprise of an in depth and justified considerations of how the targeted paradigms influence these aspects.

In this thesis an intermediate approach is taken. Firstly a systematic comparison method is introduced and motivated. Then this method is applied by discussing and comparing the influences of both CBSE and MDD over the selected aspects. These discussions and comparison are supported by references to established publications, practical case studies and where appropriate metrics. The comparison results are used as a basis for a brief and higher level comparison, which extracts the essence of the previous one and proposes ways for combining CBSE and MDD.

The method proposed in this thesis is based on a small, two-level hierarchy of what is called ***comparison aspects***. The first level of this hierarchy contains aspects correspondent to the main activities in software development. The ***Requirements*** comparison aspect corresponds to the activities of requirements gathering and requirements analysis. The ***Design*** comparison aspect corresponds to all low-level and high-level design activities. The ***Development*** aspect corresponds to the activities of implementation, validation, verification, deployment and maintenance.

Besides these, the first level also contains comparison aspects related to the decision making whether a paradigm should be used or not. Such a decision usually depends on two types of factors - factors related to the business needs and ones related to the specifics of the

organization developing the software. Thus the first hierarchy level also contains the following comparison aspects - ***Business specifics*** and ***Organizational specifics***.

The comparison aspects from the first level are very general and it is hard to directly compare development paradigms with respect to them. Thus each of them is further divided into more specific ones. Figure 4 shows the whole comparison aspects hierarchy in the form of a simple mind map. The following subsections list all comparison aspects and define the questions that should be answered in the process of applying the comparison method.

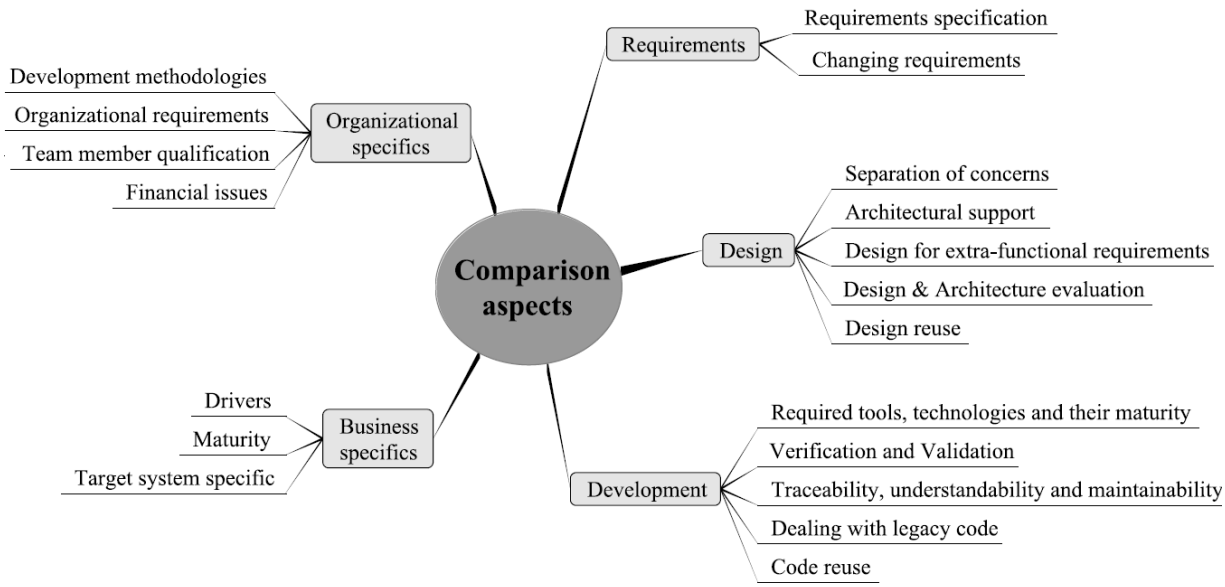

Figure 4 Hierarchy of comparison aspects

The method proposed herein is a general one and can be used to compare development approaches other than MDD and CBSE as well. Moreover it can be used to compare software engineering objects other than development approaches as well, by comparing them from the perspective of each aspect. This is because the comparison aspects defined by the method are general and are spanned in scope throughout the whole development process. This allows for systematic analysis of the impacts the objects under consideration have on the different phases of software development. Some of the considered objects may not interfere in any way with the matters implied by a comparison aspect and this should be as well taken into consideration when applying the method.

3.3.1 Business specifics

When discussing this top-level aspect the facts that make the approaches selected for comparison attractive or unattractive from a purely business point of view should be considered. Such a discussion should clarify what business needs does each of the considered approaches meet and what are the business threats posed by each of them. Also it should be reasoned about the appropriate contexts in which the paradigms can be used successfully, judging by the experiences and considered best practices. Follows a list of the sub-aspects and the questions to be considered when discussing them:

- ***Drivers*** – What are the motivations and aims for the paradigms?
- ***Maturity*** – For how long have the paradigms existed? What is the overall opinion of the practitioners? How many success stories do we know?
- ***Target system specifics*** – What kind of systems (in terms of domain, size etc.) are usually built using these paradigms?

3.3.2 Requirements

The discussion of this aspect should encompass how flexible and convenient are the approaches regarding requirement specification and reaction to requirement changes. It should also consider the benefits of each of the selected approaches when it comes to requirements-to-design transformation. The sub-aspects to consider are:

- ***Requirement specification*** – Do the paradigms help in modeling the initial requirements? If they help - how? Can the modeled requirements be easily used for a design?
- ***Changing requirements*** - How flexible are the paradigms towards changing requirements?

3.3.3 Design

This top-level aspect implies a discussion on the different approaches to software design at all levels of granularity and abstraction when using the selected paradigms. Moreover the following sub-aspects should be considered:

- ***Separation of concerns (SoC)*** - How is SoC achieved? What are the building blocks of SoC?
- ***Architectural support*** - Do the paradigms help in describing architecture? If so - how? Is there anything that should be considered when creating the architecture for software that is to be built by using these paradigms?
- ***Design for Extra-functional requirements*** - How do the paradigms deal with extra-functional requirements at design time?
- ***Design & Architecture evaluation*** - What are the design best practices for these paradigms? Is there an easy way to evaluate a design and identify problems early?
- ***Design reuse*** - If a design is once created using these paradigms is it possible to reuse it (even partially) for other systems?

3.3.4 Development

This top-level aspect implies a discussion encompassing all issues related to system development, verification, validation and maintenance. More specifically the following sub-aspects should be considered:

- ***Required tools, technologies and their maturity*** - What kind of tools are required? How mature are these tools? Is it possible to avoid vendor lock-in?
- ***Verification and Validation (V&V)*** - Are there some benefits or handicaps when verifying or validating software built using these paradigms?
- ***Traceability, understandability and maintainability*** - How easy is it to understand a solution when using these paradigms? How maintainable are the built systems? How easily can a new team member "catch up"?
- ***Dealing with legacy code*** - How legacy code can be encapsulated, so that new development can take place without having to modify it?
- ***Code reuse*** - Can we build modules that can be reused in other systems? What is the price for such a reuse - glue code, adapters etc.?

3.3.5 Organizational specifics

This aspect implies a discussion on what an organization adopting the paradigm should take into consideration. Moreover the following sub-aspects should be considered:

- ***Development methodologies*** **-** Are there any problems using these paradigms when working with the most popular development methodologies - waterfall, iterative approaches, agile approaches, RUP etc.?
- ***Organizational requirements*** **-** Is there anything special about an organization that would like to use these paradigms? How mature should it be? What kind of in-house experience and know-how should it possess?
- ***Team member qualification*** **-** What kind of qualification should the engineers have? How easy are the technologies and paradigms to adopt and learn?
- ***Financial issues*** – Are the tools, technologies and employee trainings expensive? For how long does such an investment pay off?

4 General comparison of CBSE and MDD

So far the backgrounds of CBSE and MDD have been summarized and a general comparison method has been introduced. The two paradigms are still areas of active research which tries to overcome (or at least to mitigate) their many open questions – e.g. dealing with extra-functional properties in CBSE and managing models versions. Being mostly a bottom-up approach CBSE hopefully can make use of many of the advantages that MDD provides and vice versa [5]. This makes the comparative analysis of the two paradigms a potential source of improvements. This section deals with one of the two main purposes of this thesis – the general comparison of CBSE and MDD. The comparison is focused on the theoretical properties of the two paradigms. However, the current technological state of affairs of both paradigms is also discussed in details. It even predominates when considering matters which are currently being actively researched and still there is no consensus about. For this purpose the previously defined general method is used and this section discusses all of its comparison aspects with respect to CBSE and MDD. The final subsection summarizes and analyzes the comparison results.

4.1 Business specifics

4.1.1 Drivers

The main business driver for CBSE is component reuse. Ideally a component should be relatively self-sustained and encapsulated and should implement a coherent set of functionalities. This allows for a component to be directly reused (or with a minimal adaptation) in many other systems. Obviously, this reduces the needed efforts for development, test and maintenance. Thus other drivers consequent of component reuse include reduced time-to-market, enhanced quality and simplified maintenance [11], [5].

MDD also facilitates reuse though at a different level. Some of the created models when following a MDD approach can be reused in other systems as well. However, reuse is not considered the most motivating factor to adopt MDD, though in certain cases it can be very beneficial as well – e.g. the reuse of Platform Independent Models (PIM) in MDA to implement a system in different platforms. The main driver for choosing MDD is considered

to be the ability to abstract away irrelevant details so as to design more easily. This also allows for early evaluation and correction of the chosen design approaches leading to reduced overall costs and efforts [5].

4.1.2 Maturity

It is considered that the notion of component reuse, which is paramount in CBSE, was introduced by Mcllroy [25] in 1968. However, it was only in the 1980s and the 1990s that the theory of CBSE started to evolve fast to what we know today. In the last decades dozens of CBSE related technologies have surfaced. There are CBSE technical solutions designed to be used when developing in the most popular third generation programming languages. Using component technologies like OSGI [26], Java Beans [27] and .Net [28] has become common in the development of almost all kinds of software. This comes to show that CBSE philosophy is now widely understood by practitioners and has proven itself in practice.

Unfortunately, the majority of the existing component technologies do not implement some of the theoretical achievements in the area. For example most such technologies only allow syntactic component specification, but do not allow for semantic or extra-functional ones. This can be attributed to the fact that components are often understood in different ways in academia and in industry. Practitioners tend to think of a component as a large reusable piece of software with complex internal structure that does not necessarily have interfaces well-understood in terms of semantics and extra-functional properties [15]. Another reason for this mismatch between theory and practice may be that these theoretical achievements are not adequate to the requirements to develop software within time and budget constraints which forces practitioners to resort to more informal methods. All these come to show that CBSE theory, technologies and practitioners still have a way to go before CBSE can be considered mature in its entirety.

The first formal approaches to data modeling were introduces in 1970's [29]. Among them are the highly successful Entity-Relationship (ER) model and Object Role Modeling (ORM). In 1995 UML emerged, and UML diagrams since then have become synonymous to models in the minds of many software developers. Indeed it is now quite common to use UML and ER diagrams in the development of large projects. However, practitioners' attitudes towards modeling (and UML in particular) are not uniform. There is a group of them (e.g. those practicing MDA) who consider that modeling with formalisms is a necessity for successful

software development. On the other hand MDD and more specifically MDA has been a subject to a lot of criticism for being too complex and difficult to use [29]. Many practitioners do not believe that an investment of both money and time in modeling training pays off and thus still a lot of software is developed only by coding. Very similar ideas have been expressed by advocates of the agile development principles and most notably by Ambler. However, MDD and agile principles are not mutually exclusive. Even though agilists are critical about MDA and UML, Ambler defines Agile Model Driven Development (AMDD) which is an agile version of MDD. AMDD advocates models which are "*just barely good enough*" to drive the overall development efforts [24].

All these different attitudes to modeling show that almost all practitioners value modeling to some extent, but MDD still has not proven indisputably to be a mature paradigm which defines a set of universal best practices.

4.1.3 Target system specifics

CBSE technologies have been used in the development of virtually all kinds of software. Outstanding examples of CBSE technologies used successfully in a wide range of applications are Java Beans and EJB. These technologies constitute the core of the Java Platform Enterprise Edition (Java EE) and are used throughout industry to implement web interfaces, transaction safety, messaging and persistence. Another example of a successful CBSE technology in the Java world is OSGI which has been used successfully by the following markets: enterprise, open source, mobile, telematics, SmartHome and E-Health [26]. A notable extension of the OSGI component model is the Eclipse platform [30] which has been widely used to build rich desktop applications like the Eclipse IDE itself.

CBSE has been put into practice even in domains where non-functional requirements are equally or even more important than the functional ones. Examples of this are the embedded and real-time systems. Traditionally such applications are developed in an assembler or in C [15]. CBSE is usually avoided for such systems since most of the existing component based technologies do not allow specifying the non-functional attributes of the components. There are some developments like ProCom and Cadena which bring CBSE to the development of such systems. OSGI is also being developed to accommodate such changes by the Vehicle Expert Group (VEG) [31]. Then again, CBSE is still not considered the method of choice

when developing such systems, even though methods for applying component based technologies in these domains are being actively researched.

CBSE is usually a sane method of choice when developing a product line of software products. Well designed components are ideal subjects for proactive, systematic, planned, and organized reuse. Crnkovic and Larsson discuss the advantages that the component approach has compared to alternatives like using libraries or OO frameworks for building product lines [15]. Some component models have been designed especially for the purpose of developing product lines. One notable such model is Koala [32] which is used by Philips software architects and engineers to develop a product population for mid- and high-range TV sets.

Almost every medium-sized or large software project involves some modeling, even if it is for example something as simple as a small UML use case diagram. MDD approaches can be used in every project and it is hard to say what makes a project suitable for such an approach. The opinions of the researchers and practitioners range from the idea that MDD is an absolute necessity for successful software development, to complete denial of modeling as a useful activity. It is usually up to the "taste" of the practitioners to decide whether or not to use MDD or its refinements (like AMDD and MDA) for a given project. It is however reasonable to suggest that the need for software models is proportional to the complexity and size of the target application, since MDD targets to reduce design complexity in first place.

4.2 Requirements

4.2.1 Requirement specification

Unclear and ambiguous requirements are among the main risks for CBSE [15]. In order to select or design and implement the appropriate components it is a prerequisite that the correct functional and nonfunctional requirements are known. A component based technology may provide a way for modeling requirements. However, this is often not the case and most such technologies do not provide any mean for the requirements analysis of the whole system being developed. Thus CBSE is usually augmented with techniques for requirement engineering.

On the other hand MDD provides suitable ways for the modeling and specification of requirements. MDD tries to tackle development complexity by raising the abstraction level

higher than the level of technical details and closer to the application domain. This allows expressing the requirements easily in a way which is close to the application domain.

When using MDD the initial models usually represent a formalized expression of the requirements. These models are then transformed multiple times until a fully functional system is achieved. An outstanding example for a modeling technique that allows specifying requirements are the UML use case diagrams. They are used to model the functionality provided by a system and are widely used in practice for business and requirements analysis. Other notable techniques used for modeling requirements are UML activity and state machine diagrams and BPMN [33].

4.2.2 Changing requirements

CBSE focuses on composition of components code and thus it is mainly a bottom-up approach. However, a requirement change usually imposes a "top-down" change in the software, meaning that a requirement change may imply architectural and design changes which then imply implementation changes. Such changes are hard to implement in bottom-up approaches since they do not provide traceability between requirements and implementation artifacts like components. Thus after a requirement change is introduced an analysis should be carried to evaluate which components should be replaced or adapted.

Another major problem with changing requirements in CBSE is the cyclical requirement-component dependency (CRCD) introduced by Tran et al. [34]. In brief, this cyclical dependency constitutes of the following:

1. **Reexamination of a set of previously selected components.** This usually happens after the creation or modification of a system requirement. In the worst case, the reexamination leads to the evaluation and selection of new components, which is a lengthy and laborious process.

2. **Reexamination of associated requirements.** The replacement of a set of components often leads to lack of proper support for some existing system requirements implemented by them. Such problems are usually discovered during the integration or at the worst during later phases. These problems lead to reevaluation and re-negotiation of these system requirements to ensure timely delivery, which then leads to the reexamination of a set of previously selected components (step 1).

Tran et al. suggest strategies for the mitigation of the effects of CRDC like *Ensuring appropriate support for early component evaluation* and *Prioritizing system requirements*. Still CRDC remains a key problem in CBSE that has not yet been adequately addressed [34].

A system being developed by adhering to the MDD approach usually observes very good traceability between requirements and implementation artifacts (e.g. source code). This is because each model, except for those created from scratch, is achieved by transforming others. Thus the relations between models representing system requirements and low-level models, which are close in nature to the implementation artifacts, are clear and it is straightforward to assess the impact and needed efforts.

Ideally, whenever a system requirement is added or changed then the models directly representing it should be modified to facilitate the change. After that the corresponding model transformations should be reapplied resulting in a new version of the system implementing the changed requirements. Thus the actual efforts needed for the implementation of a requirement change depends on the level of automation of the used transformations. If all transformations are fully automatic then only the changes in the initial models need to be manually applied. On the other hand, if no suitable transformation tools are available then there might be a lot of efforts needed for the propagation of a change to the low-level models. Hence the ease of implementing a requirement change depends on the quality and the level of automation of the available tools for model transformations.

4.3 Design

4.3.1 Separation of concerns (SoC)

Separation of concerns (SoC) is a paramount concept in software engineering. The introduction of the term is often attributed to Dijkstra [35], even though the ideas of SoC have been incorporated in software engineering before that.

In essence, SoC in software engineering refers to the separation of a software system into ***elements*** that correlate with each other. Ideally, these elements should have exclusivity and singularity of purpose, meaning that no element should share in the responsibilities of another or encompass unrelated responsibilities. The area of interest with which one of these elements deals is called a ***concern***. Implementing SoC leads to more stable, extensible and

maintainable systems which observe less duplication and allow for reuse of modules across other systems [36]. SoC is usually achieved through ***boundaries*** delineating a given set of responsibilities. Some examples of boundaries for the definition of core behavior would include the use of methods, objects, components, and services [36].

Clearly in CBSE the software elements (building blocks), as defined by SoC, are the components. Component interfaces can be seen as the boundaries between the components, delineating components' responsibilities and functionalities. CBSE allows software engineers to enforce SoC in a much more coarse-grained way than third generation programming languages do – by functions, classes etc. Thus components are usually subjects to SoC themselves, meaning that they are further divided into functions, classes or subcomponents.

The main objects of work in MDD are the different models of the system being developed, but not modules constituting the system, as it is in CBSE. Thus SoC elements and boundaries in MDD are not that apparent as they are in CBSE. Actually MDD does not define explicitly how SoC is achieved. Once again approaches, augmenting the core MDD ideas have been defined for the purpose. These approaches implement SoC over the models of the target system rather than the system itself. An example for this is the MADE approach proposed by Beydeda et al. [19]. It suggests specifying individual concerns at all model levels including documentation, test code, and if possible deployment information.

4.3.2 Architectural support

Architecture of a software system can be defined as "*...the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them*" [37]. It is intuitive to liken components to the software elements and interfaces to the sets of their externally visible properties from the previous definition.

Software architecture and software components are complementary to each other. While the software architecture defines the elements (in case of CBSE - the components) that make up the system, it is the components that provide the needed properties and functionality. Moreover, software architecture focuses on the early design phases when the overall structure is designed to satisfy functional and non-functional requirements. On the contrary component technologies are focused on composition and deployment, closer to or at execution time. In

many component-based systems the architecture is visible even at runtime, since the system comprises the distinct components defined in the architecture [15]. Exceptions to this are technologies that perform code optimizations, which result in indistinguishable components at run time. Examples for this can be seen in the embedded systems domain - e.g. ProCom [11].

As mentioned a component technology may allow modeling of system architecture. The survey by Feljan et al. [38], reports on the properties of 22 component models, of which only 5 do not facilitate modeling. This comes to show that most current component technologies allow for architecture definition. However, there are no widely accepted component system modeling formalisms, which may eventually lead to vendor lock-in. Also most of the existing technologies only use specifically designed for their purposes modeling formalism thus constraining designers and architects in terms of expressiveness. In addition this reduces the ability to communicate architectural decisions since fellow designers, architects and developers may not be acquainted with these technology-specific formalisms.

The use of component technologies often constrains the architectural freedom. An architect might have only a limited set of available components to use. Thus the architecture should be defined in a way that uses only available components or custom made components, in case that the time and budget constraints allow for component development. Moreover, for many of the already existing components (especially for the commercial ones) the source code is not available and the APIs are not informative enough to foresee their actual behavior and potential integration problems. This complicates the tasks of evaluating and selecting appropriate components. Obviously, a successful architecture created under such constraints may not be possible for all cases.

MDD approaches are especially suitable for documenting architectures. Perhaps the most important concept related to architecture documentation is the ***view***. A view of software architecture can be defined as "*a representation of a coherent set of architectural elements, as written by and read by system stakeholders*" [37]. A model was previously defined as "*a description or specification of a system and its environment for some certain purposes*" [20]. These definitions come to show that architectural views are in fact types of models, since a view is indeed a specification of a system and its environment for the purpose of representing a system from the perspective of a given set of stakeholders. Thus it is not surprising that approaches from MDD are widely used for describing software architectures. Today the Unified Modeling Language (UML) [39] is considered as the de facto standard notation for

documenting software architecture, even though it has some shortcomings in this respect [37]. An outstanding example of how modeling can help in the process of architecture definition is the ArchiMate [40] modeling language, which has been designed especially for the purpose of modeling architectures. Also several other architecture description languages (ADL) have been created over the years, the most notable of which are AADL [41], Wright [42], Acme [43] and EAST-ADL [44].

4.3.3 Design for Extra-functional requirements

One of the main challenges before CBSE is the design and implementation of dependable systems. CBSE provides a limited set of means to specify and ensure the non-functional attributes of the components and the system under development as a whole. Crnkovic and Larsson argue that the majority of current component specification techniques do not address adequately the extra-functional properties [15]. The design of a component based system consists of the identification and selection (or in some cases development) of the needed components. Thus component evaluation based on the specifications (both functional and extra-functional) of the available components is at the core of component based design.

To use a component successfully in the context of a dependable system, information for extra-functional properties, such as performance and capacity is required. Specifications of such properties are usually heterogeneous, because the diversity of properties that might be of interest is unlikely to be captured adequately by a single notation. As mentioned one approach for specifying the performance is to give asymptotic estimations of the memory and time consumptions. A more general approach for specifications uses the so-called credentials. A credential is a triplet of the form *<Attribute, Value, Credibility>*. The attribute is a description of a component property, the value represents a measure of that property, and credibility is a description of the way the measure has been obtained. This information is usually part of the components' metadata and thus can be automatically inspected by different tools. This approach can be further extended so that credentials can be used for interfaces and operations as well. This is especially useful because the clients of a component can take into account only the extra-functional properties of its exposed interface. Thus the component can later be replaced with another one obeying this interface, without breaking the extra-functional needs of the client [15].

Unfortunately, the extra-functional properties of the whole system are not a result of the extra-functional properties of the components only. This applies to other types of properties as well. Moreover, combining components often results in system properties, which none of the components have. These are called ***emerging properties*** and are extremely difficult to predict and handle. It is still a subject of research how the attributes of the whole system can be derived from the attributes of the components which make it up and whether this is possible at all. Thus Crnkovic and Larsson argue that component evaluation should be replaced by component assembly evaluation [15]. A ***component assembly*** is an aggregation of components that provide integrated behavior. This leads to the suggestion that when designing, whole component assemblies should be selected instead of separate components.

MDD is a very general approach and whether or not extra-functional attributes are included in the models depends on the modeling formalisms and the style of modeling. In a survey Ameller et al. distinguish two groups of MDD approaches - approaches that support extra-functional requirements and ones that do not [45]. The survey also identifies that most current MDD approaches fall into the second category that is they do not support extra-functional properties at all. Most of the current approaches that do consider non-functional requirements use UML extensions especially designed for the purpose like MARTE [46].

4.3.4 Design & Architecture evaluation

CBSE itself contributes little to the process of design and architecture evaluation. Evaluation is a central part of component based design, since it consists of the selection of the needed components or better - component assemblies as already discussed. Thus it can be considered that the design and architecture of a component based system is a result of evaluation activities itself. This does not imply that CBSE designs do not need further evaluation and consequent elucidations. Such evaluations of component based designs are usually done by standard types of assessment which are not specific to CBSE. Examples for such approaches are stakeholder-based evaluation, expert-based assessment and quality-attribute assessment. Stakeholder-based assessment aims to validate that the trade-offs between requirements in the software architecture match the actual stakeholder requirement priorities. Expert-based assessment is about allowing a team of experienced architects and designers to assess the software architecture and to determine whether a system based on it will meet its functional and non-functional requirements. Quality-attribute oriented evaluation aims to predict

quantitatively a quality attribute (e.g., maintainability or security) [15]. It is typical that more than one of these approaches is used in combination in order to assure the quality of the design or architecture in question.

These evaluation techniques are also employed to evaluate designs when using MDD. However, the presence of models focusing on different aspects of a design or architecture can significantly help in the evaluation process. For example a visual model of the GUI can help for easier stakeholder-based assessment, since it makes it easier for an end user to imagine what the final system will look like and how user friendly it will be. Also different models focusing on technical specifics of a design (e.g. a concurrency model) can help for communicating ideas when performing expert-based assessments.

Another benefit with respect to system assessment that MDD provides is the evaluation of some models based on existing theoretical achievements. As an example an Entity-Relationship (ER) model can be evaluated by checking if it complies with some of the relational data base normal forms [47]. Also object oriented models can be evaluated through a set of specialized metrics as proposed by many authors including Chidamber, Kemerer [48] and Martin [49].

4.3.5 Design reuse

CBSE is focused on component reuse. In order to reuse a component or a component assembly the component design context must be shared among the potential applications. Thus design reuse must happen before component reuse. Moreover, if architectural design is reused a greater component reuse can be achieved, since the architecture of an application defines the allocation of functionality to components and the interactions among them [50]. That is once architecture or a design is defined it can be reused in similar contexts resulting in the reuse of the previously selected components. Unfortunately, many component based technologies define their own formalisms for architecture and design description hence hindering the straightforward reuse of both design and components across different technologies.

The central items in MDD are the models and thus reusing them can be very beneficial. Obviously, if models can successfully be reused then architecture or a design can also be reused since in MDD these are represented as sets of interrelated models. A major problem

before the reuse of models is the abundance of modeling formalisms. A model defined in a given formalism is hard to be reused in another system which uses others. One reason for this is that it is impractical to use a huge set of formalisms because the software engineers would need to have knowledge in all of them which is not always possible. Also using too much formalism for different purposes makes a system harder to comprehend and analyze contrary to the main goal of MDD to reduce development complexity. Thus it is a good practice to employ a few popular modeling standards so that the created models can easily be reused in other systems built around these standards. Examples of such are the previously mentioned UML [39] and ArchiMate [40]. In that respect a notable standard is XML Metadata Interchange (XMI) [51] which allows metadata exchange of meta-models expressed in MOF [23]. A typical use case for XMI is to transfer UML models between different vendor tools and that way to facilitate reuse.

4.4 Development

4.4.1 Required tools, technologies and their maturity

As discussed a component framework is a concrete technical solution, which allows components compliant to a given component model to work together. It is what allows components obeying a given component model to communicate with each other and to have their code executed. It is the only technical solution which is needed for component base development.

A key consequence from the previous definition of component framework is that it can work with any component, provided that it is compliant to the specified component model. Unlike component frameworks, component models are not concrete technical solutions. They are specifications or more precisely standards and conventions for a component. Thus components can be developed agnostically of the exact technical environment (component framework) in which they will be executed, provided that they obey a predefined specification (component model). Such components can then be deployed into any component framework supporting this predefined component model.

This allows practitioners employing a widely used component model to switch easily between component frameworks and thus to avoid vendor lock-in. An outstanding example for this is OSGI [26], which is a public specification of a component model. Different competing

vendors provide the so-called OSGI servers, which represent component frameworks capable of "running" OSGI compliant components (a.k.a. bundles). Thus once an application obeying the OSGI specification is built it can then be executed by any of these servers and a vendor can be switched at any moment. The same idea of a public specification implemented by competing vendors is behind the EJB specification. However, if the component model and framework come from a single vendor (as is often the case) then vendor lock-in is very likely due to the lack of alternative component framework providers.

Component frameworks have been constantly evolving and improving their quality over the last decade and thus they can be considered rather mature from technical point of view. However, most of them do not implement some of the theoretical achievements in the area (e.g. extra-functional specification). Another flaw is that while there is an abundance of component technologies at disposal when using a main stream programming language like Java or C# it is very hard to find suitable solutions when using some not so popular languages. This is to become less of a problem with the advent of the JVM and Microsoft's Dynamic Language Runtime (DLR) as platforms for executing programs developed in different languages.

Reconsidering the classification of development approaches with respect to modeling from section 3.2 it is obvious that tools are needed in almost all existing model based approaches. This classification defined five groups of approaches - *code only*, *code visualization*, *roundtrip engineering*, *model-centric* and *model only*. The approaches that fall into the *code visualization* group usually rely on tools that extract and visualize diagrams based on the source code, since the manual extraction and drawing of diagrams results in an unacceptable waste of resources. The approaches that fall into the *roundtrip engineering* group need tools that generate source code (or at least code stubs) correspondent to an abstract model and update this model to reflect changes in the code. Approaches falling into the *model-centric* category rely on tools for the generation of the source code from the detailed models.

The approaches from the *code only* category do not rely on modeling tools which coincides with the fact that they can hardly be considered MDD approaches. The *model only* approaches may rely on tools for modeling even though this is often not the case, since their purpose is only to help for better understanding of business requirements, discussion, communication etc. It is arguable whether the *model only* approaches can be considered MDD approaches,

since the used models are usually informal and are not used in the actual implementation of the target system.

This comes to show that most of the models created when following a MDD approach require the use of specialized tools. The abundance of different models and modeling formalisms can easily lead to vendor lock-in. Just like in CBSE this can be avoided by employing widely used modeling standards for which there are tools provided by different vendors. An example for this is the previously mentioned standard XML Metadata Interchange (XMI) [51] which allows exchange of meta-models expressed in MOF [23]. Thus any model which can be expressed in MOF (e.g. a UML model) can be exported to an XMI descriptor and then imported by another tool. This allows modeling tools by a vendor, providing MOF-based models and supporting export to XMI descriptors to be replaced by tools from another vendor that can import these descriptors.

Tools that allow modeling in well established formalisms like UML and Entity-Relationship (ER) diagrams have been put into practical use for more than a decade. There are dozens of competing vendors of such tools, which reflects the wide adoption of these standards by practitioners. As a result some of these tools are very mature and elaborate. Examples of such tools are ones for UML reverse engineering, code stub generation based on UML class diagrams, generation of relational database schemas from ER diagrams etc. Such tools are so wide spread that they even come out of the box with some integrated development environments like MS Visual Studio.

4.4.2 Verification and Validation (V&V)

Verification refers to the process of assuring that a software component or a system meet their specification. Validation refers to the process of assuring that a component or a system fulfills its intended use when placed in its intended environment [52].

As mentioned in CBSE there is a distinction between *component development* and *system development*. The same distinction can be applied to the processes of verification and validation. Approaches for component verification are very similar to those for the verification in traditional custom software development – code inspections (manual or automatic) and tests (e.g. white box or black box ones). Reusable components must be thoroughly specified and verified because they are to be used as "black boxes" in a variety of

situations and configurations, many of which not obvious at the time of development. A bug in a component can very easily break its contracts with other components and as a consequence to lead to a failure of the whole system. Thus often software components are certified, in order for their clients to have confidence when using them. Certificates are issued by independent organizations or by applying some standard component certification procedure within the developing organization. An indirect way to certify a component is to certify the employed software development process by using a suitable standard (e.g. ISO 9000). However, the component quality cannot always be inferred from the process quality [52].

Component validation is a very difficult task, since the exact environment of the component cannot be foreseen. Hence component validation is usually done in the context of the target system by the engineers assembling the components. However, most components are black-box and their APIs are not informative enough. Thus often integration testing is the only way for component validation. On the contrary, component system validation is just like the validation of traditional custom systems, since the end users should not be aware of the internals of the system but only its functionality and quality.

Ideally, component system verification should require fewer efforts than the verification of traditional custom developed systems, since the selected components must have been verified (and maybe certified) beforehand. However, the glue code developed to integrate the components should be verified as well. Also the whole assembly of components should be verified for the presence of undesirable ***emerging properties***. The benefits of the easier verification are often diminished by the more efforts required for identifying and correcting mistakes. If a fault in the system occurs it usually cannot be found and corrected by traditional debugging techniques, since most of the components' source code is typically not available [52]. The most that can be done is to locate the components causing the problem and then to try to work-around it in the glue code.

In MDD the usage of many models representing a system from different viewpoints allows for earlier identification of design flaws. This is a major advantage of MDD over other approaches, since design time errors are very costly if found late after series of other design decisions have been taken and a significant amount of development has been done.

Another major benefit of applying a MDD approach is the ability to easily implement model-based (or model-driven) testing. Its main idea is to use a model of a system as the basis for the automation of its testing. Such models can be created exclusively for the purpose of testing, but usually development models are used directly or are modified for the purpose. The testing model should describe all aspects of the testing data [53].

Model-based testing usually has four stages [54]:

1. ***Building an abstract model of the system under test***.

2. ***Validating the testing model***.

3. ***Generating abstract tests from the model***. This step is usually automated via specialized tools, but the test engineers can control various parameters to determine which parts of the system are tested, how many tests are generated etc.

4. ***Refining the abstract tests into concrete executable tests***. This step is usually performed automatically, but is also under the control of the engineers. These transformations/refinements usually use concrete code templates for each abstract operation from the model. The resulting concrete tests can be executed on the system under test, in order to detect failures.

Reid [53] emphasizes that there are types of applications and application properties which are not suitable for such testing. For example some non-functional requirements, such as usability may not be well tested for the time being since there is still no consensus how to adequately represent such requirements in a model. However, for most applications the use of model-based testing pays off by a lower maintenance costs when the system is operational despite the large up-front costs [53].

Apart from model-based testing MDD approaches provide other ways of easier system verification and validation as well. In case that models expressed in a mathematical formalism (e.g. the Z notation [55]) are present it is possible to create tools that automatically prove the correctness of the models. Approaches implemented in such tools include automated theorem proving and model checking. However, having models expressed in such formalisms often requires efforts and software engineers with appropriate qualification that are not usually available. Some models can also help in the process of validating the system if they represent

it in a summarized way so that the end user can see that its structure meets the stated requirements.

4.4.3 Traceability, understandability and maintainability

CBSE itself provides no means for the better traceability and understandability of the components and systems built by (re)using these components. As discussed earlier changing requirements are one of the main problems before CBSE because it does not provide traceability between requirements and components. To achieve traceability of the changes in a component based system a rigorous documenting approach must be taken, which in many cases requires an unacceptable amount of resources. The understandability of component based systems also depends on the comprehensiveness and quality of the documentation and the architecture. A major threat to the understandability and maintainability of a component system is the quality of the glue code. It is usually complex and "messy" since it needs to juggle between the different requirements of the component interfaces. Hence the statistics that effort per line of glue code is about three times the effort per line of the application's code [15].

CBSE is young and little is known about the long term maintainability of component based systems [15]. It is however intuitive to suggest that there are two major factors that influence the maintainability of a component based system - the quality of the maintenance service of the used components and the maintainability of the glue code.

A component maintenance service is usually provided by the organization which originally developed it which may be external and may not be under the control of the organization developing the whole system. In case that this maintenance service is not flexible enough to provide fixes for issues in the components within an adequate timeframe a lot of glue code may be needed to work-around these problems. Also maintenance problems may arise if a component's maintainers break (purposefully or not) its backward compatibility. This may cause the clients of that component to stick to an earlier version (putting up with its shortcomings) or to modify the glue code so that the new version can be used.

The quality of a component system’s glue code is a maintenance factor since it is the only tool that the system developers have to bind together the used components and to work-around their shortcomings in terms of interfaces, functionality and quality. Thus this code usually

includes a lot of "hot fixes" and utility functionality making it hard to understand and maintain.

Following a MDD approach the target system is represented as a network of models, related to each other by transformations. Typically some of these models are high-level and represent functional and quality requirements for the whole system, while others are low-level and can be used for the generation of the final source code. This makes the software much more traceable and maintainable since each source code module can be traced back to the design entity which it represents and vice versa. The models which are at different levels of abstractions can also serve as documentation and thus a new team member can easily see the correlations between the requirements, the architecture and the implementation artifacts. The maintainability of a system can also be improved if the transformations from models to source code are fully automated. In such systems working with the source code is not needed and only the abstract models need to be maintained, which is presumably easier.

The incorporation of modeling in the development workflow can sometimes hinder traceability. More specifically models are rarely created at once and then used without any further changes. Usually models evolve over time and thus they should be managed by a versioning system, which makes the ability to compare and merge models essential. However, many models are not stored in text format only and may include graphical views, forms, dialogs, and property sheets. These are difficult to compare and merge, since visualizing the differences between the versions in a usable way is difficult. Appropriate tool support for model versioning, comparison and merging has been identified as a major challenge and success factor for MDD [56].

4.4.4 Dealing with legacy code

The main technique for coping with legacy code in CBSE is to encapsulate it into a component or a set of components [15]. Since components interact with each other only through their interfaces, the fact that a component is implemented by legacy code is not visible to its clients. Thus the legacy code is isolated and does not propagate complexity to other parts of the system. At some point the legacy components can be replaced with new modernized versions obeying the same interfaces, without breaking the established contracts with the environment. A major problem with this technique is the restructuring of the legacy code into components. One approach for this is to create a single legacy component that

encapsulates all the legacy code and exposes its functionality. Such a component is usually very big and contains unrelated to each other functionalities, making it hard to maintain and modernize. The alternative approach is to break the legacy code into smaller and more maintainable components, which are easier to modernize. However, such a breakdown usually requires in-depth knowledge of the legacy code by the developers of the component system, which is often not the case.

Following a MDD approach, in order to use legacy code in the models it is required that an abstraction of this legacy code is created. This abstraction should be at the same level and should use the same formalisms as the models interacting with it. Thus, when modeling with legacy code reverse engineering tools that allow the extraction of abstract models of the legacy code are required. Usually the models representing the interaction with the legacy code are not high-level since the domain modelers should be abstracted from such implementation details. Models representing such interactions are usually low-level (close to the code) or represent the overall technical design of the application and include direct calls to the abstractions of the legacy code.

An alternative approach is to skip the modeling of the legacy code and the interactions with it. This can be achieved by custom model-to-code transformations that replace predefined modeling constructs with actual calls to the legacy code [18]. However, such an approach requires highly customizable or even custom developed code generation tools.

4.4.5 Code reuse

The paramount purpose of CBSE is namely code reuse. However, following a CBSE approach does not mean developing without coding. Components that are business critical or unique to a specific system are typically developed by the system developers [15]. Also the quality or the price of existing components that provide needed functionality may drive for custom development of alternatives. Even though custom component development is often it does not mean that it does not pay off in the long run. It is often the case that an organization develops a set of similar products - a product family or product population. Thus these custom components can often be internally reused.

Besides custom development another factor diminishing the result from code reuse in CBSE is the glue code. As discussed developing glue code is a costly and error prone task. In case

that the selected components or the component model are inappropriate or if the components are not well understood, the cost of glue code may be greater than that of the development of the components themselves [15].

Unlike CBSE, MDD is not focused on source code reuse per se. The central notion in MDD is the model and thus it is the main goal for reuse. Models can be successfully reused (fully or partially) across multiple systems. As a side effect this may lead to reuse (full or partial) of the code correspondent to these models.

4.5 Organizational specifics

4.5.1 Development methodologies

Currently there are no theoretically refined and well established in practice methodologies for CBSE. Thus already existing software development methodologies are adapted for the purpose. However, there is one major difference between CBSE and general purpose software development – CBSE covers both component development and system development based on the selected components [57]. In CBSE it is considered that the development methodologies for these two are distinct since they include different activities which can be performed to some extent independently of each other. Besides, some component technologies (e.g. Sun's EJB) introduce new roles in the development process. These roles are not accounted by traditional development methodologies. Thus many companies are afraid to take up CBSE since it is still not clear how exactly existing engineering processes should be modified or replaced and what will be the cost of such a modification/replacement [58].

The component development process is similar to standard system development process. A major threat for component development is the requirements instability. Reusable components can be developed more easily if requirements remained constant during the development time of both component and system. However, the more reusable a component becomes, the more demands are placed on it [15]. Thus the component development methodology should specifically focus on the requirements management and should allow new component versions to be released more frequently. Thus an iterative evolutionary approach is advisable for the component development.

In the component-based system development process the emphasis is on the reuse of preexisting components. Hence it includes activities like identification and selection of appropriate components. Thus the standard established lifecycle models should be modified to emphasize on such component-centric activities [57].

MDD is much more mature than CBSE in terms of development methodologies. Parviainen et al. report on 16 known processes and methods developed for projects using MDD [56]. Among these are ones which are based on the waterfall, iterative, incremental and RUP processes and the agile version of MDD - AMDD. This comes to show that the MDD approach is applicable in almost all current methodologies.

4.5.2 Organizational requirements

The survey by Bass et al. reports on the main inhibitors that early adopters of software component technology have [58]. Figure 5 summarizes its results.

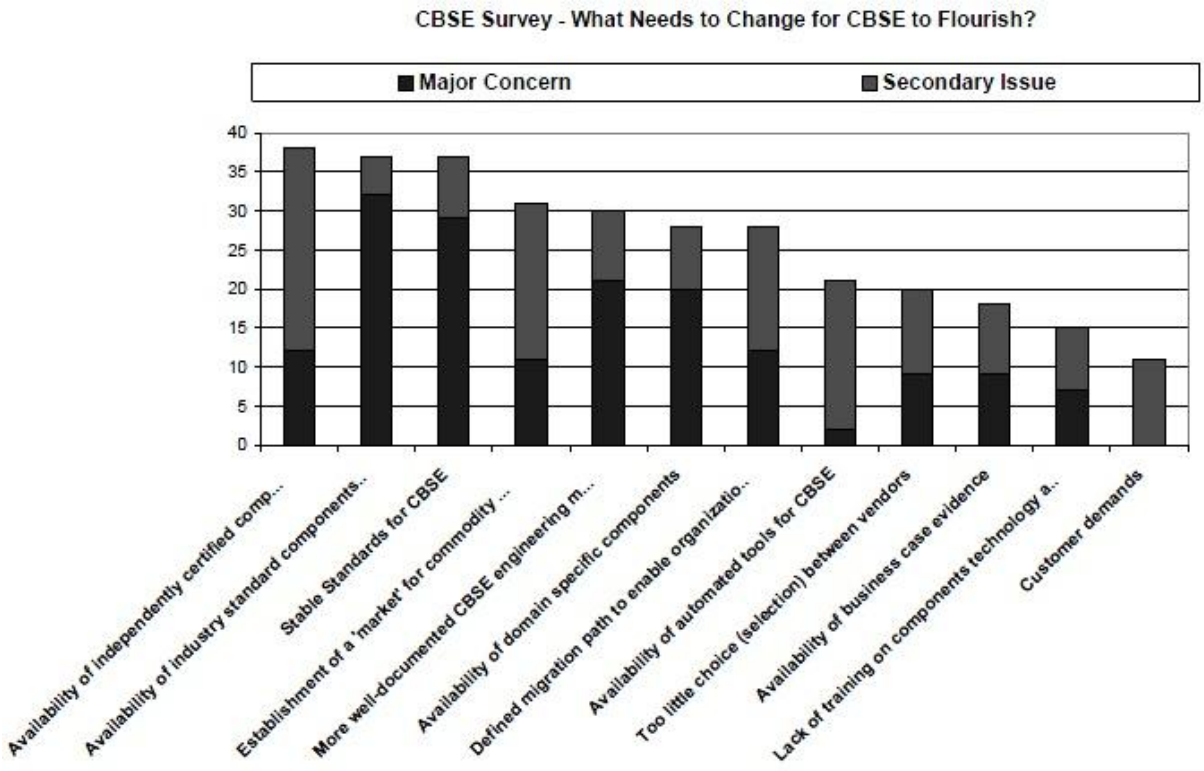

Figure 5 Statistics from an online interview on CBSE inhibitors by SEI [58].

Based on the results of this interview the survey by Bass et al. [58] concludes that the main threats for successfully incorporating CBSE techniques are:

- Lack of available components;
- Lack of stable standards for component technology;
- Lack of certified components;
- Lack of an engineering method to consistently produce quality systems from components.

The lack of available and certified components makes custom component development inevitable. In order to be profitable a software component must be reused at least three times [15]. Thus for the time being CBSE is only profitable in case that a set of products with overlapping functionality (e.g. product family or product population) is developed. Organizations which do not develop such sets of related products or do not have at their disposal a substantial set of appropriate components should avoid adopting CBSE for the time being. Such organizations may turn out to be spending too many resources on the development and maintenance of components that are never reused.

The lack of stable component standards is a problem because it may lead to vendor lock-in as discussed in section 4.4.1. However, the survey by Bass et al. [58] dates back from 2000 and since then specifications and technologies like OSGI, EJB and the .NET component model have matured and have become de facto standards for component development in many domains. As discussed with the advent of the JVM and Microsoft's Dynamic Language Runtime (DLR) as platforms for executing software developed in different languages these component technologies are becoming available for software written in languages other than Java and C# as well.

The lack of well established component based engineering methodologies brings a great risk to enterprises adopting CBSE. Such enterprises should consider developing custom modifications of their internal processes in order to accommodate CBSE. Almost all in-house engineering processes should be reformed to accommodate CBSE adoption. This requires substantial organizational maturity and insight for development processes.

The report by Parviainen et al. describes the results of a survey carried out in various organizations about their experience with MDD [56]. The major problem identified is the lack of modeling experts within the organizations. Indeed an enterprise adopting MDD should be

prepared to invest in internal specialized courses until a core group of modeling experts is trained. Such courses should be carried out regularly at a predefined period of time (e.g. annually) in order for these experts to keep up with the latest trends and for newcomers to catch up with the others.

Another important problem identified in the survey is that it is difficult to select suitable tools and techniques and adapt them in different environments. In parts this problem may be seen as a consequence of the previous one. That is if the competence of the modeling experts is increased, most likely they will be able to select suitable tools and techniques and make use of them. Another reason for this problem might be that some existing types of modeling tools are not mature enough and thus modeling experts should be capable of evaluating their maturity before putting them into practice.

4.5.3 Team member qualification

As discussed, CBSE is understood in different ways in academia and in industry. This can be attributed to the fact that most practitioners think of a component simply as a large reusable piece of software that does not necessarily have well-defined interfaces. Thus when adopting CBSE an organization should make sure that its technical staff is trained so that it understands the fundamental notions and ideas of CBSE. Fortunately, the paramount concepts in CBSE are just a few and can be presented shortly (as demonstrated in the Overview section). From technological point of view software engineers should get acquainted with the component framework, component model and the interface specification technique of the concrete technology that is to be used. This implies learning curve which can be rather steep for some technologies.

A major problem for the staff of an enterprise adopting CBSE is the previously mentioned unavailability of proven in practice suitable development methodologies. This poses a challenge for both technical staff and management since they have to define together adequate new component-aware processes. Typically this requires careful analysis and modification of already existing processes in the enterprise so as to leverage previous knowledge and investments in methodology establishment. A sensible approach is to devote a non-critical for the organization pilot project to be used for real-life experimenting before the in-house component-aware processes are defined [13]. In case this is not possible or the adopting

organization lacks sufficient in-house expertise in development methodologies to use as a basis, some external coaching may be needed to give a start to the process.

When adopting MDD the steep individual learning curves of the modeling experts are a major problem [56]. Most modern modeling formalisms are themselves very complex and require time to master - for example UML 2.2 defines 14 different types of diagrams. Moreover understanding the syntax of the formalisms and being able to model successfully a domain are two different things. Real-life modeling expertise is gathered mostly through coaching/mentoring either by an external organization or experts within the enterprise. Last but not least many existing tools also add to the complexity of employing an MDD approach. Such tools are usually inherently complex and have overburdened GUIs as a result of facilitating modeling in complex and mostly visually represented formalisms.

4.5.4 Financial issues

Generally speaking the tools and technologies for the main stream component technologies are not expensive. Moreover there are many open source tools including two popular implementations of the OSGI specification - Apache Felix [59] and Eclipse Equinox [60] and a few implementations of the EJB specification. The .Net component model is also available out of the box when developing in Microsoft .NET. Thus when adopting CBSE the main expenses are not for buying technologies and tools. As to Herzum and Sims the main costs are due to organizational learning curve and to the immaturity of current component-based technologies [13]. As to Bass et al. namely the lack of appropriate training on components technology is one of the inhibitors for CBSE adoption [58]. Thus an enterprise adopting CBSE might have to spare significant amount of in-house resources for experimenting and research.

As mentioned CBSE is a new approach and thus there are still not enough real-life examples of enterprises adopting it. Hence it is hard to foresee the period of time after which the investments in adoption would pay off. Moreover this period is very much dependent on the type, size and culture of the organization. Herzum and Sims estimate that in the late 90s the typical transition time for successfully moving to CBSE is between 1 and 3 years [13]. Given the advances in component technologies in the last decade, it can be suggested that the transition period is now significantly shorter.

Unlike CBSE, when it comes to MDD approaches appropriate modeling training is now available for many of the most widely used modeling approaches. Many universities provide courses in UML and BPM as part of their syllabi. Companies specialized in training and consulting often offer in-house training and coaching. However, this does not make a modeler's learning curve significantly less steep, since often modeling formalisms and tools are themselves complex. There are many free modeling tools for the most popular modeling formalisms like UML. However, most of them can only be used to create models as sketches and are not suitable for managing multiple models and complex transformations. Thus most enterprises adopting a MDD approach resort to commercial tools like Borland Together [61] or IBM Rational Rhapsody [62]. Alike CBSE there are still not enough practical examples of enterprises adopting MDD approaches so as to evaluate the needed period for the investments to return.

4.6 Comparison results and analysis

The following table summarizes the results from the previous detailed discussion:

	CBSE	**MDD**
Business specifics		
Drivers	Component reuse	Work at a higher level of abstraction than code. Complexity management.
Maturity	Relatively mature. Some theoretical achievements still have not made it into practice. Many practitioners still misunderstand basic concepts.	Controversy about real-life usefulness of modeling. Not yet proven in practice, though some types of modeling have been a basis for success stories.
Target system specifics	Suitable for all kinds of systems, except for systems with high demands for extra-functional requirements. Research for application in such system is under way. Especially useful for product lines.	Modeling can be applied in all kinds of systems. Most useful in large and complex ones.
Requirements		
Requirement specification	Most current technologies do not allow requirements specification, though CBSE relies on well specified requirements.	Some models are very useful for requirement specification - UML use cases, BPMN etc.
Changing requirements	Changing requirements pose great risks - hard traceability, cyclical requirement-component dependency (CRCD)	Allows adequate reaction to such a change, since the software is more traceable. If transformations are automatic changes can be applied very quickly.
Design		
Separation of concerns (SoC)	SoC is implemented through components and interfaces.	Does not define explicitly how SoC is achieved.
Architectural support	Different technology-specific formalisms for defining architecture and design. Lack of popular standards for component system modeling. Many technologies do not facilitate modeling at all.	Suitable for architecture and design specifications. Special kinds of models are used - ArchiMate, some UML models, ADLs
Design for Extra-functional requirements	Approaches for specifying component extra-functional properties (e.g. Credentials) are still not widely used. At design time component assemblies must be evaluated for such properties instead of components.	Most models are agnostic of extra-functional requirements. Most of those which are not use UML extensions especially designed for the purpose.
Design & Architecture evaluation	Does not help in architecture/design evaluation. Standard evaluation approaches are used.	Standard evaluation approaches can be used. Some models help when discussing design/architecture properties.

Design reuse	Some component system models can be reused. This is hindered by the lack of popular and standard component modeling formalisms. Actually CBSE relies on proper design reuse beforehand so that correspondent components can also be reused.	Some models can be reused in case the "donor" and "recipient" systems use the same formalisms. Thus using modeling standards like UML is a best practice.
Development		
Required tools, technologies and their maturity	The only kind of tool needed - component framework. Currently CBSE tools are considerably mature, though some theoretical achievements have not been widely incorporated yet. Vendor lock-in can be avoided by using implementations of public component model specifications.	Many modeling tools are required. If a popular modeling formalism is used (e.g. UML) there are mature tools available. Vendor lock-in can be avoided by using tools that can export models to a popular exchange formats - e.g. XMI.
Verification and Validation (V&V)	Components' quality is crucial and they should be heavily verified and in many cases certified. Component based systems are easier for verification since a lot of the work has been done at the component level. Their validation is similar to that of standard systems.	Models allow for early analysis so that design flaws are caught early. Model based testing can make verification easier. Having models expressed in a mathematical formalism allows automatic verification.
Traceability, understandability and maintainability	Depends on the quality of the documentation and the architecture. Maintenance also depends on the amount and quality of the glue code and the maintenance support of the components.	High traceability and understandability. Maintainability is usually also high, especially if the transformations are well automated. Problematic model versioning, comparison and merging.
Dealing with legacy code	Legacy code is encapsulated into designated components.	Legacy code must be "brought" to the level of the models, so that the interaction with it can be modeled. Reverse engineering tools are usually used for the purpose.
Code reuse	Reuse is the paramount idea. Custom components still need to be developed, but they are typically reused in-house later. Reuse benefits can be diminished by developing too much glue code.	Not focused on code reuse. Code reuse may be a side effect of model reuse.
Organizational specifics		
Development methodologies	Lack of real-life proven development methodologies. Existing methodologies are customized to facilitate CBSE specific activities, roles etc.	Can be used with almost all existing development methodologies, with minor adaptations.
Organizational requirements	Adopters must evaluate their eventual ROI since custom development is imminent. Adopters must have substantial in-house expertise about methodologies, so that new CBSE-aware ones can be defined.	Employees' qualification for modeling is a great concern. Eventual ROI must be evaluated having in mind the enduring trainings, coaching etc.

Team member qualification	Staff should be trained in the basic theoretical concepts of CBSE and the specifics of the selected component framework and model. Some coaching or a low-priority pilot project should be used for gathering hands-on experience.	Steep individual learning curve due to complex formalisms and tools.
Financial issues	Many free tools are available. Steep learning curve resulting from lack of appropriate know-how and training, contributes substantially to the high cost of CBSE adoption.	Steep individual learning curve, though appropriate training is often available. Enterprise modeling tools contribute to the price of adoption as well.

Table 1 Summary of the comparison discussion.

Both MDD and CBSE are relatively mature approaches. There are certain technologies and tools based on these approaches which are popular among practitioners and there have been many success stories using them. However, some of the theoretical achievements of CBSE have not yet made it into practice and the benefits of MDD are doubted by some practitioners. Both MDD and CBSE are general purpose approaches since they have been employed successfully in almost all kinds of projects. Then again CBSE is still not the method of choice when developing systems with high demands for extra-functional requirements, though there have been promising researches and implementations of component models and frameworks for the purpose. CBSE is especially suitable for developing product lines.

Current MDD technologies are better for requirement specification than current component based ones. This is clearly something that CBSE technologies can borrow from the MDD ones, which define several popular formalisms for this purpose. Indeed, integrating requirements modeling support into a component based technology may lead to better incorporation of requirements gathering activities in the component based development process. This can lead to better traceability between requirements and components implementing them hence making CBSE more agile to requirements changes. Namely changing requirements are a major vulnerability of CBSE since it is hard to estimate the impact of such a change and because of the so-called cyclical requirement-component dependency (CRCD). On the contrary MDD approaches allow for faster reaction to requirements changes, since a system developed this way is usually more traceable. Moreover, if model transformations are automatic, such changes can be addressed very quickly.

When it comes to design and architecture current component based technologies are again less powerful than current model based ones. Many CBSE technologies do not allow modeling at

all. CBSE theory is not concrete about what are the required properties for appropriate component based modeling formalisms. Component based technologies that do support modeling typically use their own formalisms thus making the developed models hard to reuse in another environment. Moreover, having a single modeling approach at disposal limits system architects and designers in terms of expressiveness. It would be really beneficial for software architects and designers if component based technologies accommodated simultaneously several different formalisms and allowed for transformations between models of different types, just as it is in MDD. Also there is a great need for popular component system modeling standards and exchange formats (like XMI) in order to facilitate model exchange across tools. Actually these are the features that make current MDD technologies superior in terms of architectural support, design & architecture evaluation and design reuse. Both CBSE and MDD have their approaches for designing with respect to extra-functional requirements, though none of them is widely used in practice.

In general, CBSE requires fewer tools than MDD. Actually CBSE requires only a component framework, while MDD may require multiple tools for the different used formalisms. Vendor lock-in can be avoided in both approaches if implementations of public specifications are used. MDD promotes the usage of models for early evaluation of system design, which can result in reduced costs because major design issues can be addressed early. Another benefit of using models is model-based testing, which in many cases makes verification easier. Component based systems have one major advantage when it comes to verification - components are usually heavily verified before they are used. This reduces the needed efforts for system verification, though glue code and emerging properties still need to be verified. On the other hand the verification of the components themselves is a laborious task. Components must be heavily verified before they are used in practice, since a malfunction of a component may have dramatic consequences to the whole system.

Both approaches have their advantages when it comes to verification and it is reasonable to look for ways of combining them. Obviously, if suitable modeling is incorporated in component based system development this will allow for early verification. Also model-based testing of component based systems is a viable research domain. Since component based modeling is focused on representing components and their interactions, model-based testing seems a reasonable approach for creating integration tests, which verify that the whole component based system behaves as expected. More specifically model-based integration testing can be used to verify the correctness of the glue code and the presence of undesirable

emerging properties, which are the factors contributing the most to the verification costs of component based systems.

With respect to system traceability, understandability and maintainability MDD can be considered superior. In parts this can be attributed to the fact that MDD is a top-down approach, while CBSE has many bottom-up features. Hence MDD provides better traceability between the initial requirements and the design and implementation artifacts. Once again component based approaches can be enriched with suitable modeling in order to facilitate similar traceability and as a result to increase system understandability and maintainability as well. CBSE is a superior approach for reusing code and integrating legacy code. In MDD code reuse can be a side effect of model reuse and dealing with legacy code often requires specific tooling. MDD can benefit from modeling formalisms that explicitly allow representing encapsulated legacy components. This would allow the interaction with such components to be represented in the different models, thus facilitating easier handling of legacy code.

Component based development poses a great challenge in terms of development methodologies, since it implies activities and roles not present in conventional development. The major costs for an organization adopting CBSE are the investments for adapting existing in-house engineering processes and the trainings of employees with respect to these refined processes. Contrarily, a major inhibitor for adopting MDD is the steep individual learning curve, since many of the modeling formalisms and tools are very complex. This should be taken into considerations when combining both approaches. Incorporating appropriate modeling into component development can indeed lead to multiple benefits, as it was mentioned. However, incorporating complex modeling approaches in component based development can make the individual learning curve steeper and thus can increase unacceptably the costs of adoption. Hence, modeling approaches incorporated in the development of component based systems should be of limited complexity in order to make the overall approach feasible for adoption.

5 Background of the comparison with respect to ProCom

The goal of this section is to pave the way for the comparison of CBSE and MDD with respect ProCom, which is the second main goal of this thesis. Thus the section starts with a brief introduction to ProCom and its properties. Then a comparison method derived from the previously defined general one is briefly described and motivated.

5.1 Overview of ProCom

Embedded system development has always been a difficult task due to the restrictions in available resources (power, CPU and memory) and harsh requirements in terms of safety and dependability. The growing demands for more functional, scalable and distributed (comprised of subsystems communicating over a network) embedded software makes the development of such systems even more complex. Thus it seems natural to look for ways to mitigate the complexity and reduce the costs of the development of such systems by employing a component based approach. There have been several component based approaches for the development of embedded systems, most of which are useful for solving particular problems or target only some development stages [63].

ProCom is a component model designed for the development of component based embedded systems in the vehicular-, automation- and telecommunication domains. It has been developed within Progress, a strategic research centre whose goal is to provide theories, methods and tools to increase the quality and reduce the costs of embedded system development. Compared to other similar component based approaches, the approach embodied in ProCom is distinguishable for employing components all the way from early design to development and deployment. ProCom also allows for an explicit separation of concerns at different levels of granularity [63].

From a high-level perspective, systems usually consist of relatively independent big units, each of which encapsulates a coherent set of complex functionalities. Looking at a system from a lower-level perspective, it can be seen as comprising small units providing more dedicated and restricted functionality, simpler communication (often inside a single physical node) and stronger requirements for timing and synchronization [4]. When considering complex embedded systems, components representing big parts of the whole system are

essentially different from those responsible for a small part of some control functionality [63]. Big and small parts of a system differ in terms of execution models, communication style, synchronization, information needed for appropriate analysis etc. Hence, ProCom allows embedded systems to be considered from both perspectives, thus providing two distinct levels of granularity in its constituent component models ProSys and ProSave.

ProSys is the layer of ProCom that allows a system to be represented in coarse grained manner. In ProSys, a system is modeled as a collection of ***subsystems*** which communicate with each other concurrently. From the CBSE perspective these subsystems can be considered as components conforming to the component model ProSys. In the context of distributed systems, subsystems are usually meant to be deployed on separate physical nodes. However, such deployment information cannot be specified in a ProSys model but can be modeled in separate deployment model.

A subsystem is specified by typed input and output ***message ports***, which define what type of messages the subsystem can receive and send. Besides message ports, the external view of a subsystem also contains a set of attributes and models related to functionality, reliability, timing and resource usage. They are used for analysis and verification throughout the development process. That is from CBSE perspective the interface of a subsystem consists of message ports, attributes and models.

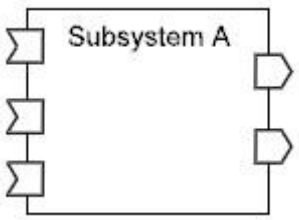

Figure 6 External view of a subsystem with three input message ports and two output message ports [4].

Subsystems are active components, since they can initiate activities either as a response to internal events or on a periodical basis. They can also react to external events in response to the arrival of a message at an input message port. Subsystems within a given system communicate with each other through ***message channels***. A message channel connects one or more output ports to one or more input ports. The types of the input and output ports connected by a message channel must be compatible. Message passing is a non-blocking (asynchronous) action.

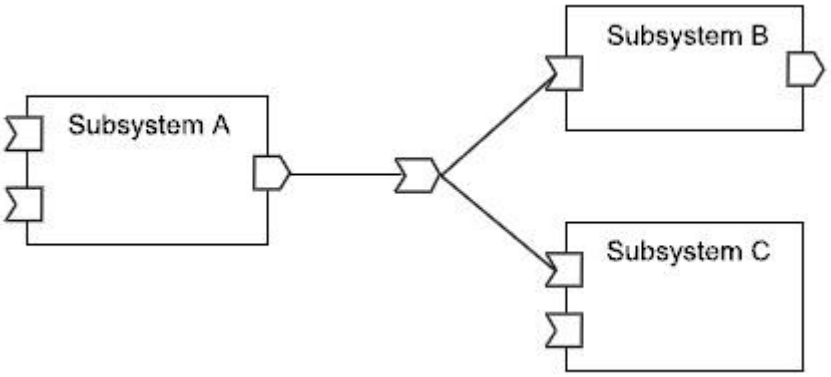

Figure 7 Subsystems and a message channel [4].

ProSys is a hierarchical component model, meaning that subsystems can consist of other subsystems. Subsystems made up of other subsystems are called ***composite***, while those which are not are called ***primitive***. Primitive subsystems are usually internally modeled by ProSave components. Alternatively, they can be implemented by code conforming to the runtime interface of Progress subsystems, which is still to be fully defined. Primitive subsystems are the smallest inseparable units when it comes to deployment, meaning that a single primitive subsystem may not be allocated to multiple physical nodes. As mentioned a ProSys composite subsystem internally consists of other subsystems thus forming a hierarchy. The subsystems comprising a composite one are related with message channels, which provide them with a mechanism for message exchange [4].

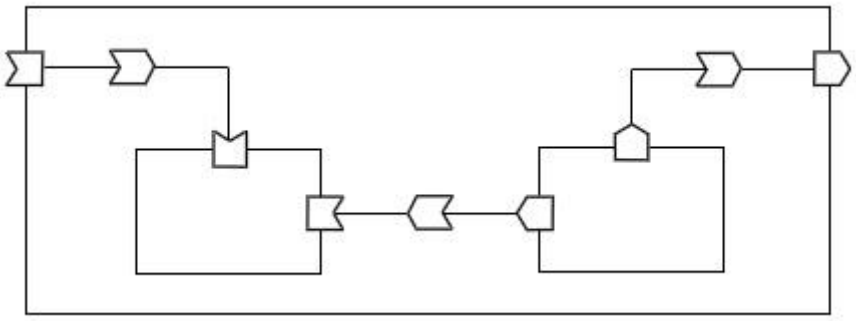

Figure 8 An example of a composite subsystem [4].

ProSave is the layer of ProCom that allows a subsystem to be represented in a fine grained manner. More specifically it is component-based design language especially targeting subsystems with complex control functionality. In ProSave, a subsystem is constructed by binding together components which encapsulate relatively small and rather low-level, non-distributed functionality. Unlike ProSys, ProSave components are passive, meaning that they

cannot initiate activities on their own. Thus in order to get their code executed ProSave components must be activated by an external event. After activation, ProSave components perform their task and return to passive state until they are activated again. ProSave components usually exist only at design time and cannot be distinguished at runtime, unlike the components defined by many existing component models. This is needed in order to achieve better efficiency and to avoid the usage of a component framework at runtime.

The external view (interface) of a ProSave component consists of ports and attributes. Alike ProSys, the attributes provide structured information about the components for the purposes of different analyses. Ports are used to access the functionality provided by a component. In ProSave ports can be classified as either ***input*** or ***output ports***. In essence, input ports are used to receive input data and to trigger the execution of a component, while output ports are used to write the results of this execution. Another definition may be that input ports are the ports that belong to an input port group, while output ports are those that belong to an output port group. Port groups are explained in a successive section. ProSave ports can also be classified as ***data ports*** or ***trigger ports***. Input data ports are used to receive parameters needed by the component when it is activated. Input trigger ports are used to activate a component and to make it execute its code. Output data ports are used as storage for the results of a component execution. Output trigger ports are used to signal that a component has finished its execution.

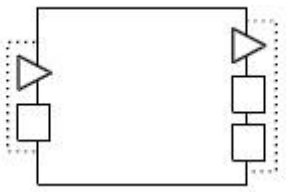

Figure 9 A ProSave component with two input ports (on the left) and three output ports (on the right). Triangles and boxes denote trigger ports and data ports, respectively [4].

The functionality of a ProSave component is exposed to external components in the form of ***services***. Services represent independent functionalities that can be run concurrently. To the external components a service is nothing more than two groups of ports. More specifically a service is defined by:

- An ***input port group*** which consists of a single trigger port and a set of data ports. The data ports are used to store the parameters needed for the component execution. The

trigger port can be used to activate the component and make it produce output based on the parameters in the data ports.

- A set of ***output port groups***. Each output group consists of a single trigger port which indicates when the resulting data is ready and a set of data ports used to store these results.

Each group of ports contains a single trigger port. Each port belongs to a single group and each group belongs to a single service. All data ports have a correspondent type defined as a type definition in C. Attributes are attached to a port, a group of ports or a service. They are used to hold additional information like the worst case execution time of a service or the range of values produced at a data port.

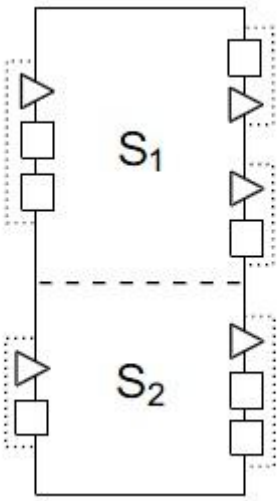

Figure 10 A component with two services - S1 with two output groups and S2 with single output group [4].

The call to a component service follows a sequence of steps. Firstly, the input trigger of the service must be activated. Then all input data ports are read in an atomic operation. After that the service switches from inactive to active state and performs a series of computations. As a result output is produced to the ports of its output group. The writing of results to the output data ports and the triggering of the trigger port of an output group are guaranteed to happen in a single atomic step. A service cannot return to passive state until the results of its computation has been written to all output groups. Also a service cannot be activated in case it is not in passive state.

ProSave defines a model element called ***connection***, which is used to represent communication between components. A connection is a directed edge between an output port of a component and an input port of another one. These ports must be both either data ports or trigger ports. In case they are both data ports their data types must be compatible. There may be at most one connection attached to a data port, except for the case of composite ProSave components which may connect their ports to the ports of an inner component. Connections between data ports denote data transfers. Connections between trigger ports define control flow transitions. The transfer of both data and triggering over a connection is loss-less and atomic. In case that data and triggering transfers appear together, the data must be delivered before the triggering. When a component writes to an output data port this data is not transferred through the associated connections outside the component until the trigger port of its output port group is activated [4].

Besides connections, ProSave defines another type of model elements that represent communication between components – ***connectors***. They do not represent unconditional transfer of data or triggering like connections do. Connectors can be used to manage the data- and control-flow between components. Usually connectors are represented by rounded rectangles with their names inside, but the most widely used connectors may have simplified notations. In some sense a connector is similar to a ProSave component service, since it also has input and output ports and thus can be connected to other connectors or components through connections. A major difference between components and connectors is that connectors are not limited in terms of a single input and a single output trigger port. A connector may have multiple input and output trigger ports or may not have any trigger ports at all. Also unlike component services, connectors may produce output without being explicitly activated on an input trigger port. An example of a connector is the so-called ***data fork***. It has a single input data port and multiple output data ports of the same type as the input one. Whenever a value arrives at the input port it is duplicated to all output ports.

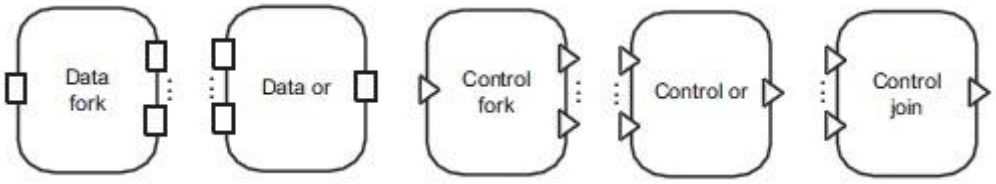

Figure 11 Examples of connectors [4].

Alike ProSys, ProSave components can be classified as either ***primitive*** or ***composite***. Primitive components are implemented as C functions. Optionally a primitive component may define an additional initialization function which is executed upon system startup. In ProSave a composite system consists of components (either primitive or composite), connections and connectors. The ports of the parent component appear "inverted" from the point of view of the inner components and connectors. That is an input port of the enclosing component acts as an output port when seen from the inside. Similarly output ports act as input ports from the perspective of the enclosed components and connectors. This allows the incoming data and triggering to be passed to the inner components and the result of their computations to be passed to the output ports of the parent component.

ProSave and ProSys are used to model the functional architecture of an embedded system. They are designed with the purpose of addressing the concerns of two different levels of granularity. When it comes to embedded systems there are many constraints in terms of efficiency that need to be met in addition to the functional requirements. Thus preserving at runtime a component framework which executes components as they are defined in ProSave and ProCom can in some cases turn to be a significant burden for the overall efficiency. Moreover when developing a distributed embedded system there is a need to specify which parts of the software are executed at the different physical nodes, which cannot be defined in ProSave and ProSys. Thus ProCom allows the modeling of the relations between design-time components and run-time entities.

For this purpose ProCom introduces the notion of ***virtual nodes***. Virtual nodes can be described as an intermediate level in the allocation of functional units to physical nodes. A virtual node represents an encapsulated abstraction of behavior with respect to timing and resource usage. Hence, a virtual node is also a reusable design unit like the other design-time components in ProCom. Each virtual node is associated with one or many ***resource budgets*** defining a minimum level of resource availability provided to a subsystem deployed in it. An example for this is the so-called ***CPU budget***, which can be described as a pair in the form <C, T>, denoting that during each time period of length T the correspondent virtual node is guaranteed access to the CPU for at least C time units [11].

ProSys components (subsystems) are assigned to the virtual nodes of the system being developed. Typically the subsystems are assigned to virtual nodes with resource budgets allowing them to meet their extra-functional requirements. Given that a subsystem is assigned

to a virtual node, it cannot be simultaneously assigned to another one. If a composite subsystem is assigned to a given virtual node all of its subsystems are considered assigned to it as well. Each virtual node is mapped to a single physical node. The distribution of virtual nodes to physical ones is done in such way, so that the aggregation of the resource budgets of the virtual nodes does not exceed the hardware capabilities of the physical nodes [11].

5.2 Method of comparison

The method to be used for the comparison of CBSE and MDD from the perspective of ProCom extends the previously applied general one. In brief the new method implies detailed comparison of the employed in ProCom features of both development paradigms with respect to each of the previously defined comparison aspects. When considering a given aspect the comparison should discuss what is the ProCom approach (if any) to coping with the problems of both paradigms identified in the general comparison and how ProCom makes use of their advantages. The scope of the comparison aspects guarantees that the discussion covers the major factors and stages of the development lifecycle. Like the general comparison of CBSE and MDD, the comparison with respect to ProCom extends the detailed discussion on the comparison aspects with a higher-level comparison, which extracts the essence of the previous one. The new method also implies the consideration of possible improvements of ProCom based on the earlier discussion.

6 Comparison of CBSE and MDD with respect to ProCom

In this section the previously introduced method for the comparison of CBSE and MDD from the viewpoint of ProCom is applied. The comparison is focused on the long term vision for ProCom not only on the current state of affairs. However, the current technological state of ProCom is also reflected in the discussion. Just like the general comparison this one is organized into subsections discussing the comparison aspects, only this time from the perspective of ProCom, and a final subsection that summarizes and analyzes the comparison results.

6.1 Business specifics

6.1.1 Drivers

Aimed at the embedded system development, ProCom embodies one of the few approaches that make use of components throughout the whole development process. In ProCom components are not present only at runtime for performance reasons. However, the synthesis of executable entities from model entities (the ProCom components, connectors, etc.) is automatic and thus transparent to the software developers. Thus the drivers for using ProCom for the embedded domain are more or less the same as those for applying CBSE in general. That is the main driver is to achieve reuse which as a consequence can lead to reduced time-to-market, enhanced quality and simplified maintenance. In addition to bringing component based development to the embedded domain, ProCom also employs many model driven techniques, as it will be explained in the successive sections. These modeling techniques allow for early evaluation and abstraction of irrelevant details at design time. These can also be considered as drivers for choosing ProCom, since embedded system development is proverbial for being overly complex, expensive and error prone.

6.1.2 Maturity

ProCom is a component model created in an academic environment. It has not been tested in industrial environment and there have not been any publications describing practical

experience in either industrial or academic settings. Hence the evaluation of the ProCom maturity from a practical point of view is not possible for the time being.

From a theoretical perspective ProCom can be considered a relatively mature component model. Unlike many other component models ProCom allows for the specification of extra-functional properties. Also ProCom includes a lot of model driven techniques, thus addressing many of the issues of CBSE identified in the analysis of the general comparison. Last but not least the research for related work carried by the ProCom developers has not identified another component model for the embedded domain employing components all the way from early design to development and deployment [63]. Then again ProCom is still under active research and improvements for some of its features (e.g. extension of its attribute framework) are underway and hence it may be too early to declare it as mature.

6.1.3 Target system specifics

ProCom has been designed for the development of distributed embedded systems. However, the idea to design at two distinct levels of granularity is applicable for other types of systems as well. Moreover, many concepts from ProSave and ProSys (e.g. subsystems, message ports, channels, connectors etc.) can be used for generic system modeling as well. Also the notion of virtual nodes assigned to physical ones can be used for deployment modeling of arbitrary distributed systems - even large scale web applications deployed on multiple application and web servers.

However, the current version of ProCom is intended only for the embedded domain and using it for other types of systems would require significant modifications. Moreover, since ProCom supports as a programming language at the lowest level only C, but not higher level languages like Java and C#, it is unlikely that it is used in other domains. Then again research about the applicability of the core ideas and concepts of ProSys and ProSave for modeling other types of systems is still viable.

6.2 Requirements

6.2.1 Requirement specification

ProCom is one of the component models which do not provide a generic way to model system requirements. From some perspective the attribute framework of ProCom can be seen as a way to model requirements for a system or a component. However, in the context of ProCom attributes are meant to specify extra-functional properties/requirements only. Attributes attached to ProCom artifacts are not as expressive in terms of specifying functionality and desired behavior as other modeling formalisms - e.g. UML activity diagrams. Thus system engineers would typically need to use other modeling formalisms for this purpose and then to manually transform these to an architecture in the form of a ProSys model. In this respect it is sensible to define model-to-model transformations from requirements represented in popular formalisms to ProSys models. As an example such transformation may convert an activity diagram annotated with MARTE to a ProSys model with subsystems' attributes, describing their non-functional properties. Facilitating such a transformation within the specification of ProCom would be a model driven feature that increases the overall traceability of the system under design.

Actually, there has been previous research by Petricic et al. [64] [65] about the transformation of UML component diagrams to SaveCMM models (the predecessor of ProCom). This research shows the feasibility of UML to domain specific language (like ProSys and ProSave) transformation. Thus a viable research domain is the definition of transformations from UML diagrams representing requirements to ProCom models.

However, using domain specific UML extension in ProCom may introduce additional complexity to a component model, which has already introduced its own domain specific modeling formalisms. Also, it is clearly the method of choice of the developers of ProCom to employ domain specific formalisms explicitly aimed at embedded systems, rather than to use modifications of existing established formalisms which may be "clumsy" for that particular domain. Thus a better approach may be to incorporate a new modeling formalism for requirement specification, which is again specifically aimed at embedded systems (like the one proposed by Feyerabend [66]). In order to promote traceability it is essential that the ProSys subsystems can be traced back to the correspondent elements of the models represented in such formalism.

6.2.2 Changing requirements

Even though ProCom is a component model, the approach embodied in it can hardly be considered as bottom-up. Unlike many existing component based approaches, the amount of modeling employed by ProCom is significant. Detailed ProSys and ProSave models defining the overall architecture and the low-level design are required before implementing the designed entities in code or reusing existing ones. Hence ProCom has much more features of a top-down approach than of a bottom-up one. As discussed in the general comparison most component-based approaches are bottom-up, which makes it hard to facilitate "top-down" changes resulting from changing requirements. By extending the traditional component based approach with top-down, model-driven techniques, ProCom strives to achieve better traceability between design-time and implementation artifacts resulting in increased resilience to changes. However, ProCom does not provide means to directly trace a changed requirement to the correspondent ProSys subsystems since it does not define modeling formalisms for specifying requirements. Incorporating such a modeling formalism into ProCom can lead to even better traceability and resilience to requirements changes.

ProCom systems can observe the phenomenon called cyclical requirement-component dependency (CRCD) introduced by Tran et al. [34], which is distinctive for the component based approaches. This clearly needs to be addressed by some of the suggested approaches for mitigating the effects of CRCD [34]. Once again, documenting the requirements-to-components transformations turns out to be beneficial, since it is one of the main approaches identified by Tran et al. for mitigating the effects of CRCD.

6.3 Design

6.3.1 Separation of concerns (SoC)

Being a component model, the software elements in ProCom, as defined by SoC, are the different components as they are defined by ProSys and ProSave. ProSys subsystems and composite ProSave components are further subjects to SoC themselves, meaning that they are further divided into subcomponents. From model-driven point of view SoC is achieved again through these components, since ProSave and ProSys components are the only design-time entities in ProCom and are used to express the different views of the system being developed.

Another viewpoint to SoC from the perspective of MDD is the different models/views the components consist of.

6.3.2 Architectural support

As discussed in the overview section 5.1 the purpose of ProSys is to represent a system in a coarse grained manner. ProSys allows the modeling of the overall system structure by using black box components called subsystems, their externally visible properties in the form of attributes and models and message channels connecting the subsystems. Hence even though this is not explicitly stated in the documentation of ProCom, the ProSys layer can be used to express software architectures. Moreover a ProSys model can be considered a typical architectural view falling into the component-and-connector category of views as to the classification introduced by Bass et al. [37]. The views from this category represent a system as a set of components (which are the principal units of computation) and connectors (which facilitate the communication between components).

However, ProSys can be used to describe only a single structural view of architecture. This view can be represented in different levels of detail by uncovering or hiding some of the lower level models that constitute the primary subsystems. Then again the multiple levels of granularity do not provide significantly different viewpoints to the system. Describing a system with different views/models is a paramount idea in the study of software architectures. Multiple views intended for distinct groups of stakeholders allow for easier and earlier communication of crucial architectural decisions and for evaluation of the overall architecture by stakeholder assessments. Fortunately, ProCom defines another mean in addition to ProSys that allows the representation of an architectural view - the deployment model. While ProSys models mostly represent a logical view of architecture, the ProCom deployment models represent the allocation of components to physical nodes. The specification of ProCom defines well what are the mappings, relations and transformations between these views thus adhering to the best practices of both software architecture development and MDD.

6.3.3 Design for Extra-functional requirements

ProCom uses attributes to specify extra-functional properties. The primary purpose of an attribute is to provide complementary information about a component. More specifically attributes are used to describe the behavior and capability of a component regarding its

performance, resource consumption, reliability, maintainability, security etc. The vision of the framework for specifying extra-functional properties in ProCom [67] considers that attributes should not be limited to components and interfaces only and should be assignable to other design entities like ports and connectors as well. This allows considering factors like communication latency when analyzing ProCom models.

In ProCom an attribute consists of a type identifier and a set of values each of which consists of data, metadata and a set of validity conditions, or more formally:

Attribute = <TypeIdentifier; Value+>
Value = <Data; Metadata; ValidityCondition>*

A type identifier is a unique identifier of the extra-functional property, described by the attribute. It is defined by a short descriptive name (e.g. "Static Memory Usage"), a list of elements to which it is applicable to, and the data format of its attribute instances. Attribute data contains the concrete value for the property, which may be of primitive type (e.g. integer, float) or a composite one.

In ProCom each attribute can have multiple values. This is reasonable since an early evaluation of an extra-functional property may be later augmented with more refined ones. Also an evaluation described by an attribute's value may be valid only in some contexts, and thus other values are needed to describe the corresponding extra-functional property in the other plausible contexts of usage. The metadata element of a value allows distinguishing among the different values of a single attribute. Typical data stored in it are a timestamp of the measurement and a description of the used measurement method. The validity condition elements may also be included in the metadata about a given value. They explicitly describe the particular contexts in which the corresponding value can be trusted [67].

This approach to working with extra-functional properties is a general one and can be applied to other component models as well. It is actually based on the previously mentioned approach for specifying extra-functional properties in component models by credentials introduced by Shaw [68]. Clearly, in terms of modeling extra-functional properties, ProCom takes a component based approach and does not use explicitly any model driven techniques.

6.3.4 Design & Architecture evaluation

The Progress approach strongly emphasizes on analysis so as to allow for estimations and guarantees of important properties. Moreover, Progress brings the idea of evaluation and analysis throughout the whole development lifecycle of embedded systems. As it was identified in the general comparison allowing for early evaluation is one of the main advantages of MDD in comparison with CBSE. In MDD this is achieved by analyzing the abstract preliminary representations (models) of the system under development. In the context of Progress the ProCom models are used for the purpose. The official Progress documentation [4] identifies the following types of analyses as planned for Progress:

- Reliability predictions;
- Analysis of functional compliance - e.g., ensuring compatibility of interconnected interfaces;
- Timing analysis - analysis of high-level timing as well as low-level worst-case execution time analysis;
- Resource usage analysis - memory, communication bandwidth, etc.

ProCom can be used for both high-level and low-level analyses since it includes the ProSys and ProSave models representing a system from different perspectives in terms of granularity. However, much of the planned analyses require information about the allocation of components to physical nodes and the underlying platform. Such information is not present in ProSys and ProSave models, though it is crucial to evaluating embedded systems. In order to perform such analyses the component models and the deployment models should be related. This is the case in ProCom whose deployment model is tightly bound to ProSys thus allowing for allocation and environment specific analyses.

Clearly ProCom takes a model based approach to system analysis by providing interrelated models/views suitable in terms of granularity and represented information for each planned type of analysis. In the general comparison it was identified that a major benefit of MDD is the evaluation of some popular models based on existing best practices and theoretical achievements. This is not the case in ProCom, since it introduces more or less new modeling formalisms. Hence analysts of embedded systems developed in ProCom cannot directly make

use of previous know-how with other modeling formalisms. Also the Progress documentation and white papers do not provide clear guidelines and best practices for the evaluation of ProCom models, leaving this crucial activity to the discretion of system architects and analysts. Accumulating and documenting such know-how about how ProCom systems should be analyzed is important for the overall usefulness of ProCom.

6.3.5 Design reuse

Design reuse is a prerequisite for component reuse, as it was identified in the general comparison. Design reuse is however much more inherent to MDD than CBSE, since it focuses on system modeling from different perspectives and levels of granularity. ProCom tries to combine both approaches by providing modeling formalisms suitable for representing architecture and design in terms of components. Thus reusing ProCom models automatically leads to reuse of the correspondent components as well.

ProCom is aimed at the embedded domain and more specifically at complex distributed embedded systems. Such systems may have different network topologies and nodes with different hardware properties. Thus in order to achieve both component and design reuse ProCom separates the component design from the deployment modeling [11]. ProSys and ProSave models are agnostic of the underlying infrastructure. As a result ProCom design models and their respective components can be reused across different hardware and network environments by specifying suitable deployment model for each case.

6.4 Development

6.4.1 Required tools, technologies and their maturity

Model driven approaches require greater tool support than CBSE, as discussed in the general comparison. Specialized tools are needed for almost all types of models. Being primarily a component model ProCom also incorporates many model driven techniques and defines new modeling formalisms. Since these formalisms were introduced as a part of ProCom and are not used elsewhere, no existing tools can be reused. Thus the approach embodied in ProCom is in need for new tools that allow the creation, maintenance and analysis of these new models. This is essential to the usability of ProCom since both ProSys and ProSave models are visual in nature and are very much interrelated. Failing to provide adequate tool support

that allows to intuitively work with such models and to navigate between container and contained components may significantly hinder the usefulness of ProCom.

This has been foreseen by the creators of ProCom and they initiated the creation of the ProCom Integrated Development Environment (PRIDE) [69]. PRIDE is an integrated development environment built as an Eclipse RCP application, which has two major implications. Firstly, the Eclipse RCP platform is widely known for its extensibility and thus new plug-ins can be easily added. This allows for the creation of third party plug-ins for PRIDE providing additional analysis functionality. Secondly, PRIDE follows many of the concepts for editors, explorers and navigators implemented in the Eclipse IDE, which is one of the most popular development environments in the Java world. This can significantly reduce the efforts needed to get acquainted with PRIDE for many developers.

It is beyond the scope of this thesis to provide a detailed description of PRIDE, which is still under active development, and here only its driving requirements will be outlined. The PRIDE documentation [70] identifies the following requirements as the principles guiding its development:

- Allowing to navigate between any development stages;
- Displaying the consequences of a change in the system or within a component;
- Supporting the coupling with the hardware platform;
- Enabling and enforcing the analysis, validation and verification steps.

Since both PRIDE and ProCom are still in a research/experimental phase and are still used mostly in an academic environment, evaluating the possibility for vendor lock-in is not relevant for the time being. In the future if ProCom is used in industrial settings, vendor lock-in can be avoided in case that the ProCom specification has been set as a standard implemented by multiple tools like PRIDE. Another factor for avoiding vendor lock-in is the presence of data formats for the exchange of ProCom models across tools. XMI can be used for this purpose, in case that ProCom models are expressed in MOF. Alternatively a new ProCom specific exchange format can be defined.

6.4.2 Verification and Validation (V&V)

As discussed in the general comparison verifying the properties defined in a component interface is crucial to its successful reuse in a variety of systems. When using ProCom, component and system verification is even more important because of the high quality demands in the embedded domain. The ProCom documentation is not specific about how to perform component and system verification. The vision for appropriate tool support for ProCom [70] mentions that analysis and verification utilities are planned, but no specifics are provided. Bures et al. [63] also mention the possibility to use formal verification of properties of the components and the system as a whole, but again no details are given. The research about verification and validation (V&V) of ProCom components and systems can be considered in an early conceptual phase. Thus this section only speculates about possible approaches borrowed from MDD, CBSE and general purpose software engineering that can be used in ProCom.

The lowest level components in ProCom are the primitive ProSave components, which are implemented as C functions. Unit testing is the predominant approach for testing such modules and can be applied in this case for both white-box and black-box testing. Moreover, since components are meant to be reused in different contexts, they should be comparatively self-sustained an independent. This makes them very suitable for unit testing, since they are not very much coupled with other modules and thus are testable in isolation. Higher order ProSave components can also be subjected to unit testing, since they can be represented as C functions calling the ones of the aggregated components. Unit tests can also be used to verify the extra-functional properties of a ProSave component. Unfortunately many of the extra-functional properties are dependent on some deployment information, which is not present at the time of development. Unit testing ProSys components is also possible albeit not that easy. Executing unit tests against ProSys components would require actual deployment on a test environment.

Providing many formal models, ProCom components can be subjected to many automatic inspections aimed to verify their properties. One approach is to use formal verification [63]. More specifically model checking approaches represent a viable research domain. In brief, a model-checking tool takes as input system requirements or design (called models) and a property of the system/component (called specification). Based on this input the tool then outputs if the model satisfies the specification. In case the model does not satisfy the

specification a counterexample pinpointing the source of the error is provided [71]. There are many tools for model-checking, each of which relies on formalisms for the representation of the models and the specification. By translating ProCom models and attributes to such formalisms existing tools can be used for the verification of both component and system properties.

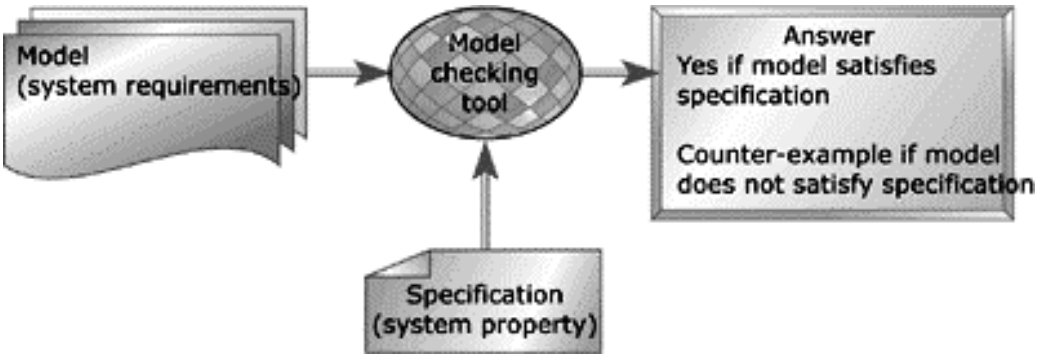

Figure 12 The model checking approach [71].

Another approach for verification via automatic inspections is to create tools that search for bad practices in the ProCom models. This approach is similar to the approach of some static source code tools that automatically check for well-know bad practices in source code written in a popular programming language. The difference is that in the case of ProCom such tools would look at a higher level component models rather than at the source code itself. As mentioned still there has not been enough real-life experience with ProCom to deduce a set of clear design guidelines and best/bad practices. This is a major obstacle for the creation of such tools.

Providing rich component models, ProCom allows for model-based testing, discussed in the general comparison. At first abstract tests can be obtained from the ProSys and ProSave models. For example such tests may be independent of the exact deployment specifics, so they can be reused in different environments. These abstract tests can then be enriched with allocation information and run on a dedicated test environment. Once again the need to have actual deployment on a test environment is a major drawback of this approach.

Last but not least, the amount of modeling employed in ProCom allows for early and easier validation. The ProSys visual models representing a system in high level of granularity allow an analyst to see the system in a summarized way and to reason whether its structure meets the stated requirements.

6.4.3 Traceability, understandability and maintainability

In the general comparison it was identified that CBSE itself does not define adequate practices that lead to improved traceability and understandability. Fortunately, ProCom augments the core ideas of CBSE with model driven approaches and thus the previously described benefits of MDD with respect to traceability, understandability and maintainability can be observed in ProCom as well. Indeed the presence of interrelated models at different levels of granularity (ProSave, ProSys, deployment models etc.) allows a software engineer to easily navigate between implementation and design entities. These models also can serve as a systematic documentation of the relations between architecture, lower level design, implementation artifacts and resource allocation thus allowing a new team member to get easily acquainted with a system under development.

ProCom models are visual in nature which unfortunately leads to problematic model comparison and merging. This may turn to be a major problem if ProCom is used in big projects where multiple models are simultaneously worked on. This is an inherent problem of almost all visual models and its adequate resolution is essential for both ProCom and MDD in general.

6.4.4 Dealing with legacy code

ProCom takes a component based approach to reusing legacy code - that is the legacy code is encapsulated into black-box components. If the legacy code has been originally developed for a ProCom system in the form of ProSys or ProSave components the reuse is straightforward, though the synthesis process of the new system should be done anew. In case the legacy code is not in such a ready to reuse form there are two approaches for reuse. The first is to break down the legacy code into C functions, which can then be reused in the form of primitive ProSave components. As discussed in the general comparison breaking the legacy code into multiple small components leads to better maintainability and makes the code easier to modernize. However, it also requires in-depth knowledge of the legacy code. A typical embedded system may be very complex, concurrent and usually contains a lot of low-level platform specific logic. Breaking such a system into relatively independent primitive ProSave components (C functions) may not always be possible or may require enormous efforts. Last but not least this approach only works if the legacy code is written in C.

The second approach is to encapsulate the legacy code in the form of one or many primitive ProSave subsystems. In order to achieve this, the code should undergo some modifications or additions in order to conform to the so-called runtime interface of Progress subsystems. Unfortunately, this interface has not yet been fully defined [4] and thus it is not possible to assess the needed efforts for such modifications. However, since ProSave subsystems are at much higher level of granularity decomposing existing systems into such subsystems should be much easier than in the first approach. Moreover, primitive ProSys components are active, concurrent and coarse grained by definition and are designed to be deployed separately. Thus, in many cases the architectural decomposition in components of the legacy system can be used as a source for legacy components.

As discussed in the general comparison, to make use of legacy code when applying a model driven approach, this legacy code should be abstracted in models. These models should be at the same level of abstraction and should be described by the same formalisms as the others. By encapsulating the legacy code into black box components, which in ProCom are design time entities only (models), both previous approaches stick to this idea. Once abstracted into ProSys or ProSave components, legacy components can be used at design time just like the other components.

6.4.5 Code reuse

As identified in the general comparison the critical or unique to a given system components are typically not reused from preexisting developments but are designed and developed specifically for that system. Thus developing a set of related systems is a fine opportunity for component reuse, since the custom developed business specific components can be reused across the whole system population. This is often the case in the embedded domain, where a company would typically manufacture series of similar product models (e.g. vehicles, TV sets) which have a lot of common core functionalities. Besides functionality many products within such a series often share common hardware components and topology. This allows for the reuse of deployment models and executable code as well.

Like dealing with legacy code ProCom makes use of the best practices from both CBSE and MDD when it comes to code reuse. Following a component based approach the core business logic for a suite of embedded products can be encapsulated in ProSys and ProSave components. Components in ProCom are primarily design time entities and are not physically

represented at runtime. Thus component reuse and model reuse in ProCom is more or less the same thing. Hence, as it is in MDD, in ProCom design reuse is a prerequisite for component/code reuse. Besides, component reuse ProCom facilitates the reuse of executable code as well, since it allows for deployment modeling. As mentioned, often within a given suite of embedded products hardware and network topology are similar. This allows for the reuse of deployment models describing the allocation of ProCom components to physical nodes, which results in direct reuse of executables for these nodes.

6.5 Organizational specifics

6.5.1 Development methodologies

ProCom is still in research phase, and so far mostly its technical aspects have been worked on. There has not been any significant industrial or academic research about what are its implications on the used development methodologies. However, being a component model ProCom differentiates between component development and system development, which as discussed in the general comparison is among the main inhibitors for using standard developing methodologies in CBSE.

It is logical to assume that ProCom implies new roles in the development process. The design and development of the coarse grained ProSys components may be done by system architects, who can then delegate the design and implementation of the lower level ProSave components to fellow engineers. Moreover, the deployment modeling may require specialists with deep knowledge about embedded hardware and network topology to work with the architects and engineers for the appropriate allocation of components to physical nodes with respect to the extra-functional component properties and the requirements for the whole system. These roles are not accounted by traditional development methodologies.

In terms of development methodologies ProCom observes many of the problems inherent for CBSE and alike other component technologies, existing methodologies should be modified when using ProCom. For this purpose new research should be carried out to identify the new roles, development phases and activities implied by ProCom. Such research is needed as a basis for further study (either academic or industrial) about how existing methodologies can be modified for this purpose.

6.5.2 Organizational requirements

Revisiting the survey carried by Bass et al. [58] and discussed in the general comparison, the main inhibitors for adopting a component based approach are:

- Lack of available components;
- Lack of stable standards for component technology;
- Lack of certified components;
- Lack of an engineering method to consistently produce quality systems from components.

Most of these are also valid threats when it comes to ProCom. The lack of available and certified components is an even greater threat to ProCom than to other component technologies, because of the specifics of the embedded domain. Typically an organization developing embedded software should take into consideration the hardware and network topology this software is going to be deployed on. Thus the requirements for extra-functional properties with respect to the underlying platform are high and it is unlikely that components initially developed and tested for other platforms can meet these requirements. An organization using ProCom for a few unrelated projects may find itself developing a huge set of custom components that are never reused afterwards. Hence using ProCom may not be beneficial in case that the adopting organization does not develop a suite of related products with common core functionality and similar hardware and topology.

The general lack of stable standards for component technologies is not something that should bother an organization adopting ProCom. On the contrary, if ProCom itself is set as an independent standard which is implemented by multiple vendors then an adopting organization can eliminate the risk of vendor lock-in. However, the lack of proven engineering methods is a threat when adopting ProCom as discussed in the preceding section. Thus a lot of the existing in-house processes should be reformed so as to accommodate the new roles and approach implied by ProCom.

The lack of modeling experts within an organization, which is the major inhibitor for adopting MDD, is an even graver inhibiting factor when it comes to ProCom. As discussed it is the method of choice of the developers of ProCom to use new domain specific modeling

formalisms rather than existing and well known modeling standards. A major drawback of this approach is that an adopting organization cannot make use of already accumulated in-house modeling know-how and should invest in the arrangement of regular trainings and mentoring.

6.5.3 Team member qualification

When taking up ProCom engineers should be familiar with the basics of both CBSE and MDD, since ProCom makes use of techniques from both approaches. Another factor contributing to the steepness of the ProCom learning curve are the newly created modeling formalisms. ProCom introduces new formalisms instead of reusing standard ones, which results in a steeper individual learning curve and a greater need for trainings. The developers of ProCom envision that such an investment in training should pay-off in the long run because of the better suitability of the new formalisms for the embedded domain. Then again this has not been yet validated and a substantial upfront investment in training is needed before/if eventually it is returned.

Also the previously described unavailability of established development methodologies and best practices for ProCom, drives for substantial efforts from both technical and managerial staff in order to modify accordingly the existing in-house practices. As discussed in the general comparison, a suitable approach for defining a component aware approach is to devote a non-critical project for experimenting purposes, since no related research has been carried before. Another way may be to resort to external coaching, though it is unlikely that suitable expertise will be present for ProCom which is still being actively developed and as discussed faces series of open problems.

6.5.4 Financial issues

ProCom is still in research phase and some of its constituents (e.g. the interface of Progress subsystems) have not yet been fully defined [4]. Because of this it has not yet been adopted by an organization and no publications describing what the impact of such an adoption may be exist. Since obviously ProCom and its tooling are about to undergo continuing development in the near future, this section only outlines factors that from what is the current state of affairs seem to have financial effect on an adopting organization.

One such factor is the learning curve, since ProCom introduces brand new modeling formalisms (ProSys and ProSave) and a new way of deployment modeling. Though the formalisms are relatively simple and their concepts and syntax can be learned rather quickly, mastering the different styles of modeling implied by ProSys and ProSave would require significant training. Moreover, since most of the models in ProCom are visual, specialized tooling is needed to create and work with the models, which also adds to the learning curve. The idea behind introducing so much new modeling formalisms and tools aimed explicitly at the embedded domain is to enable easier design and development of embedded systems by avoiding the "clumsiness" of the existing standards and tools when it comes to embedded system development. Thus, in the long run adopting ProCom holds the potential to pay off for the upfront investment in training by making embedded system development easier and faster. However, this has not yet been validated and research in different industrial settings measuring aspects of the return of investment (ROI) in learning and training when adopting ProCom is needed.

Another factor influencing financially an adopting organization is the need for investments in modifications of the existing in-house engineering process so as to facilitate the specifics of ProCom. The fact that no related research has been carried so far about ProCom implies that an adopting organization would need to carry its own research to identify ProCom's specifics in terms of activities and roles. Such an in-house research may take significant amount of time and requires the involvement of key managerial and technical staff.

With respect to tools, the ProCom development environment PRIDE can be downloaded for free and thus does not present an additional expenditure. PRIDE provides the essentials for working with ProCom models, but alike ProCom it is still in research and development phase. When it comes to tooling, a possible expenditure may be for third party tools or add-ons to PRIDE that provide additional specific analyses over the ProCom models. An example for this may be a tool that provides automatic analysis for design anti-patterns in the ProCom models. Such tools are very likely to be developed as ProCom gains popularity, in order to further ease the development of well designed, high-quality embedded systems.

6.6 Comparison results and analysis

The following table summarizes the previous discussion. For each of the comparison aspects summaries of the ProCom properties inherent to CBSE and MDD are provided:

	Summary
Business specifics	
Drivers	Drivers for using ProCom are the same as those for CBSE & MDD, since it makes heavy use of approaches from both of them. ➢ **CBSE perspective**: Component reuse ➢ **MDD perspective**: Work at a higher level of abstraction than code. Complexity management. Early evaluation.
Maturity	Relatively mature, though still under development. Still not tested in industrial environment. ➢ **CBSE perspective**: Relatively mature from theoretical perspective - incorporates components throughout the whole lifecycle, allows specification of extra-functional properties ➢ **MDD perspective**: Incorporates the basic ideas of MDD - defines modeling formalisms for representing a system from different viewpoints, a system is developed as a series of models related with transformations.
Target system specifics	Embedded systems.
Requirements	
Requirement specification	ProCom does not provide a general purpose way for specifying requirements. ➢ **CBSE perspective**: The attribute framework for specifying extra-functional component properties can be used to model extra-functional requirements. ➢ **MDD perspective**: N/A
Changing requirements	In terms of changing requirements ProCom is better than many existing component technologies, because of the amount of modeling extending the core CBSE ideas. ➢ **CBSE perspective**: Model driven techniques mitigate many of the inherent for CBSE problems. Cyclical requirement-component dependency (CRCD) is still a problem. ➢ **MDD perspective**: ProCom systems are more traceable due to the employed modeling techniques. This allows for an adequate reaction to a change in the requirements.
Design	
Separation of concerns (SoC)	Being a component model the SoC elements in ProCom are the different types of components. ➢ **CBSE perspective**: SoC is implemented through components and interfaces. ➢ **MDD perspective**: Components in ProCom are design-time entities and thus they are the SoC elements in terms of MDD as well.

Architectural support	ProSys models and the deployment models can be considered as two architectural views. ➢ **CBSE perspective**: Alike other component based technologies ProCom introduces its own ways for architecture specification formalisms - ProSys and the deployment modeling formalism. ➢ **MDD perspective**: ProSys can be seen as a modeling formalism for specifying architecture. ProSys and the deployment models can be seen as two related architectural models of a system.
Design for Extra-functional requirements	➢ **CBSE perspective**: ProCom uses a custom attribute framework for specifying extra-functional requirements. It is based on the approach for describing extra-functional properties in component models by credentials. ➢ **MDD perspective**: N/A
Design & Architecture evaluation	➢ **CBSE perspective**: N/A ➢ **MDD perspective**: ProCom takes a model driven approach to design and architecture evaluation by providing interrelated models/views suitable for different analyses.
Design reuse	➢ **CBSE perspective**: N/A ➢ **MDD perspective**: Functional design in ProCom can be reused, since it is expressed through ProSys & ProSave components which are design time entities (models). Another aspect of design reuse in ProCom is the deployment modeling which allows information about allocation of functionality to hardware to be reused across similar technological environments.
Development	
Required tools, technologies and their maturity	Currently the only tool supporting ProCom is PRIDE, which is still being actively developed and yet cannot be considered mature. ➢ **CBSE perspective**: N/A ➢ **MDD perspective**: Due to the support for visual modeling formalisms ProCom implies the use of sophisticated IDEs like ProCom
Verification and Validation (V&V)	The research about V&V of ProCom components and systems is in an early conceptual phase. One possible approach is unit testing of the components, which is especially useful for the lowest level ProSave components. Model driven approaches performing analyses on the variety of ProCom models and model based testing are also possible. ➢ **CBSE perspective**: N/A ➢ **MDD perspective**: model based testing, model analyses.
Traceability, understandability and maintainability	➢ **CBSE perspective**:N/A ➢ **MDD perspective**: High traceability and maintainability because of the many interrelated models. Problematic versioning and merging of the visual models.
Dealing with legacy code	➢ **CBSE perspective**: Legacy code is divided into components of different granularity and then reused. ➢ **MDD perspective**: Components in ProCom are design-time entities and thus when the legacy code is divided into components these can be used for modeling as well.

Code reuse	➢ **CBSE perspective**: Components are the main artifacts of reuse. Because of the specifics of the embedded domain, significant reuse is most likely to occur within a suite of similar projects. ➢ **MDD perspective**: In ProCom model reuse and component reuse is more or less the same thing. Reusing deployment models leads to reuse of executables.
Organizational specifics	
Development methodologies	➢ **CBSE perspective**: ProCom implies customization of existing development methodologies alike other component technologies. ProCom may imply new roles in the development process, which should be incorporated into it accordingly. ➢ **MDD perspective**: Unlike most MDD technologies, ProCom implies significant changes in the used development methodologies.
Organizational requirements	➢ **CBSE perspective**: Alike CBSE, when adopting ProCom an organization should evaluate its eventual ROI, since custom component development for specific hardware is imminent. Also an adopting organization should have substantial expertise about methodologies so as to incorporate the ProCom specific activities and roles. ➢ **MDD perspective**: ProCom introduces new modeling formalisms and thus employees' qualification is of even greater concern than in other MDD technologies. There is great need for employee trainings. It is unlikely that an organization taking up ProCom for just a few projects can have substantial ROI because of the investments in training.
Team member qualification	➢ **CBSE perspective**: Substantial efforts from staff to modify accordingly the existing in-house practices. A pilot low-priority project may be used for experimenting and training purposes. ➢ **MDD perspective**: Steep individual learning curve due to brand new formalisms and tools.
Financial issues	➢ **CBSE perspective**: Need for investments in modifications of the in-house practices. Freely accessible sophisticated tooling, though still in development phase - PRIDE. ➢ **MDD perspective**: Steep individual learning curve because of the new formalisms and great need to training. Freely accessible modeling tool - PRIDE.

Table 2 Summary of the comparison with respect to ProCom

ProCom is a relatively mature component model from theoretical point of view. In the general comparison, when discussing several comparison aspects it was identified that component based approaches could benefit from modeling extensions allowing engineers to model a system design from different perspectives and levels of granularity. Clearly ProCom takes a step in this direction by introducing several new modeling formalisms for this purpose. Actually ProCom models are only design time entities (models) and detailed models of the system under development are needed before it is built and deployed. All these models are interrelated just as it is in MDD. Also the generation of executable entities (a process called synthesis) can be thought of as model-to-code transformation. In this respect ProCom is not

only a component technology, but also can be considered a model based one since it employs the core approaches of MDD.

Besides the amount of modeling, another feature of ProCom speaking for its theoretical maturity is the support for extra-functional modeling, which is absent from most existing component based technologies. However, ProCom is still in research phase and faces many open problems (e.g platform modeling and inclusion of hardware into the models) and thus declaring it as mature may be too early. Not to mention that there is a gap between academia and industry and often things considered mature and well developed from academic point of view are not well accepted in practice. Currently no research describing experience with ProCom in either academic or industrial environment exists. Thus further research is needed about the applicability of ProCom in a variety of projects and organizations.

Alike many existing component technologies, ProCom does not provide any general mean to model requirements. As discussed in the general comparison this may result in difficulties to quickly evaluate the impact of a changed requirement. If requirements models were present and interrelated with the architectural description in the form of ProSys models this would result in a greater traceability and easier impact analysis of changing requirements.

It is reasonable to extend ProCom with further modeling capabilities for requirement specification. One approach to this is to take up an existing standard for modeling requirements and to define suitable model-to-model transformations that interrelate the requirements models to the ProCom models. Similar research has been done before, showing the plausibility of such an approach. Another approach is to define new ProCom specific formalisms for representing requirements. This approach is closer to the method of choice of ProCom to define new embedded specific formalisms, rather than to reuse established standards. Whatever the approach it is important that ProCom model entities can be traced to the requirements they implement and vice versa. This is important not only for impact analysis but also for the mitigation of the risk called cyclical requirement-component dependency (CRCD) as identified by Tran et al. [34].

Defining architecture and designing at different levels of granularity are the two activities that benefit from the incorporation of model driven approaches in ProCom. As discussed in the general comparison CBSE theory is not concrete as to how and what should be modeled and many component technologies do not allow modeling at all. Most component technologies

that allow modeling introduce their own modeling formalisms. In the general comparison it was identified that having a single modeling formalisms at disposal is limiting in terms of expressiveness. Having multiple different formalisms at disposal and allowing for transformations between models described in these formalisms (as in MDD) would be very beneficial for a component model. Exactly this is the approach taken in ProCom which provides several modeling formalisms for representing different views. ProSys and the deployment models can be used to represent different architectural views of a system. ProSave models are used for low-level modeling and the attribute framework allows for modeling extra-functional properties for an arbitrarily design element. These are all interrelated thus adhering to the best practices of MDD. Incorporating model driven approaches in ProCom results in many of the benefits, considered as drivers for MDD - early design evaluation, analyses over a variety of models etc.

The downside of incorporating so much modeling in a component model like ProCom is the greater need for tool support. Moreover since the ProCom models are visual they are hard to merge and compare. This is a major problem for big projects where multiple models are simultaneously worked on. Currently the PRIDE development environment provides the essential tooling for using ProCom. PRIDE is still in development phase and it is too early to think about possible vendor lock-in. Then again it can be speculated that if ProCom defines a format that allows its models to be exported and imported, this can significantly reduce the probability of vendor lock-in if multiple tools for ProCom are developed. One approach for this is to define transformations between the ProCom models and XMI. Alternatively a custom exchange format can be defined.

Verification and Validation (V&V) are also among the activities that benefit from the amount of modeling in ProCom. The variety of models allows for model based testing and different analyses over the models. Another approach to verifying the lowest level components, though not specific to MDD or CBSE, is unit testing. Other benefits of incorporating model driven approaches are the improved traceability, understandability and maintainability. In the general comparison it was identified that a major benefit of a model driven approach is the traceability between the initial requirements and the design and implementation artifacts. In the case of ProCom requirements are not explicitly represented and this is a problem for the overall traceability and understandability of a ProCom system.

ProCom adheres to the CBSE approach when it comes to code reuse and dealing with legacy code. In ProCom the legacy code is divided into components which are then used in the design process. Interestingly enough, this approach also coincides with the model driven approach to handling legacy code. In MDD models are extracted from the legacy code, which are then used in the development process. Since in ProCom components are only design time entities, the disassembling of the legacy code into ProCom compliant components effectively means extraction of ProCom models of the legacy code. These models are then used at design time just like the others.

Evaluating the organizational specifics implied by the adoption of ProCom is a rather difficult task, since ProCom has not yet been incorporated into real-life projects. Moreover, ProCom still faces many open problems and is still under active research and development. Thus the discussion of these comparison aspects only speculates about factors that may possibly influence an adopting organization. Being a component model ProCom differentiates between component development and system development. Hence, alike other component models a major issue for ProCom is the unsuitability of the existing development methodologies. What makes things worse for ProCom, is that it may imply new roles and phases specific to the embedded domain which are also not handled by most methodologies. This may affect significantly an adopting organization, since it would have to invest into research and modification of its existing practices. Another inhibiting factor is that ProCom introduces new modeling formalisms which results in a steep individual learning curve and a need for significant investment in training. Last but not least, it should be taken into consideration that if an adopting organization does not use ProCom to develop suites of similar products with similar hardware and network, it may not have significant ROI. Such an organization may find itself developing components that are rarely or never reused afterwards.

7 Conclusion

In this thesis two systematic comparisons of CBSE and MDD were carried out. The first one compared CBSE and MDD in general, based on both theoretical and practical research about their foundations and properties. For this purpose two sections were dedicated for succinct overview of the main notions, theoretical backgrounds and current trends of both approaches. Following a new comparison method was introduced, which is based on a two-level hierarchy of comparison aspects. Each of these aspects is correspondent to an important development activity or a factor influencing the decision whether to adopt a development approach or not. The general comparison consists of detailed analysis of the features of CBSE and MDD with respect to each comparison aspect and a section summarizing and analyzing these detailed discussions.

The general comparison showed that CBSE is superior to MDD when it comes to code reuse and dealing with legacy code. Component based approaches typically require fewer and simpler tooling than model driven ones. Also the learning curve for a component technology is usually less steep than the one for a model driven approach. The detailed analysis of the comparison aspects showed that CBSE theory is not concrete as to what modeling (if any) should be used when developing a component or a system. Many existing component based technologies either do not incorporate modeling at all or only allow for restricted modeling. Consequently, the general comparison identified that typically using MDD results in more traceable, maintainable and resilient to changes systems than using CBSE. Also current MDD technologies are superior to current component based ones when it comes to defining software architecture or design. It was identified that by extending the core CBSE concepts with adequate modeling capabilities these shortcomings of the current component based approaches can be mitigated. Another shortcoming of CBSE is the need to modify existing development methodologies so as to incorporate component specific activities. Unlike CBSE, MDD approaches can be used with almost all existing development methodologies, with minor adaptations.

The second comparison presented in the thesis analyzes CBSE and MDD in the context of ProCom. For this purpose a section is dedicated for a brief overview of ProCom. The applied comparison method extends the previous one and is based on the same hierarchy of comparison aspects. The comparison itself includes discussions about what are the features of

ProCom (if any) with respect to each aspect and how these features are related to those of CBSE and MDD identified in the general comparison. Also the comparison includes a section summarizing and analyzing the previous discussions.

The discussions of the comparison aspects identified that ProCom augments the core concepts of CBSE with several model driven approaches, as proposed in the general comparison. Consequently, the shortcomings of CBSE concerning traceability, maintainability, analyzability and specifying architecture and design are mitigated in ProCom. It was identified that these could be further improved if requirements modeling capabilities were incorporated into ProCom. Such capabilities could also make ProCom systems more agile to requirement changes. The incorporation of modeling into ProCom also has some negative impacts – need for sophisticated tooling, steep learning curve due to the new modeling formalisms. A major problem for ProCom is the need for specific development methodologies, which is an inherent problem to CBSE.

8 References

1. Jones, C.: *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies 1st edn.* McGraw-Hill Osborne Media (2009)

2. McConnell, S.: *Code Complete: A Practical Handbook of Software Construction 2nd edn.* Microsoft Press, Redmond, Washington, USA (2004)

3. Mälardalen University: *The PROGRESS Centre for Predictable Embedded Software Systems.* (Accessed March 10, 2011) Available at: http://www.mrtc.mdh.se/progress/

4. Bures, T., Carlson, J., Crnkovic, I., Sentilles, S., Vulgarakis, A.: *ProCom - the Progress Component Mode, version 1.1.* Reference Manual, Mälardalen University, Västerås, Sweden (2010)

5. Törngren, M., Chen, D., Crnkovic, I.: *Component-based vs. model-based development: a comparison in the context of vehicular embedded systems.* In : 31st EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Porto, Portugal, pp.432 - 440 (2005)

6. Bunse, C., Gross, H.-G., Peper, C.: *Embedded System Construction – Evaluation of Model-Driven and Component-Based Development Approaches.* In : 11th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences), Toulouse, France, pp.41-50 (2008)

7. Bunse, C., Gross, H.-g., Peper, C.: *Applying a Model-based Approach for Embedded System Development.* In : Conference on Software Engineering and Advanced Applications, pp.121 - 128 (Augst 2007)

8. Schmidt, D., Balasubramanian, K., Krishna, A., Turkay, E., Gokhale, A.: *Model Driven Engineering for Distributed Real-time and Embedded Systems.* In Gerard, S., Babau, J.-P., Champeau, J., eds. : Model Driven Engineering for Distributed Real-Time Embedded Systems. Wiley-ISTE (2005)

9. Hatcliff, J.: *DARPA PCES Final Report.*, Department of Computing and Information Sciences, Kansas State University (2005)

10. Childs, A., Greenwald, J., Jung, G., Hoosier, M., Hatcliff, J.: *CALM and Cadena: metamodeling for component-based product-line development.* Computer 39(2), 42 - 50 (Februray 2006)

11. Carlson, J., Feljan, J., Mäki-Turja, J., Sjödin, M.: *Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems.* In : 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Lille, France, pp.74 - 82 (2010)

12. Przybylek, A.: *Post object-oriented paradigms in software development: a comparative analysis.* In : 1st Workshop on Advances in Programming Languages In International Multiconference on Computer Science and Information Technology, Wisła, Poland, pp.753-762 (2007)

13. Herzum, P., Sims, O.: *Business Component Factory : A Comprehensive Overview of Component-Based Development for the Enterprise 1st edn.* Wiley (2000)

14. Szyperski, C.: *Component Software Beyond Object-Oriented Programming 2nd edn.* Addison-Wesley (2002)

15. Crnkovic, I., Larsson, M.: *Building Reliable Component-Based Software Systems.* Artech House Publishers (2002)

16. Eiffel Software: *The Power of Design by Contract.* (Accessed March 10, 2011) Available at: http://www.eiffel.com/developers/design_by_contract.html

17. Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: *Volume II: Technical Concepts of Component-Based Software Engineering.* Technical report, Carnegie Mellon Software Engineering Institute (SEI) (2000)

18. Selic, B.: *The Pragmatics of Model-Driven Development.* IEEE Software 20(5), 19-25 (September 2003)

19. Beydeda, S., Book, M., Gruhn, V.: *Model-Driven Software Development 1st edn.* Springer (2005)

20. Rahmani, C., Zand, M., Siy, H., Srinivasan, S.: *A Survey on Model Driven Software Development.* In : International Conference on Software Engineering and Data Engineering, Las Vegas, Nevada, USA, pp.105-110 (2009)

21. Rothenberg, J.: *The Nature of Modeling.* In Widman, L., Loparo, K., Nielsen, N., eds. : Artificial Intelligence, Simulation, and Modeling. John Wiley and Sons, Inc., New York (1989) 75-92

22. *UML Use Case Diagrams: Tips and FAQ.* (Accessed March 10, 2011) Available at: http://www.andrew.cmu.edu/course/90-754/umlucdfaq.html

23. OMG: *OMG's MetaObject Facility.* (Accessed March 10, 2011) Available at: http://www.omg.org/mof/

24. Ambler, S.: *Agile Modeling (AM) Home Page - Effective Practices for Modeling and Documentation.* (Accessed 2011 10, March) Available at: http://www.agilemodeling.com/

25. McIlroy, M.: *Mass Produced Software Components.* In : NATO Software engineering conference, Garmish, Germany, pp.138-151 (1968)

26. OSGi Alliance: *OSGI.* (Accessed March 10, 2011) Available at: http://www.osgi.org

27. Sun Microsystems (now acquired by Oracle): *Java Beans: Introducing Java Beans.* (Accessed March 10, 2011) Available at: http://java.sun.com/developer/onlineTraining/Beans/Beans1/

28. Microsoft MSDN: *Understanding.NET Components.* (Accessed March 10, 2011) Available at: http://msdn.microsoft.com/en-us/library/ms973807.aspx#componentsnet_topic1

29. Liddle, S.: *Model-Driven Software Development.*, Brigham Young University (2010)

30. IBM: *Eclipse Platform Technical Overview.* (Accessed March 10, 2011) Available at:

http://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf

31. OSGi Alliance: *VEG - Vehicle Expert Group.* (Accessed March 10, 2011) Available at: http://www.osgi.org/VEG/

32. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: *The Koala component model for consumer electronics software.* Computer 33(3), 78 - 85 (March 2000)

33. OMG: *Object Management Group/Business Process Management Initiative*. (Accessed March 10, 2011) Available at: http://www.bpmn.org/

34. Tran, V., Hummel, B., Liu, D.-B., Le, T., Doan, J.: *Understanding and managing the relationship between requirement changes and product constraints in component-based software projects.* In : The Thirty-first Hawaii International Conference On System Sciences, Kohala Coast, HI , USA, pp.132 - 142 (1998)

35. Dijkstra, E.: *On the role of scientific thought.* In : Edsger W. Dijkstra, Selected Writings on Computing: A Personal Perspective. Springer-Verlag, New York (1982) 60–66

36. Greer, D.: *The Art of Separation of Concerns.* (Accessed March 10, 2011) Available at: http://www.aspiringcraftsman.com/default/2008/01/03/art-of-separation-of-concerns/

37. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice 2nd edn.* Addison-Wesley Professional (2003)

38. Feljan, J., Lednicki, L., Maras, J., Petricic, A., Crnkovic, I.: *DICES technical report Classification and survey of component models.* Technical report (2009)

39. OMG: *Unified Modeling Language.* (Accessed March 10, 2011) Available at: http://www.uml.org/

40. Open Group: *ArchiMate.* (Accessed March 10, 2011) Available at: http://www.archimate.org/

41. SAE AADL team: *Architecture Analysis and Design Language.* (Accessed March 10, 2011) Available at: http://www.aadl.info/aadl/currentsite/

42. School of Computer Science, Carnegie Mellon: *The Wright Architecture Description Language.* (Accessed March 10, 2011) Available at: http://www.cs.cmu.edu/afs/cs/project/able/www/wright/

43. School of Computer Science, Carnegie Mellon: *The Acme Project.* (Accessed March 10, 2011) Available at: http://www.cs.cmu.edu/~acme/

44. The ATESST Consortium: *EAST ADL 2.0 Specification.* (Accessed February 28, 2008) Available at: http://www.atesst.org/home/liblocal/docs/EAST-ADL-2.0-Specification_2008-02-29.pdf

45. Ameller, D., Franch, X., Cabot, J.: *Dealing with Non-Functional Requirements in Model-Driven Development.* In : 18th IEEE International Requirements Engineering Conference (RE), Sydney, NSW , pp.189 - 198 (2010)

46. OMG: *MARTE.* (Accessed March 10, 2011) Available at: http://www.omgmarte.org/

47. Kent, W.: *A Simple Guide to Five Normal Forms in Relational Database Theory.* In: A Simple Guide to Five Normal Forms in Relational Database Theory. (Accessed March 10, 2011) Available at: http://www.bkent.net/Doc/simple5.htm

48. Chidamber, S., Kemerer, C.: *A metrics suite for object oriented design.* IEEE Transactions on Software Engineering 20(6), 476 - 493 (June 1994)

49. Martin, R.: *OO Design Quality Metrics.* (Accessed October 1994) Available at: http://www.objectmentor.com/resources/articles/oodmetrc.pdf

50. Kang, K.: *Issues in Component-Based Software Engineering.* In : International Workshop on Componentbased software engineering, Los Angeles, CA, pp.207-212 (1999)

51. OMG: *XML Metadata Interchange (XMI).* (Accessed March 11, 2011) Available at: http://www.omg.org/spec/XMI/

52. Wallin, C.: *Verification and Validation of Software Components and Component Based Software Systems.* Extended Report for I. Crnkovic and M. Larsson (editors), "Building Reliable Component-Based Systems" Artech House, July 2002, Chapter 5, Industrial

Information Technology Software Engineering Processes ABB Corporate Research (2002)

53. Reid, S.: *Model-Based Testing.*, Cranfield University Royal Military College of Science, Swindon, UK

54. Utting, M.: *Position Paper: Model-Based Testing.* In : Verified Software: Theories, Tools, Experiments (VSTTE), Zürich, Switzerland (2005)

55. Spivey, J.: *The Z Notation: a reference manual 2nd edn.* Prentice Hall International (UK) Ltd, Oriel College, Oxford, OX1 4EW, England (1992)

56. Parviainen, P., Takalo, J., Teppola, S., Tihinen, M.: *Model-Driven Development - Processes and practices.*, VTT Technical Research Centre of Finland (2009)

57. Crnkovic, I.: *Component-based software engineering - new challenges in software development.* In : 25th International Conference on Information Technology Interfaces , Cavtat, Greece, pp.9 - 18 (2003)

58. Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: *Volume I: Market Assessment of Component-Based Software Engineering.* Technical report, Carnegie Mellon Software Engineering Institute (SEI) (2000)

59. Apache Software Foundation: *Apache Felix.* (Accessed March 10, 2011) Available at: http://felix.apache.org/site/index.html

60. Eclipse Foundation: *Equinox.* (Accessed March 10, 2011) Available at: http://www.eclipse.org/equinox/

61. Borland, part of Micro Focus: *Borland Together.* (Accessed March 10, 2011) Available at: http://www.borland.com/us/products/together/index.html

62. IBM: *Rational Rhapsody.* (Accessed March 10, 2011) Available at: http://www-01.ibm.com/software/awdtools/rhapsody/

63. Bures, T., Carlson, J., Sentilles, S., Vulgarakis, A.: *A Component Model Family for Vehicular Embedded Systems.* In : International Conference on Software Engineering

Advances, ICSEA, Sliema, Malta, pp.437 - 444 (2008)

64. Petricic, A., Crnkovic, I., Zagar, M.: *Models Transformation between UML and a Domain Specific Language.* In : Eight Conference on Software Engineering Research and Practice in Sweden (SERPS 08), Karlskrona, Sweden, pp.1-10 (2008)

65. Petricic, A., Lednicki, L., Crnkovic, I.: *Using UML for Domain-Specific Component Models.* In : Fourteenth International Workshop on Component-Oriented Programming, East Stroudsburg, PA, USA, pp.23-31 (2009)

66. Feyerabend, K., Josko, B.: *A visual formalism for real time requirement specifications.* In Bertran, M., Rus, T., eds. : Transformation-Based Reactive Systems Development. Springer Berlin/Heidelberg (1997) 156-168

67. Sentilles, S., Stepan, P., Carlson, J., Crnkovic, I.: *Integration of Extra-Functional Properties in Component Models.* In : 12th International Symposium on Component-Based Software Engineering , East Stroudsburg, PA, USA, pp.173 - 190 (2009)

68. Shaw, M.: *Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does.* In : International Workshop on Software Specifcation and Design, Schloss Velen , Germany, pp.181 - 185 (1996)

69. Mälardalen University: *PRIDE.* (Accessed March 10, 2011) Available at: http://www.idt.mdh.se/pride/

70. Crnkovic, I., Sentilles, S., Leveque, T., Zagar, M., Petricic, A., Feljan, J., Lednicki, L., Maras, J.: *PRIDE.* In : DICES workshop @ SoftCOM, Split-Bol, Croatia (2010)

71. Palshikar, K.: *An introduction to model checking.* (Accessed March 10, 2011) Available at: http://www.eetimes.com/design/embedded/4024929/An-introduction-to-model-checking