Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes

Ivica Crnkovic[1], Magnus Larsson[2], and Otto Preiss[3]

[1] Mälardalen University, Department of Computer Science and Engineering
Box 883, 721 23 Västerås, Sweden
ivica.crnkovic@mdh.se
http://www.idt.mdh.se/~icc
[2] ABB Corporate Research, 721 59 Västerås, Sweden
magnus.larsson@se.abb.com
[3] ABB Corporate Research, CH-5405 Baden-Daettwil, Switzerland
otto.preiss@ch.abb.com

Abstract. One of the main objectives of developing component-based software systems is to enable efficient building of systems through the integration of components. All component models define some form of component interface standard that facilitates the programmatic integration of components, but they do not facilitate or provide theories for the prediction of the quality attributes of the component compositions. This decreases significantly the value of the component-based approach to building dependable systems. If it is not possible to predict the value of a particular attribute of a system prior to integration and deployment to the target environment the system must be subjected to other procedures, often costly, to determine this value empirically. For this reason one of the challenges of the component-based approach is to obtain means for the "composition" of quality attributes. This challenge poses a very difficult task because the diverse types of quality attributes do not have the same underlying conceptual characteristics, since many factors, in addition to component properties, influence the system properties. This paper analyses the relation between the quality attributes of components and those of their compositions. The types of relations are classified according to the possibility of predicting properties of compositions from the properties of the components and according to the influences of other factors such as software architecture or system environment. The classification is exemplified with particular cases of compositions of quality attributes, and its relation to dependability is discussed. Such a classification can indicate the efforts that would be required to predict the system attributes which are essential for system dependability and in this way, the feasibility of the component-based approach in developing dependable systems.

1 Introduction

Component-based development (CBD) is of great interest to the software engineering community and has achieved considerable success in many engineering domains. CBD has been extensively used for several years in desktop environments, office applications, e-business and in general in Internet- and Web-based distributed applica-

R. de Lemos et al. (Eds.): Architecting Dependable Systems III, LNCS 3549, pp. 257 – 278, 2005.

tions. The component technologies (for example COM/DCOM, CORBA, EJB and .NET) used in these domains originate from object-oriented (OO) technologies. The basic principles of the OO approach, such as encapsulation and class specification have been further extended. The importance of component interfaces has increased; a component interface is treated as a component specification and the component implementation is treated as a black box [26]. A component interface is also the programmatic means of integrating the component in an assembly. Component technologies include the support of component deployment into a system through the component interface. On the other hand, the management of the component quality attributes has not been supported by these technologies. This topic has instead been treated separately from the applied component-based technologies.

In many other domains, for example dependable systems, CBD is utilized to a lesser degree for a number of different reasons [3]. One is the difficulty of implementing the same component technologies because of various system constraints such as limited resources which is one typical characteristic of small embedded systems. Another reason is the unclear distinction between system components which include both hardware and software parts and software components which may be encapsulated in system components or distributed through several system components. In this article, whenever we use the term “components” we assume “software components”. Finally an important reason is the inability of component-based technologies to deal with quality attributes as required in these domains. For dependable systems, a number of quality attributes are as important as the functions these systems provide, and the development effort related to realizing quality attribute requirements is most often greater than the effort related to the implementation of particular functions. In general, the problem of CBD for dependable systems is that, if components are considered black boxes, it is difficult to obtain evidence that they behave according to their specifications. Moreover, depending on the deployment and usage context a component’s behavior might change. Dependability arguments can be obtained only if the complete behavioral specification of a component is known beforehand. If the advantages of component-based technologies are limited to the functional domain only and cannot be utilized in the domain of quality attributes, or, even worse, introduce difficulties in the management of quality attributes, these technologies cannot be fully utilized.

The component-based approach is closely related to software architecture. The use of a component-based technology decreases chances to get an architectural mismatch by standardizing certain architectural decisions. A software component model specifies rules for component composition and interoperation and in this way simplifies the development process and similar to software architecture makes it possible to reason about quality properties largely independent of a particular application. The main difference between a software component-based approach and the software architecture-oriented approach is that the former focuses on reusability of already existing components, whereas the latter focuses on a conceptual approach in identifying components, their interconnections and evaluation of overall configuration.

Some of the main advantages of CBD are reusability, higher abstraction level and separation of the system development process from the component development process [3,4]. These advantages have however implications on other aspects of software and system development. The final success of the utilization of CBD depends not only on its advantages but also on these implications – the degree to which they are positive and negative. Since for dependable systems, particular quality attributes are of the greatest importance, a question which arises is to what extent does CBD influence the achievement of these properties: CBD can introduce new difficulties, it can be irrelevant for those properties, or can have a positive effect. For this reason it is of interest to analyze the ability of CBD to cope with requirements related to quality attributes.

Component-based software engineering (CBSE) faces two types of problems in dealing with quality attributes. The first, common to all software development, is the fact that the quality attributes are often imprecisely defined or difficult to estimate and measure. Further, values of certain properties may be different in different contexts. The second, specific to component-based systems, is the difficulty of relating system properties to component properties. In CBD one desired feature is that components can be selected and integrated in an automatic and efficient way. This goal is achieved for the functional part; components are selected and integrated through their interfaces. It is questionable if a similar approach can be applied to quality attributes.

For component-based systems crucial questions in relation to quality attributes are the following:

- Given the system quality attributes required**,** which attributes are required of the components concerned and which attributes are required from the component design- and runtime infrastructure?
- Given a set of component attributes, which system attributes are determined?
- How can the quality attributes of a system be accurately predicted, from the quality attributes of components which are determined with a certain accuracy.
- To which extent, and under which constraints are the emerging system attributes determined by the component attributes?

These and similar questions have been addressed at a series of CBSE symposia [4], and particular models of certain properties have been analyzed [14], but so far very little work has been done in the systematization and classification of quality attributes in accordance with the questions above. Although there are other classifications of quality attributes such as [6,7,17,23], these have not considered the predictability and composability aspects of the quality attributes.

Some system quality attributes can be derived directly from the component attributes; others might require a complex calculation model, related to the component model and the system architecture. Some system attributes, such as safety, do not exist on the component level and are the result of a complex combination of the system interaction with its environment, system architecture and different attributes of components involved.

In this paper, our intention is to demonstrate the diversity of quality attributes and the different methods which can be used for predicting system properties from the properties of the components involved. The quality attributes can be classified according to our ability to accurately calculate their compositions, i.e. the ability to predict the properties of component compositions. Such a classification indicates the feasibility of the component-based approach for building dependable systems**.**

The paper is organized as follows: Section 2 provides basic definitions needed for a classification and a classification framework which is used in our CBD-specific classification. Section 3 identifies the types of properties according to the principles for predicting the properties of component assemblies. Section 4 discusses the proposed classification with respect to a possibility of combining the types identified in the previous section, and with respect to recursive composition. Section 5 analyzes composability of properties of dependable systems and discusses possible benefits for dependable models of utilizing CBD and Section 6 concludes the paper.

2 Composability of Quality Attributes

In this section we will take a more fundamental look at what quality attributes or properties are and what they are good for. We then investigate the various notions of property decompositions, so that we can properly position our empirical and composition-oriented classification of properties.

2.1 What Are Properties?

The discussion in this paper is not primarily concerned with the theories behind individual types of properties (such as what is *green*, or what is *having a latency of,* or *what is security*). It is rather concerned with how we can generalize our understanding of the notion of property or its synonyms to a level where we can suggest a principled manner to conceptualize them in the context of software systems and software components where we can suggest a principled manner to reason about them in a decompositional way.

2.2 The Philosophical View on Properties – What Are Properties Good for?

Coming to grips with properties is pervasive in philosophy. Plato's theory of Forms ([19], p. 93) (where Form is said to be Plato's term for property,) seems to be one of the earliest accounts on what is today called properties. The term property includes attributes, qualities, features, or characteristics of things. It even encompasses relations such as *being faster than*.

The need for properties is motivated by their explanatory roles they have to fill. They came into being to describe phenomena of *interest* (like when we say: the system response is *very fast*). Because a stakeholder is a role that represents groups of people who have similar interests in the same phenomenon, the choice of properties and their importance is clearly related to certain stakeholders or stakeholder classes.

From an ontological viewpoint, the existence of properties is determined empirically. As a first important rule this means that properties and their definitions are conceived by humans and there is no a priori, logical or conceptual method to determine which properties exist [24]! This also means that the notion of a property and every type of property is an abstract concept only. As with any concept, humans define its name as well as its definition and its related theories. We therefore do not have to argue about the universal truth behind or correctness of a property as long as its definition and purposeful theory fulfills our goals.

From a natural language viewpoint, there is no single idiom to talk about and use properties. In other words, properties are distinct from their representations and *the same property may have different representations*. In the English language for example, properties can appear as single terms with any of the many suffixes such as '-ity', '-ness', '-hood', '-kind', '-ship', etc. (e.g. as in '*safety'*), or as predicative expressions in multiple ways (e.g., '*executes safely', 'is safe'*). Hence, since properties are conceived by humans, not only their meaning need to be defined but also their possibly numerous, not only natural language-based, representations.

As with any concept (i.e. purely knowledge-related construct), humans tend to categorize also properties, i.e. we describe properties by means of inherent characteristics of certain categories. Two such inherent characteristics that are very important in the discourse of "quality attribute compositions": complexity, and specificity.

Complexity refers to the fact that properties can be simple or compound/complex. A complex property is some form of a logical structure or combination of properties. This combination must of course be defined to understand a complex property. As an example for a complex property consider 'being my grandfather'. It implies that the person this property is ascribed to is 'male and older than I am'. Or 'being CMM Level 3 certified' implies that the software development unit this property is ascribed to 'has a software process that is documented and standardized, and that all software projects use an approved, tailored version of this standard software process for developing and maintaining software'[22].

Specificity refers to the fact that a property can be a determinable or a determinate. The distinction, however, is relative in that a determinate property is a more specific version of a determinable. For example, "up-time" is a determinate property (i.e. a more specific one) of the determinable "availability". The measure "time passed between failures" is in turn one possible determinate of "up-time". The hierarchy of determinables and determinates is generally expected to bottom out in completely specific, absolute determinates. In software engineering, such leave determinates would be called quality-carrying properties, or direct properties, or tangible/measurable properties, to name a few.

In software engineering, hierarchies or taxonomies of properties, which are fundamentally based on the notions of complexity and specificity, are at the heart of the decompositional approaches of quality models (e.g. [8] and its predecessors).

2.3 Realization-Oriented Decomposition Versus Other Forms of Property Decompositions

Before we proceed to the patterns of realization-oriented decomposition/traceability in the next main chapter, we want to clarify by means of Fig 1 how two other types of property decompositions are related to the realization-oriented decomposition: (1) classification oriented quality attribute decompositions, and (2) the analysis-oriented decomposition for non-functional requirements.

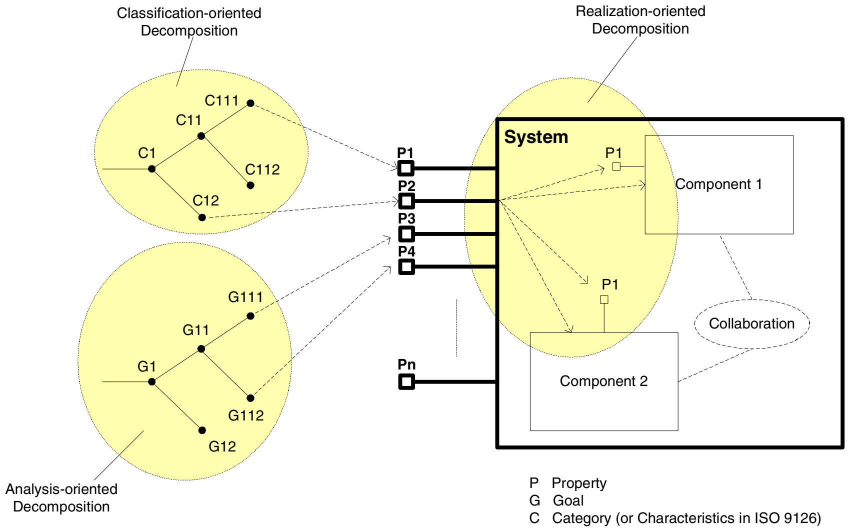
Classification-oriented Decomposition
C111
C11
C1
C112
C12
Analysis-oriented Decomposition
G111
G11
G1
G112
G12
Realization-oriented Decomposition
P1
P2
P3
P4
Pn
System
P1
Component 1
P1
Collaboration
Component 2
P Property
G Goal
C Category (or Characteristics in ISO 9126)

Fig 1. Different Types of Property Decompositions

Fig 1 shows a *System* and its ascribed properties *P1…Pn*. The System is composed of two components (*Component 1* and *Component 2*) that engage in a *Collaboration*. In this simple example, every component has just one property *P1*. If we envision a designer who needs to design a *System* with the required properties *P1…Pn*, the constituents of the *System* would be called the realization elements.

In a **realization-oriented decomposition** we want to relate a system-level property to the elements that realize the system and that cause the property to manifest in the requested way. Fig 1 illustrates a simple case in which the *Component 1* and *2* and their respective property *P1* realize the system-level property *P2*. Let us take the simple case where *P2* of the *System* expresses its power consumption in Watts. *P1* of *Component 1* and *2* would simply be the respective consumption per component. Hence, *P2* of the *System* is no more than the sum of the two properties *P1* of the two components. This is of course the simplest case and it is, in fact, the subject of the rest of this paper to elaborate on other types of realization-oriented decompositions.

A **classification-oriented decomposition** on the contrary refers to a hierarchy represented as a tree of determinables and determinates, where the leaf determinates could be selected as the relevant, required properties of a system. Hence, it is a classification that serves the purpose of knowledge structuring. It represents a decomposition of high-level properties into more tangible ones so as to end with a set of quantifiable properties on some scale. The ISO/IEC 9126-1 [8] is a representative for such a classification because it defines a set of characteristics, which are decomposed into subcharacteristics, which in turn shall be decomposed into potentially measurable properties. Such a classification can therefore serve as starting point for defining the system-level properties to be realized. In Fig 1, such a classification is used to derive the required properties *P1* and *P2* of the *System*. For instance, *P1* could be the required physical property power consumption, whose value must be below a certain threshold. *P1* could have resulted from the ISO/IEC 9126-1 derived classification Efficiency (*C1*) -> Resource Utilization (*C11*) -> Power Consumption (*C111*).

The third kind of decomposition - **analysis-oriented decomposition** – is shown in the figure for completeness reasons only. It relates to the decomposition of requirements. For more details of this category and in general on these classes of decomposition refer to [18].

2.4 Definitions of Certain Terms

We feel that terms such as non-functional property, extra-functional property, quality attribute, etc. are very often not used carefully enough. Based on our research, we would suggest the following distinction which we used in this paper.

- *Attribute/property* are treated as synonymous and are used in the most general sense as defined by standard dictionaries, e.g.: "a construct whereby objects and individuals can be distinguished" [15] "a quality or trait belonging and especially peculiar to an individual or thing" or "an effect that an object has on another object or on the senses" [16]
- A *required attribute/property* is expressed as a need or desire on an entity by some stakeholder. We may call such a property a *requirement.*
- An *exhibited attribute/property* is an attribute/property ascribed to an entity as a result of evaluating the entity. The evaluation may be direct, in the sense that one does some measurement with the entity in question, or it may be indirect. The latter may be the case when we ascribe a property to an entity because we evaluate related artifacts or because someone made us believe that the entity has this (typically conceptual) property, although we can hardly measure it on the entity itself.
- *Quality*: The totality of exhibited attributes/properties of an entity that bear on its ability to satisfy stated or implied needs, i.e. to satisfy its requirements. Quality thus represents the set of all exhibited attributes/properties that have a relationship to required properties.
- *Quality attribute/property*: Refers to an exhibited attribute/property that is part of the Quality of an entity.

Having discussed the basic classes of property decompositions that are being used today, we can now focus on the conceptualization of the realization-oriented decompositions that go along with building software-intensive systems based on software components.

3 Classification of Properties

A great number of quality attributes are encountered in software engineering. They are classified in many different ways, frequently in a non-orthogonal manner. One example of classification is related to the system lifecycle: run-time properties (visible and measurable during the program execution) and lifecycle properties (those that characterize different phases in a development and maintenance process). Another example is the quality model defined in ISO/EIC 9126-1 "Software engineering -

product quality” standard [8], which classifies quality attributes as external and internal. Quality attributes are the measurable, quantifiable properties of a software product. The latter also includes all its intermediate development artifacts. Quality attributes that refer to the internal quality – internal quality attributes - are typically applied to intermediate deliverables at certain development stages (e.g. attributes of a design specification, source code, etc.). Internal therefore has the connotation of “development internal view”. The relation between internal and external quality attributes is not unambiguous though; an internal quality attribute may have impact on different external quality attributes and of course an external quality attribute is a result of combination of internal attributes.

The classification we consider here is related to *composability*. We classify properties according to the principles applied in deriving the system properties from the properties of the components involved. Instead of the term “system”, we shall use a generic term *Assembly (A)* which simply denotes a set of interacting components. Such an assembly can be a part of a software system (for example a functional unit, or a subsystem), or the entire system. The only characteristic we want to relate to an assembly is a set of integrated components – an assembly can be assumed as a component (however composed of other components). Some properties, however, cannot be related only to an assembly, but are explicitly related to the entire system and its interaction with the environment. In such cases we refer to a *System (S).*

We distinguish the following types of properties:

a. *Directly composable properties.* A property of an assembly which is a function of, and only of, the same type of property of the components involved.

b. *Architecture-related properties.* A property of an assembly which is a function of the same type of property of the components and of the software architecture.

c. *Derived properties.* A property of an assembly which depends on several different properties of the components.

d. *Usage-depended properties.* A property of an assembly which is determined by its usage profile.

e. *System environment context properties.* A property which is determined by other properties and by the state of the system environment.

Let us discuss these cases and give examples in the following subsections.

3.1 Directly Composable Properties

Definition: *A directly composable property of an assembly is a function of, and only of the same property of the components.*

$$
\begin{aligned}
&P = \text{attribute}, \;\; A = \text{assembly}, \quad c = \text{component} \\
&A = \{c_i : 1 \le i \le n\} \\
&P(A) = f(P(c_1), P(c_2), \dots, P(c_n))
\end{aligned}
\tag{1}
$$

Note that the property of the assembly is the same as the component property. Further, the component technology is not explicitly specified in the relation (1). However it is obvious that the function *f* itself is dependent on the technology since the mechanisms to assemble components is provided by the component technology.

An example of a property of this type is the static memory size of a component or an assembly, this is also known as the memory footprint. The simplest composition model is the calculation of the static memory of an assembly as the sum of the memories used by each component:

$$M(A) = \sum_{i=1}^{n} M(c_i) \tag{2}$$
$$M = \text{memory size}, A = \text{assembly}, c_i = \text{components}$$

The function M(ci) is different for different technologies. For example in the case of the separation of composition time from run-time which is usually used in embedded systems, M(ci) will be a constant, possibly parameterized by configuration factors. In such cases the static memory size of an assembly will be a constant. A more complicated model can be found in the Koala component model [25], in which additional parameters, such as size of glue code, interface parameterization and diversity are taken into account (i.e. the parameters determined by the component technology used).

The equation (2) is also valid for a dynamic memory, with the difference that M(ci) is not a constant, but a function which may depend on the usage profile. When using a particular technology, design patterns or parameterized resources this function may be limited on a particular value or budgeted. In such a case the total amount of memory can be calculated.

$$M(A) \le \sum_{i=1}^{n} M_{\max}(c_i) \tag{3}$$

The properties of this type can be calculated directly from the component properties if the components comply with particular restrictions for memory allocation. These restrictions can be built in component technologies.

For this type of composition there are no other assumptions and therefore these properties are the easiest to specify and predict. This does not mean that the composition functions are easy or even possible to express formally. However the fact that the property is visible on component and assembly level, and that the assembly property is dependent only on the component properties simplifies the prediction procedure and makes the prediction valid in any application using these components.

3.2 Architecture-Related Properties

Definition: *An architecture-related property of an assembly is a function of the same property of the components and of the software architecture.*

$$\begin{aligned} &A = \{c_i : 1 \le i \le n\} \\ &P(A) = f(P(c_1), P(c_2), \ldots, P(c_n), SA) \\ &SA = \text{software architecture} \end{aligned} \tag{4}$$

In this case the assembly properties depend not only on the component properties but on the architectural structure. The software architecture is often used as a means for improving particular properties without changing the component properties. These types of properties can be tuned by different architectural solutions or variations. An example of such a property is a performance predictability model for J2EE (Java 2 Platform, Enterprise Edition) application presented in [9,29]. A typical application implemented in this technology would be a distributed web-based application in which the variability in scalability is achieved by it being possible to add new clients and new computational (business) components to the server as illustrated in Fig 2. To achieve concurrency the components are executed in different threads. A possible extension variation of this architecture is the possibility to include several nodes with web servers and business applications.

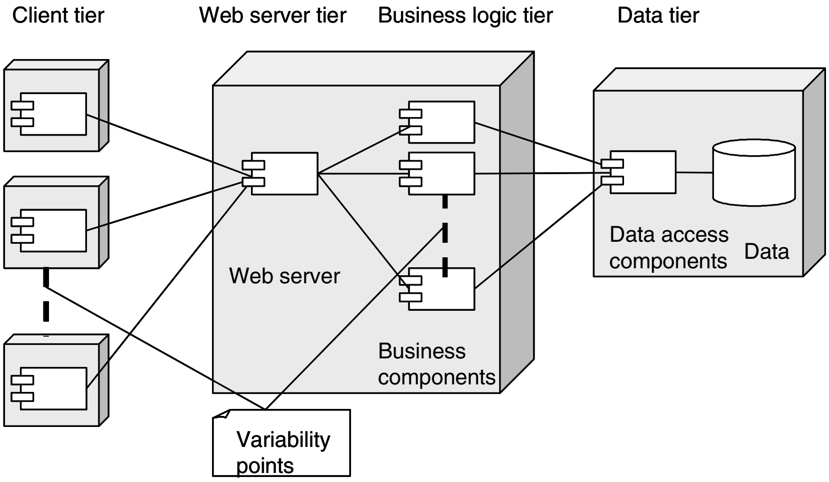

Fig. 2. A typical multi-tier architecture with client and servers variability points affecting the performance quality property

The performance of the system shown in the Fig 2 is related to the number of clients and the number of server components. A typical requirement for such applications is the performance and scalability, i.e. the dependencies between the performance and number of clients and active business components.

According to [9,29] the time per transaction T/N expressed in equation (5) depends on several factors related to the system architecture: The first factor comes from the concurrent requests that compete for service from the server component. This includes the network bandwidth and underlying transport mechanisms. The second factor describes a case in which accepted requests compete for a thread to execute the business components. The third factor results from concurrent access to the database by the concurrent server threads.

The first factor is proportional to the number of clients, the second to the number of clients and inversely proportional to the number of threads (i.e. number of components on the server) and the third factor is proportional to the number of threads.

$$\begin{aligned}
&T / N = ax + b\frac{x}{y} + cy \\
&T / N = \text{execution time per transaction} \\
&x = \text{number of clients; } y = \text{number of components} \\
&a, b, c = \text{proportional factors for a particular implementation}
\end{aligned} \tag{5}$$

The form of the equation shows that it is possible to calculate the optimal number of threads in relation to the number of clients to achieve a minimum respond time per transaction.

3.3 Derived Properties

Definition: *A derived property of an assembly is a property that depends on several different properties of the components.*

$$\begin{aligned}
&A = \{c_i : 1 \le i \le n\} \\
&P(A) = f\begin{pmatrix} P_1(c_1), P_1(c_2), \cdots, P_1(c_n), \\ P_2(c_1), P_2(c_2), \cdots, P_2(c_n), \\ \vdots \\ P_k(c_1), P_k(c_2), \cdots, P_k(c_n) \end{pmatrix} \\
&P = \text{assembly attribute} \\
&P_1 ... P_k = \text{component attributes}
\end{aligned} \tag{6}$$

In the same way that a function of an assembly is more than the sum of the component functions, there are properties that are the result of the composition of different component properties.

An example of such a property in a real-time system is the end-to-end deadline (a maximal response time) that is a function of different component properties, such as worst case execution time (WCET) and execution period as shown in the following example. Let us consider real-time port-based component models with provided and required interfaces and interfaces to an underlying operating system or I/O devices, as discussed in [5,10,28]. In these models, components are implemented as tasks, parts of a task or a set of tasks. An assembly consisting of two components, where every component is realized as a task is shown on Fig 3. Each basic component includes properties such as WCET and execution period. A composition of this simple model is achieved by connecting ports and identifying provided and required interfaces.

The question is whether we can calculate WCET for an assembly of components executing with different periods. In a case in which the execution periods are the same, this would be possible. In a case in which these periods are different, we cannot specify WCET of the assembly, but we can specify end-to-end deadline and a period.

An end-to-end deadline is the maximum time interval between the start of the first component in an assembly and the finish of the last component in the assembly. The assembly period will be a number to which the components periods are divisors.

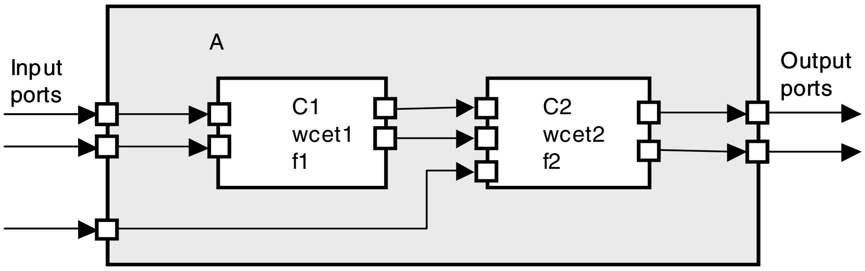

Fig. 3. Composition of port-based components

In a similar way we can calculate latency, or response time, from the real-time properties of components if particular assumptions about real-time system characteristics, such as scheduling policy, and mapping between component and real-time entities are taken. In a case in which components are mapped to tasks and the fixed priority scheduling is used, a worst case latency of component c_i, L(c_i), can be calculated as [11]:

$$L^{n+1}(c_i) = c_i.wcet + B(c_i) + \sum_{\forall c_j \in hp(c_i)} \left\lceil \frac{L^n(c_i)}{c_j.T} \right\rceil c_j.wcet \qquad (7)$$

B is the blocking time, $hp(c_i)$, is the set of components having tasks with higher priority than component i, $c_j.T$ is the period and $c_j.wcet$ is the worst-case execution time of component c_i.

Emerging properties, i.e. properties that are pertinent on a system (or an assembly) level but are not visible on the component level are of special interest in this category. For such properties the major challenge is to identify the properties of the components that have impact on them.

3.4 Usage-Dependent Properties

Definition: *A Usage-dependent property of an assembly is a property which is determined by its usage profile.*

$$\begin{aligned} &P(A,U_k) = f(P(c_i, U'_{i,k})) :\ i,k \in N \\ &P = \text{attribute for a particular usage profile} \\ &U_k = \text{assembly usage profile} \\ &U'_{i,k} = \text{component usage profile} \end{aligned} \qquad (8)$$

The behavior of an assembly and consequently of a system depends not only on the internal properties of the components and their composition but also on the particular use of the system. A usage profile U_k which determines a particular attribute P_k must be transformed to the usage profile $U'_{i,k}$ to determine the properties of the components.

Properties of this type introduce particular problems as they depend on the use of the system. This means that the component developers must predict as far as possible the use of the component in different systems – which may not yet exist. A second problem is the transfer of the usage profile from the assembly (or from the system) to the component. Even if the usage profile on the assembly level (U_k) is specified, the usage profile for the components ($U'_{i,k}$) is not easily determined especially when the assembly (and the system) configuration is not known.

A particular problem with this type of property is the limited possibility of reusing measured and derived properties. If the usage profile is changed, the properties must be re-calculated or re-measured. An example of such property is reliability which in software is calculated or measured for particular usage profiles. The question arising here is the possibility of reusing previous specifications of the property [5]. The first thought would be that this is possible if the domain of the new usage profile is a sub-domain of an old usage profile. In this case the value of a property will be within the range of possible values of the property for the old usage profile; the local maximum and minimum value being in the range of values for the old usage profile (see Fig 4).

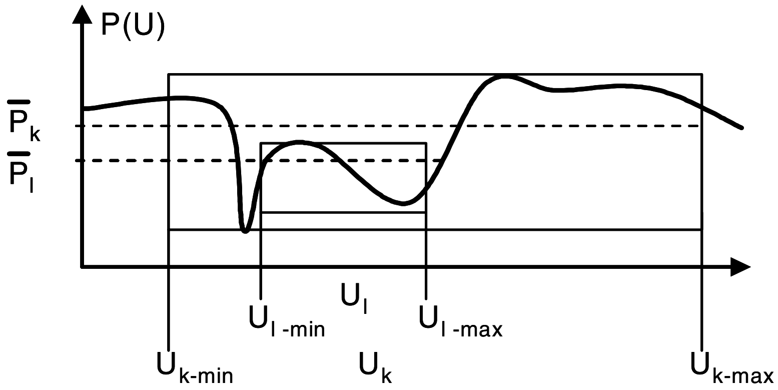

Fig. 4. Property for different usage profiles

If the new requirements of a property in a new usage profile are equal of or less stringent than the old requirements, we can use the property value from the old usage profile. This means, for example, that we do not need to measure the component properties.

$$U_l \subseteq U_k \Rightarrow P_{k-\min}(A,U_k) \le P_l(A,U_l) \le P_{k-\max}(A,U_k) \tag{9}$$

In a case in which a property is expressed as a statistical value (such as a mean value), the property value in an interval can be changed in an unwanted direction; Fig 4 illustrates such example in which the mean value of the property P(U) in the interval [U_{l-min}, U_{l-max}] is lower than in the entire interval [U_{k-min}, U_{k-max}], although the minimum and maximum values are higher. For certain properties (such as availability, or different quality of services) in certain domains (for example multimedia) the average plays a more important role than min or max values.

3.5 System Environment Context Properties

Definition: *A System Environment Context property is a property which is determined by other properties and by the state of the system environment.*

$$\begin{aligned} &P(A,U_k) = f(P(c_i,U'_{i,k}),C_k): \ i,k \in N \\ &P = \text{attribute for a particular usage profile} \\ &U_k = \text{system usage profile} \\ &C_k = \text{system context} \\ &U'_{i,k} = \text{component usage profile} \end{aligned} \tag{10}$$

The property depends not only on the system property determined by the usage profile, but also on the environment in which the system is used (denoted by C_k in (10)). This case of composition is very much related to the "Usage dependent properties" type of composition, because the set of system profiles include a subset of all usage profiles. However, the property itself can be different in different contexts (i.e. surrounding environment) in which the system is placed. By this we emphasize that it is not possible to determine the value of the property even the if the usage profiles are known. An example of such a property is safety. As the safety property is related to the potential catastrophe, it is obvious that in different circumstances, the same property may have different degrees of safety even for the same usage profile. We can argue that these properties are out of the scope of the predictable assembly, as they depend on the surrounding environment. In contrast to "compositionally" deriving assembly properties from component properties, the approach for such kinds of properties is more like "given the system environment and the system properties, what are the requirements on the assembly and component properties. Nonetheless, system environment context properties are also dependent on component properties. Further, for most of the cases an environment can be assumed or even be required for a usage profile.

A system can exhibit numerous properties and certainly not all of them have the same characteristics; some are easy to perceive and measure while others are very difficult to analyze, or measure (for instance administrability). Analyzable properties, which can be measured, are potential candidates for automatic reasoning about the behavior of a system. Properties that depend on the environment in which a system is deployed are generally hard to derive from the component properties.

4 Composition of Properties

For a classification it is important that it is complete and orthogonal. The completeness assumes that all cases fit into the classification. The orthogonally means that a particular case belongs to and only to a particular type in the classification. The presented classification is an idealization and an abstraction; in systems, in particular in complex systems, we could have many properties that are important for the stakeholders but are by their nature not precisely and formally specified, and which have different manifestations at component and system levels. While the question whether there are other types of properties, i.e. that there are types which do not fit into the classification cannot be formally justified or falsified, we can certainly find properties whose composition is the result of a combination of the principled types described in the previous section. For this reason it is of interest to see which combinations of these basic types are feasible and which combinations are of fundamental character. Further, there is a question related to recursive composability. Similarly to the question "can we provide component models that support recursion, in which we treat assemblies as components?" we can state a question: which properties and under which constraints are recursively composable?

In this section we discuss these two aspects of composability: composition of different types of properties and recursive compositions.

4.1 Composition Combination of Different Types of Properties

We are analyzing here a possibility of combining basic types of properties; can a system property be a result of a combination of different composition types? Theoretically we can have 26 combinations (single, double, triple, fourfold and fivefold combinations) of basic property types. Some of the combinations do not make sense. For example, a derived (emerging) property by definition cannot be at the same time a directly composable property. Similarly, combinations between directly composable and usage-dependent, or system environment-related properties are not feasible. This reduces the number of combinations. Further we shall see that some of the combinations cannot be found in practice.

In [11] we have analyzed and classified many properties grouped with respect to different concerns and validated the classification by inquiring a dozen researchers through a questionnaire to classify almost 100 properties. Since in general the properties and their definitions are the result of concerns, limitations and requirements it is possible to find an arbitrary number of different properties. To make the questionnaire manageable we have collect properties in groups, which correspond to different concerns (such as performance, dependability, usability, business, etc.). The results of the questionnaire indicated that there are many properties, in particular emerging properties, which are a combination of two, three or more basic classification types.

Here follows all possible combinations of the basic types of properties and identify those which we have never seen in practice (indicated by N/A), and give examples of possible combinations. From Table 1 we can see that a rather small number of combinations seem to be feasible.

Table 1. Examples of properties that are related to combinations of basic types of properties (a. Directly composable properties (DIR), b. Architecture-related properties (ART), c. Derived properties (emerging properties) (EMG), d. Usage-depended properties (USG), and e. System environment context properties (SYS))

No	a)	b)	c)	d)	e)	Concerns/Properties
	DIR	ART	EMG	USG	SYS	Examples
1	x	x				Performance/Scalability
2	x		x			N/A
3	x			x		N/A
4	x				x	N/A
5		x	x			Performance/Timeliness
6		x		x		Dependability/Reliability
7		x			x	N/A
8			x	x		N/A
9			x		x	N/A
10				x	x	Dependability/Security
11	x	x	x			N/A
12	x	x		x		Performance/Responsiveness
13	x	x			x	N/A
14	x		x	x		N/A
15	x		x		x	N/A
16	x			x	x	N/A
17		x	x	x		Dependability/Security
18		x	x		x	N/A
19		x		x	x	N/A
20			x	x	x	Dependability/Safety
21	x	x	x	x		N/A
22	x	x	x		x	Business/Cost
23	x	x		x	x	N/A
24	x		x	x	x	N/A
25		x	x	x	x	N/A
26	x	x	x	x	x	N/A

4.2 Recursive Composition of Quality Properties

We have used the generic term "assembly" for a set of integrated components. In section 2 it is shown that specification of some properties should distinguish between an assembly and a system; the difference is whether only internal parameters are assumed, or a combination of internal and external factors, not part of the system, are included. A software system may consist of a set of assemblies, which turns out to be a set of components. Several questions arise when composing assemblies: Can the assemblies composed be treated as components in the new assembly, or are they treated in the new assembly as a set of the original components loosing the assembly identity? A similar question we can state for a properties; can a property be expressed in a recursive form of (the same) properties of hierarchical components? An ideal situation would be to have a means of using a hierarchical and recursive model which permits the same reasoning on all levels of the hierarchy.

We can distinguish two types of assemblies supported by existing component technologies. The first is the 1st order assembly which is not treated as a component in the component model. This type of assembly is merely a set of components integrated together, creating an application or a part of an application. In this case an assembly is seen as a virtual boundary of the component set and not as a separate entity. An assembly of the 1st order does not follow the semantics of a component. The second type of assembly is hierarchical which means that the assembly, created from components, is treated as a new component inside the component model.

There are different criteria which must be satisfied if an assembly is to be treated as a component. The basic criteria are the ability to provide recursive principles on (i) operational (construction) interface, (ii) component deployment and (iii) component quality properties.

The way to obtain the property value of an assembly is different from obtaining assemblies from components, and a recursive composition of properties is not related to the (recursive) constructions of assemblies. Rather it depends of the type of the property. For example the directly composed properties are by definition recursive; for recursive assemblies these properties will be recursive. In this way a property of an assembly of assemblies will be a composition of assembly and component property functions. For example, the properties of type (a) from the section 3 will be derived in the following way:

$$\begin{aligned} &P(A_a) = f(P(A_k) = f(f(P(c_i))); i,k \in N \\ &A_a = \{A_k\};\ A_k = \{c_i\} \\ &P = property, A_a = assembly\ of\ assemblies \\ &A_k = assemblies, \quad c_i = components \end{aligned} \tag{11}$$

For the memory consumption case in equation (2), we have:

$$M(A_a) = \sum_{i=1}^{k} M(A_i) = \sum_{i=1}^{k} \sum_{j=1}^{n} M(c_{ij}) \tag{12}$$

For derived properties, it is in general not possible to achieve recursion. The same is valid for component properties which are not relevant on the assembly level.

5 Composability of Dependability Properties

To illustrate the property classification, we take dependability as an example. Dependability is defined as the ability of a system to deliver service that can be trusted and the ability of a system to avoid failures that are more severe and frequent than are acceptable to the users. According to [1] dependability is a complex property including six basic attributes, namely, availability, reliability, safety, confidentiality, integrity and maintainability.

The questions of interest to component-based software engineering or development are:

- To which category belong the dependability properties? In particular, which of the dependability properties are emerging or derived system properties, and which are both system and component properties?
- How are these properties in a component-based system related to other component properties?
- To which extent (and how) can these properties can be determined from component properties?
- To which extent can the unpredictability of these properties be minimized and how much is it related to the uncertainty of the component properties?

Reliability

The definition of reliability originates from the probability that a system will fail within a given period of time. The probability of failure is directly dependent on the usage profile and context of the module under consideration. One possible approach to the calculation of the reliability of an assembly is to use the following elements [20,21]:

- Reliability of the components – Information that has been obtained by testing and analysis of the component given a context and usage profile;
- Usage paths – Information that includes usage profile and the assembly structure. Combined, it can give a probability of execution of each component, for example by using Markov chains.

A model based on this approach needs the means for calculating or measuring component reliability and an architecture which permits analysis of the execution path. Component models that specify provided and required interface make it possible to develop a model for specifying the usage paths. This is an example in which the definition of the component model facilitates the procedure of dealing with the quality attribute. The system reliability can be analyzed by (re)using the reliability information of the assemblies and components (which can be derived or measured).

Availability

Availability is defined as the probability of a module being available when needed. The difference between reliability and availability is that availability is not only dependent of the system properties but also on a repair process, which implies that the availability of an assembly cannot be derived from the availability of the components in the way that its reliability can be derived from the reliability of its components. If the repair rate of the components are known, it also must be known the repair time of the system integration. In a larger context, non run-time attributes must be taken into a consideration; availability is related to the maintenance and support of the components constituting the assembly.

Safety

Safety is an attribute involving the interaction of a system with the environment and the possible consequences of the system failure [12]. It is a system attribute, neither a component nor an assembly attribute. Its safety depends on where and how the system is deployed. Since safety is a system attribute that is dependent on the system's environment, a means for analyzing safety is a top-down architectural approach, a decomposition rather than composition. Examples of such approach can be found in [2,27] In the analysis process, the components' attributes are used as selection criteria or are identified as demands that should be met. For this reason a component-based approach might not have the apparent advantage – on the contrary, if the starting idea is a reuse of existing components, the components' attributes cause new constraints and in this way might decrease the system safety. However, when the constraints are identified and unambiguously related to the constraints on the system level, the system safety can increase. Also, some attributes, such as reliability, might improve the accuracy of the system safety prediction, especially if known or measured when used in other applications.

Confidentiality and Integrity

Security properties, confidentiality and integrity, defined as follows apply to dependable systems [1].

- Confidentiality is defined as a measure of the absence of unauthorized disclosure of information;
- Integrity is defined as the absence of improper system state alterations.

From the definitions it is apparent that these attributes are not directly measurable and composable, and this is the main obstacle to the development of a theory for their prediction. Confidentiality and integrity are emerging system attributes that can be tested and analyzed on the system and architectural level but not on the component level. Usage profiles can be used for testing and analysis, but it is impossible to automatically derive these attributes from the component attributes.

Maintainability

Maintainability is related to the activities of people and not of the system itself, although there exists self-repairable systems which in some cases can reconfigure themselves in order to continue to provide services. Component technologies might provide support for dynamic upgrading/deployment of components which can improve the maintainability of a system. In this case the maintainability is much a matter of component technology, and not of the component itself. The system architecture thus has an impact on maintenance.

There are many parameters that can be measured and then used to estimate the maintainability of a code (for example McCabe Metrics for complexity [13]). These parameters can be identified for each component. It is however not clear how these parameters can be defined on the assembly level. One possibility is to define a mean value of all components normalized per lines of code.

6 Conclusion

The full advantage of the component-based approach to developing software will only be achieved when, in addition to a compositional reasoning of a system's functionality, we are able to more easily and accurately predict the system behavior with its quality attributes. When systems are designed and build from components, many system properties can be derived from the component properties. Hence, a generic support for the definition and measurement of the properties, which is built into the component models and technologies, would be greatly welcomed. However, the predictability of properties does not depend only on such a support in the component models but more on the types of properties themselves. Consequently, there is no silver bullet to deal with all types of properties. For each type of property, a theory of the property, its relation to the component model, composition rules and their contextual dependence and relation to requirements must be known.

Dependability properties belong to a class of properties which compositions are the most difficult; they are system properties and are result of different properties on component level, and system usage context. The feasibility of a bottom-up approach is questionable, but a more feasible challenge is to achieve an incremental composability when adding a new or modifying a component in a system, and being able to reason about the system properties from the properties of the old system and the properties of new component.

Because no generic approach will do, the paper suggests a classification of properties according to their principled way of compositional reasoning. Each type of the classification is characterized by the required parameters for obtaining predictability on the system level. Some types show clear composable characteristics, while others are not directly related to compositions.

The existing component models differ considerably and how the assemblies' and components' properties are treated will be highly dependent on these models, especially for those properties that are directly composable or are related to the architecture. For example, if the component model has independently deployable components with a 1st order assembly model, it is likely that the properties of the components cannot be propagated further than the assembly level without considering the environment.

In spite of diversity of properties, technologies, and theories, it should be possible to create reference frameworks that by identifying type of composability of properties can help in estimation of accuracy and efforts required for building component-based systems in a predictable way. These frameworks can be built for particular component-models in combination with architectural solutions and particular domains. Our future work will continue in these directions in which different component technologies and architectural solutions in the domain of embedded systems, such as automotive or automation systems will be considered.

Acknowledgements

We would like to express our gratitude to the reviewers for their excellent analysis of the paper and their valuable suggestions to improve it.

References

1. Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C., "*Basic concepts and taxonomy of dependable and secure computing*", IEEE Trans. Dependable Sec. Comput., Vol. 1, Issue 1, 2004
2. Bondavalli, A., Chiaradonna, S., Cotroneo, D., Romano, L.: Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications, *IEEE Trans. Dependable Sec. Comput.* Vol. 1, Issue 4, 2004
3. Bouyssounouse, B., J. Sifakis (Eds), "*Embedded Systems Design - The ARTIST Roadmap for Research and Development*", Springer-Verlag, LNCS Vol. 3436, 2004
4. Crnkovic I., Schmidt H., Stafford J., and Wallnau K. C., "5th Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly", In *Software Engineering Notes*, Vol. 27, Issue 5, 2002.
5. Crnkovic I., and Larsson M., *Building Reliable Component-Based Software Systems*, Artech House, 2002.
6. Dromey G.R., "A Model for Software Product Quality", In *IEEE Transaction on Software Engineering*, Vol. 21, Issue 2, 1995.
7. ISO/IEC, *Information technology - Software product quality - Part 1: Quality model*, report ISO/IEC FDIS 9126-1:2000 (E), ISO, 2000.
8. ISO/IEC, "*Software engineering - Product quality - Part1: Quality model*", ISO/IEC, International Standard 9126-1:2001(E).
9. Jogalekar P., and Woodside M., "Evaluating the Scalability of Distributed Systems", *IEEE Transactions on Parallel & Distributed Systems*, Vol 11, Issue 6, 2000.
10. Hissam S. A., Hudak J., Ivers J., Klein M., Larsson M., Moreno G. A., Northrop L., Plakosh D., Stafford J., Wallnau K. C., and Wood W., *Predictable Assembly of Substation Automation Systems: An Experience Report*, report CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, 2002.
11. Larsson M., *Predicting Quality Attributes in Component-based Software Systems,* PhD Thesis, Mälardalen University Press, Västerås, Sweden, March 2004
12. Leveson, N., "*Software: System Safety and Computers*", Addison-Wesley, 1995
13. McCabe T.J., "A Complexity Measure", *IEEE Transaction on Software Engineering*, Vol. 2, 1976.
14. Moreno G. A., Hissam S. A., and Wallnau K. C., "Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling", *Proceedings of 5th ICSE workshop on CBSE*, 2002.
15. Miller, G. A. (2002). WordNet®. Cognitive Science Laboratory, Princeton University [Online]. Available: http://www.cogsci.princeton.edu/~wn/
16. Merriam-Webster (2004, May), "*Collegiate Dictionary: Merriam-Webster Online*", [Online]. Available : http://www.m-w.com/dictionary.htm
17. Preiss O., Wegmann A., and Wong J., "On Quality Attribute Based Software Engineering", *Proceedings of EUROMICRO 2001, Component Based Software Engineering workshop*, IEEE, 2001.
18. Preiss O., *Foundations of Systems and Properties: Methodological Support for Modeling Properties of Software-Intensive Systems*, PhD Thesis Nr. 3013, Swiss Federal Institute of Technology, Lausanne, May 2004.
19. Russell B., (2002, July 20). The Problems of Philosophy. Home University Library [Online]. Available: http://www.ditext.com/russell/russell.
20. Schmidt H. and Reussner R. H., "Parametrized Contracts and Adapter Synthesis", *Proceedings of 5th ICSE workshop on CBSE*, 2001.

21. Schmidt H., "Trustworthy components: compositionality and prediction", In *Journal of Systems & Software*, Vol. 65, Issue 3, 2003.
22. SEI. (2003, November). Capability Maturity Model® for Software (SW-CMM®). Carnegie Mellon Software Engineering Institute [Online]. Available: http://www.sei.cmu.edu/
23. Svahnberg M., *Supporting Software Architecture Evolution*, Ph.D. Thesis, Blekinge Institute of Technology, Sweden, 2003.
24. Swoyer, C.. (2002, July). Properties. The Metaphysics Research Lab, Stanford University [Online], http://www.science.uva.nl/~seop/entries/
25. van Ommering R., "The Koala Component Model", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, Artech House, 2002.
26. Szyperski C., "*Component Software - Beyond Object-Oriented Programming*", Second Edition, Addison-Wesley/ACM Press, 2002
27. Valérie Issarny, Apostolos Zarras: Software Architecture and Dependability, *LNSFM 2003*, LNCS Vol. 2804
28. Wall A., Larsson M., and Norström C., "Towards an Impact Analysis for Component Based Real-Time Product Line Architectures", *Proceedings of Euromicro Conference on Component Based Software Engineering*, IEEE, 2002.
29. Yan L., Gorton I., Liu A., and Chen S., "Evaluating the scalability of enterprise javabeans technology", *Proceedings of 9th Asia-Pacific Software Engineering Conference*, IEEE, 2002.