Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 166 (2007) 27–46
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Methodological Approach to Choose Components in Development and Evolution Processes

Bart George[1]

VALORIA Laboratory
University of south Brittany
Vannes, France

Régis Fleurquin[2]

VALORIA Laboratory
University of South Brittany
Vannes, France

Salah Sadou[3]

VALORIA Laboratory
University of South Brittany
Vannes, France

Abstract

One of Software Engineering's main goals is to build complex applications in a simple way. For that, software components must be described by their functional and non-functional properties. Then, the problem is to know which component satisfies a specific need in a specific composition context, during software development or evolution. We claim that this is a problem of substitution, and we propose a need-aware substitution model that takes into account functional and non-functional properties.

Keywords: Software components, non-functional properties, substitution, ideal component, discrepancy.

1 Introduction

Component-oriented programming should allow us to build or maintain a software like a puzzle whose parts would be units "subject to composition by a third party"

[1] Email: bart.george@univ-ubs.fr
[2] Email: regis.fleurquin@univ-ubs.fr
[3] Email: salah.sadou@univ-ubs.fr

doi:10.1016/j.entcs.2006.06.014

[19]. Examples of such units are COTS (*Components-Off-The-Shelf*), which are commercial products from several constructors and origins. When one develops and maintains component-based softwares, some problems occur, and we have noticed two main ones: how to select, during development of such softwares, the most suitable component in order to satisfy an identified need? And during maintenance, if this need evolves, will the chosen component remain suitable, or will we replace it?

We think that these two problems are related to a substitution problem. In fact, when one conceives or maintains an application, needs appear. And to describe them, the designer or the maintainer can imagine *ideal components*. These are virtual components representing the best ones satisfying these specific needs. Then the problem is to find the concrete components which are the closest to the ideal ones. In other words, trying to compose or maintain components means trying to substitute ideal components by concrete ones.

However, composition doesn't concern only functional properties. Most components are "black boxes" which must describe not only functional, but also non-functional properties. As every software needs a certain quality, one cannot think about composing components whose non-functional properties are unknown, and at the same time hope to have its quality requirements satisfied anyway. In embedded systems in particular, non-functional properties are as important as functional ones [6]. This is why substitution must take functional and non-functional properties into account.

So, how to substitute? Some may say we just have to use sub-typing, as some object-oriented languages made it a general way of substitution. However, an ideal component doesn't describe "general" needs: it describes an application's specific needs in their *context*, a notion that is absent from objects. Let us explain what we mean by "context". If we take a need, modeled by an ideal component, we will try to find a concrete one to substitute it. Now, let us suppose that we already found a suitable component. We may need to check if there isn't another one better than the first one. However, we will not try to substitute the old candidate by a new one, because the key notion isn't the candidate, but the need it is supposed to satisfy. Moreover, if this need changes, a former candidate may not be suitable any more. So substitution of an ideal component by a concrete one is performed only in the context of an application and its needs, modeled by the ideal component. This is why a candidate component can replace another one without any subtyping relationship between them, as every candidate is compared only to the ideal component.

The remain of this paper will be organized as follows: in the next section we discuss about substitution in development and evolution processes. In section 3, we describe the generic component and quality models used in all our definitions. Then, we define a component-oriented substitution model, including substitutability rules for every functional and non-functional element of our model (section 4). In section 5, we illustrate the possibilities of such a model by using a short application example. Before concluding, we describe related work (section 6).

2 Substitution in development and evolution processes

2.1 Substitution need in the component life cycle

A component-based approach differs from a non-component-based approach in the assumption that component-based systems are built from preexisting components. Consequently, we must distinguish two development processes: in the first one, we will focus on designing components "for reuse", while in the second one, we will focus on designing systems "by reuse" (by finding the proper, already built, components and verifying them). The first process should be achieved when the second one starts. For the second process, Ivica Crnkovic *et al.* considered the well-known V development process and adapted it for component-based development [7]. The different phases of this process are: requirements analysis and specification, system and software design, components selection and testing (which replace implementation and unit testing in non-component-based approaches), system integration, system verification and validation, and operation support and maintenance.

Whereas, in a non-component-based approach, much effort is required for implementation, in a component-based approach the effort is not in implementation but in selection of components. This seems simpler, but as told in [6] two problems appear: First, we must have a process for finding and evaluating components; Second, if the selected components do not totally fit with our overall design, we must be able to adapt and test them before integrating them into the system. These two problems constitute, indeed, the problematic of our substitution approach: which component is the best match for the need, and if this best component cannot substitute exactly the ideal one how far is it from the ideal one?

2.2 Substitution method

As we saw at previous subsection, we need a way to select a concrete component among many others, according to the functional and non-functional specification of an ideal one. We have to rank concrete components using quantitative and qualitative measures. But to compute this sort of "distance", called in the remaining part of the paper "discrepancy", we need a way to associate a value to the presence, partial respect or absence of a property. Also, between several properties we need to select the most important ones. Consequently, the definition of an ideal component not only concerns its functional and non-functional properties, but also a comparison framework enabling the computation of the discrepancy between itself and a concrete candidate. This two-level structure is established by the system's architect who knows which are the most valuable properties of its component in the system architecture. This is why our approach targets "specialists", i.e. designers who already know what they need, because they will be able to model ideal components with precision.

In order to organize substitutions, do we perform a parallel substitution or a sequential one? More precisely, do we substitute all ideal components by concrete ones at the same time, or do we substitute one ideal component after another? Indeed, when we use the notion of an ideal component, you have to "forget" the other

components in the system. Let us consider the case where A is the best candidate for the needs of ideal component X, and component B is the best match for the needs of ideal component Y, but A and B are not well suited to work together (e.g. their interaction provokes a deadlock). So, A has a good enough reliability, and so does B, but not when A works with B. To cope with this kind of problem the substitution process should be sequential. We consider that the system's architecture is built incrementally. As we said, our notion of ideal component describes an application's need in the context of this application. This means that each ideal component is specified according to what we know about the "concrete" part of its application. More precisely, every time a concrete component substitutes an ideal one, the concrete architecture is modified, so the next ideal component is modified and specified according to this new concrete architecture.

3 Component and quality models

Definitions given in this paper are placed in the following framework: one component model, holding a type system such as Java for EJB, and one quality model such as ISO 9126 standard [14]. In this framework, we suppose the existence of metrics to measure non-functional properties (such as those defined in [21]), so that our contribution will focus only on the substitution model definition. We also consider only "black-box" components and do not take their internal structure into account, so we do not distinguish "primitive" components from "composite" ones.

3.1 The generic model

Our goal is not to give yet another definition of what a component is, or what non-functional properties are. It is to define a component-oriented substitution that we can apply on many existing component and quality models. That is why we prefer to give generic models, on which we can apply our substitution concepts.

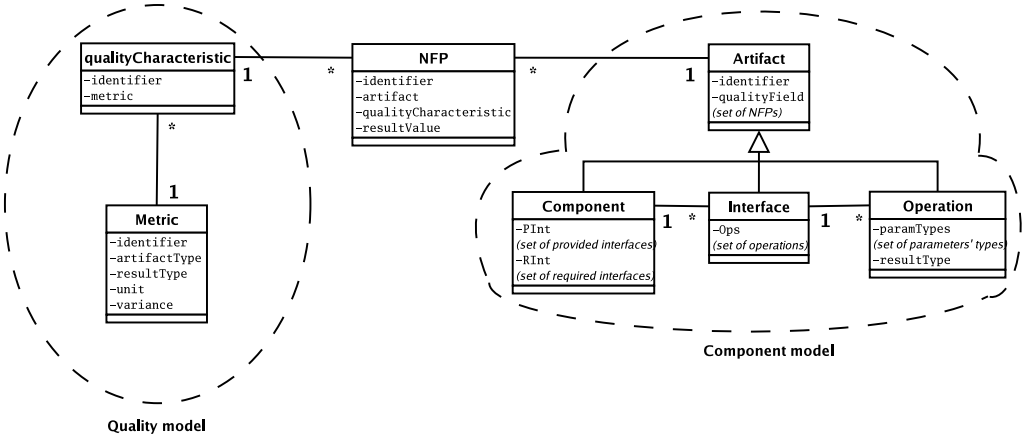

Fig. 1. Generic model

The generic component model includes component **artifacts**, representing the

component's architectural elements, which are common to most existing component models, and which have non-functional properties. As shown in figure 1, we chose to keep three kinds of component artifacts: components themselves, interfaces, and operations. A component contains provided and required interfaces, and interfaces contain operations. In the remaining part of the paper, we refer to **a candidate component** and **a substitutable component**, when the first one tries to substitute the second one. Their elements are called respectively **candidate elements** and **substitutable elements**. When we find the best candidate for the substitution, we say the substitutable component or element can be **replaced** by this candidate.

Beside the component model, we define a generic quality model. Its elements are quality characteristics (such as those from ISO 9126 [14]), and metrics. We use existing metrics to evaluate and compare non-functional properties (see [11] for a survey). But why metrics ? In the literature, several methods for defining and evaluating non-functional properties already exist (see [1] for a survey). But such methods usually focus on one specific property, or family of properties, for example quality of service, which is only a part of the whole software quality. Metrics may be applied to many families of properties, and allow comparisons. This is why we think that in our case, metrics represent the best method for comparing different non-functional properties.

A component's quality properties are based on our generic quality model. We start by describing elements of this quality model in the next subsection, before introducing their link with the elements of the component model.

3.2 Elements of the quality model

This quality model is composed of two elements: quality characteristics which represent non-functional properties, and metrics, which measure these characteristics (see left part of Figure 1). For the remaining part of this paper, we consider that a metric may measure several quality characteristics (as proposed in the IEEE standard 1061-1998 [13]), but each characteristic is measured by only one metric. Elements of the quality model are defined as follows:

3.2.1 Quality characteristics:

A quality characteristic, or simply characteristic, represents a given quality property, preferably a fine-grained attribute (such as latency), because of our claim that only one metric can measure such a characteristic.

3.2.2 Metrics:

A metric contains a set of quality characteristics it measures. It also contains a set of artifact types on which it can be calculated (for example: {`component, interface`}), the result's type, and its unit. The metric's variance explains the relation between the metric's result and the evaluated quality characteristic. For example, if a metric calculates an execution time, the variance stipulates that the lower the value is, the better it is.

Two metric values are *comparable* only if the metrics are the same. So having two "comparable metric values" M_1 and M_0 means that we have the same metric M, and we try to compare the value of M on the candidate artifact A_1 with the value of M on the substitutable artifact A_0. Having two comparable metric values M_1 and M_0, we can check if M_1 is *superior to* M_0 *according to the variance*. For example, if the metric type is an integer representing the execution time in milliseconds, then its variance is decreasing. In this case, if M_1 is greater than M_0 according to integer comparison, M_1 is in fact inferior to M_0 according to M's variance.

3.3 Non-functional properties

A component artifact is linked to a quality element using a **non-functional properties** (noted **NFP**). An artifact may be related to several quality elements, so several different NFPs can belong to a same artifact. An NFP describes the effect of a quality characteristic on the artifact it belongs to, and uses the metric applied to the latter. Several NFPs of a same component artifact may share the same metric, but not the same characteristic.

In Figure 1, the *resultValue* attribute of an NFP is given by the metric's measurement on the artifact. In the case of an ideal component, this attribute value is given by the application's designer.

One can try to compare two NFPs only if the artifacts they belong to are of the same kind and comparable (see next subsection for comparison definitions). Two NFPs are comparable if they refer to the same characteristic. Two NFPs are equal if they are comparable and their *resultValue* attributes are equal.

3.4 Artifacts

The main element of our generic component model is the artifact. All artifacts, whatever their kind is, have a *quality field*, which is a set of NFPs. Two artifacts' quality fields are comparable if, for each NFP of one quality field, there is one comparable NFP in the other quality field. Two quality fields are equal if for at least one NFP of one quality field, there is an equal NFP in the other quality field, and *vice versa*.

Let us now describe the different kinds of artifacts:

3.4.1 Operations.

An interface's operation is defined by its *signature*, also called a *type*. An operation's type is defined by the set of its parameters' types $(\alpha_1, \ldots, \alpha_n)$[4] and its result's type β. It is noted $(\alpha_1, \ldots, \alpha_n) \longrightarrow \beta$.

Two operations are comparable if their signatures are comparable. Two operation signatures T and U are comparable if they are equal or if there exists a type substitution V so that $V.T$ equals to U, or T equals to $V.U$.

[4] For reasons of simplicity, in the current version of our model we do not take into account parameters' order.

For example, if we consider Java's *Object* type, signature $Object \longrightarrow Object$ may be replaced by $Integer \longrightarrow Integer$ if we let *Integer* substitute *Object*. It corresponds to Zaremski and Wing's exact and generalized signature matching for functions [22].

Two operations are equal if their signatures are equal modulo the renaming of the type names, and if their quality fields are equal.

3.4.2 Interfaces.

A component's interface is defined by a set of operations.

A candidate provided interface PI_1 is comparable to a substitutable provided interface PI_0 if for each operation of PI_0 there exists a comparable operation in PI_1. A candidate required interface RI_1 is comparable to a substitutable required interface RI_0 if for each operation of RI_1, there exists a comparable operation in RI_0. Two interfaces (provided or required) are equal if their quality fields are equal and if, for each operation of one interface, there exists an equal operation in the other interface, and *vice versa*.

3.4.3 Components.

A software component is defined by a set of provided interfaces and a set of required interfaces.

A candidate component C_1 is comparable to a substitutable component C_0 if for each provided interface of C_0 there exists a comparable provided interface of C_1, and for each required interface of C_1, there exists a comparable required interface of C_0. If C_1 is not comparable to C_0, it can not substitute C_0.

4 Substitution model

The goal of our model is to measure what distinguishes a candidate component from an ideal one. And the way we chose to achieve this goal is to compare NFPs' result values obtained on artifacts.

4.1 Weights, penalties and discrepancy

For each NFP, we attach a **weight** (or comparison weight) noted $Comparison_P$, and a **penalty** noted $Penalty_P$ (P being the NFP). These two values define the NFP's importance for the artifact it belongs to. The higher these two values are, the more important this NFP is. If a substitutable artifact owns an NFP and a candidate artifact owns a comparable one with a superior value, the candidate's chances increase proportionally with the comparison weight. Else, the penalty will be used to sanction this lack. A candidate component may also bring his own new NFPs that the substitutable component doesn't have. These new elements will be evaluated by the ideal component designer.

The **substitution discrepancy**, or discrepancy, is defined using these weights, penalties, and NFP' *resultValue*s. This discrepancy will inform on the substitutabil-

ity of an NFP or an artifact. The best candidate for substitution is the one with the lowest discrepancy. If the discrepancy is negative, the candidate element can be considered as "better" (in terms of quality) than the substitutable one, according to the current context. If the discrepancy is positive, then the candidate is worse. If the discrepancy equals to 0, then the two compared elements are "equivalent", but it does not mean that they are equal.

For each ideal component, there is a **maximal discrepancy** for substitution, fixed by its designer. Let us consider a component C_1, a candidate for the substitution of another component C_0. If the substitution discrepancy between C_1 and C_0 is bigger than the maximal discrepancy associated to C_0, then C_1 will be rejected.

4.2 Normalizing comparisons of NFPs' result values

For each NFP P, the *resultValue* attribute is given by P's metric's measurement on the artifact that owns P. The problem is that into a same artifact belonging to an ideal component, an NFP P_X can measure, for example, a screen resolution characteristic whose metric is in millions of pixels, while an NFP P_Y can have a ratio metric whose value is between 0 and 1. Let us suppose we found a comparable artifact belonging to a candidate component and owning two NFPs P_A and P_B that fit respectively to P_X and P_Y. The problem is: how can we put together into a same discrepancy algorithm a comparison of resolutions, whose result can be (for example) in millions of pixels or in hundreds of thousands of pixels, and a comparison of ratios, whose result is between 0 and 1 ?

We may only adjust weights and penalties "roughly", according to different result values in order to have a coherent discrepancy value in the end (for example, put weight 0.00005 on P_X and weight 20 on P_Y), but it can lead to arbitrary decisions. Plus, even though our approach targets specialist users, our goal is to help them specify their needs correctly. This is why we need to normalize NFP's result values into a common balance so that users can focus on their weights and penalties according to this balance.

We consider two steps in normalization : the absolute (syntactic) one and the semantic one.

4.2.1 Absolute normalization

Let us consider a substitutable artifact A_0, one of its NFPs P_0, a candidate artifact A_1 that is comparable to A_0, and one of A_1's NFPs P_1 that is comparable to P_0.

Let us also consider the difference between P_0's and P_1's $resultValue$ attributes, denoter: $resultValue_{P_0} -_{Variance} resultValue_{P_1}$. It is a subtraction between $resultValue_{P_0}$ and $resultValue_{P_1}$ depending on their metric's variance. For example, if its type is integer or float and variance is increasing, the measurement will be equal to: $resultValue_{P_0}$ - $resultValue_{P_1}$. If variance is decreasing, it will be equal to: $resultValue_{P_1}$ - $resultValue_{P_0}$.

We are going to take this difference and transform it into a percentage value, which will be called the **absolute discrepancy**. Its goal is to rationalize what

separates a candidate NFP from a substitutable one.

The absolute discrepancy between P_1 and P_0 (denoted AD) is defined as follows:

$AD(P_1, P_0) = (resultValue_{P_0} -_{Variance} resultValue_{P_1}) / resultValue_{P_0}$

For example, if one substitutable NFP P_0, that belongs to an ideal component and measures a screen resolution, expects a 2 million pixels result value, and if one candidate NFP P_1 has a 1.5 million pixels value, this means the absolute discrepancy between P_1 and P_0 equals to 25%, or +0.25. One can have a negative absolute discrepancy. For example, if P_0's $resultValue$ equals to 2 million pixels while P_1's $resultValue$ equals to 2.5 million ones, the absolute discrepancy will be equal to -0.25.

4.2.2 Semantic normalization

The ideal component's designer has to give a sense to a syntactic discrepancy, by choosing its appropriated satisfaction level. So she/he has to give a satisfaction value, or **semantic discrepancy**, which is a function f taking an absolute discrepancy as parameter and whose result is a numeric value between 1 (the worst possible result) and -1 (the best possible result), whereas 0 means this is exactly what was expected. The absolute discrepancy can be amplified, or limited, or unchanged, by the semantic discrepancy, depending on what the designer chooses. This bounded interval between 1 and -1 allows her/him to harmonize several different metric comparisons on a common balance (it is up to the designer to define his satisfaction scale). Let us go back to the resolution example: if the designer considers that 0.25 is a too high absolute discrepancy, the semantic discrepancy can make it remain unchanged and equal to 0.25, which is bad according to the designer's satisfaction scale. However, if the designer considers that it is an acceptable low discrepancy, and that it is more or less what was expected, the semantic discrepancy can equal to 0, which is better.

The semantic discrepancy between P_1 and P_0 (denoted SD) is defined as follows:

$SD(P_1, P_0) = f(AD(P_1, P_0))$. f is a function projecting the syntactic discrepancy between P_1 and P_0 to a value included between 1 and -1. f is up to the designer's choice.

4.2.3 Substitution discrepancy between NFPs

The substitution discrepancy between P_1 and P_0 (denoted ND) is defined as follows:

$ND(P_1, P_0) = Comparison_{P_0} * SD(P_1, P_0)$

4.3 Substitution discrepancy between artifacts

Now that we managed to normalize discrepancies between NFPs, the global discrepancy between artifacts can be pragmatically brought to a global sum. So here, we will define a calculus that will give the discrepancy separating a candidate component (denoted C_1 for the remain of this section) from a substitutable component (denoted C_0 for the remain of this section) in a given context. This context is de-

fined by the weight and the penalty allocated to the NFPs of C_0's artifacts. So, before talking about discrepancy between artifacts, let us present the discrepancy between their quality fields.

We will suppose that there exists a relation $MIN_{x \in E}\ f(x)$, which selects an element x from a set of elements E so that the function $f(x)$ has the lowest value.

4.3.1 Discrepancy between artifacts' quality fields.

Let us consider a substitutable artifact A_0, a comparable candidate one A_1, and their quality fields (denoted Q_{A_1} and Q_{A_0}). The substitution discrepancy between these quality fields (denoted QD) is defined as follows:

$QD(Q_{A_1}, Q_{A_0}) = \sum_{P_0 \in Q_{A_0}} QDProp(Q_{A_1}, P_0)$ - $\sum_{P_1 \in Q_{A_1}} QDBonus(P_1, Q_{A_0})$

with:

$QDProp(Q_{A_1}, P_0) =$ - $ND(P_1, P_0)$ if $\exists\ P_1 \in Q_{A_1}$ that is comparable to P_0; else, $Penalty_{P_0}$.

and:

$QDBonus(P_1, Q_{A_0}) =$ 0 if $\exists\ P_0 \in Q_{A_0}$ that is comparable to P_1; else, a value given by C_0's designer.

To measure the discrepancy between the quality fields, we try to find for each P_0 a comparable NFP P_1 in A_1 (there can be only one, as NFPs of a same artifact cannot share the same characteristic). Substitutable NFPs without any comparable P_1 are taken into account through their penalty value $Penalty_{P_0}$. Candidate NFPs without any comparable P_0 are taken into account through a value given by C_0's designer.

4.3.2 Discrepancy between incomparable artifacts.

If two artifacts are incomparable, there will not be any substitution discrepancy measurement between them. In other words, substitution discrepancies are measured only for comparable elements.

4.3.3 Discrepancy between comparable operations.

Let us consider a substitutable operation O_0 and a comparable candidate operation O_1. The substitution discrepancy between them (denoted OpD) is defined as follows:

$OpD(O_1, O_0) = QD(Q_{O_1}, Q_{O_0})$

As long as O_1 and O_0 are comparable, the discrepancy between them is in fact the discrepancy between their quality fields.

4.3.4 Discrepancy between comparable provided interfaces.

Let us consider a substitutable provided interface I_0, a comparable candidate provided interface I_1, and their sets of operations Ops_{I_1} and Ops_{I_0}. The substitution discrepancy between I_1 and I_0 (denoted PID) is defined as follows:

$PID(I_1, I_0) = \sum_{O_0 \in Ops_{I_0}} MIN_{O_1 \in Ops_{I_1}} OpD(O_1, O_0) - \sum_{O_1 \in Ops_{I_1}} POBonus(O_1, I_0) + QD(Q_{I_1}, Q_{I_0})$

with:

$POBonus(O_1, I_0) =$ 0 if $\exists\ O_0 \in Ops_{I_0}$ that is comparable to O_1; else, a value given by C_0's designer.

To measure the discrepancy between the interfaces, we take into account only the lowest found discrepancy for each O_0. Candidate operations without any comparable O_0 are taken into account through a value given by C_0's designer.

4.3.5 Discrepancy between comparable required interfaces.

Let us consider a substitutable required interface I_0, a comparable candidate required interface I_1, and their sets of operations Ops_{I_1} and Ops_{I_0}. The substitution discrepancy between I_1 and I_0 (denoted RID) is defined as follows:

$RID(I_1, I_0) = - \sum_{O_0 \in Ops_{I_0}} MIN_{O_1 \in Ops_{I_1}} OpD(O_1, O_0) - \sum_{O_0 \in Ops_{I_0}} ROBonus(I_1, O_0) - QD(Q_{I_1}, Q_{I_0})$

with:

$ROBonus(I_1, O_0) =$ 0 if $\exists\ O_1 \in Ops_{I_1}$ that is comparable to O_0; else, a value given by C_0's designer.

The principle of discrepancy between required interfaces is symmetrical to the principle of discrepancy between provided interfaces. For provided interfaces, it is better to have I_1 providing better quality than I_0, whereas for required interfaces, it is better to have I_1 requiring less quality than I_0.

4.3.6 Discrepancy between comparable components.

Let us consider a substitutable component C_0, a comparable candidate component C_1, their sets of provided interfaces $PInt_{C_1}$ and $PInt_{C_0}$, and their sets of required interfaces $RInt_{C_1}$ and $RInt_{C_0}$. The substitution discrepancy between C_1 and C_0 (denoted CD) is defined as follows:

$CD(C_1, C_0) = \sum_{PI_0 \in PInt_{C_0}} MIN_{PI_1 \in PInt_{C_1}} PID(PI_1, PI_0) + \sum_{RI_1 \in RInt_{C_1}} MIN_{RI_0 \in RInt_{C_0}} RID(RI_1, RI_0) - \sum_{PI_1 \in PInt_{C_1}} PIBonus(PI_1, C_0) - \sum_{RI_0 \in RInt_{C_0}} RIBonus(C_1, RI_0) + QD(Q_{C_1}, Q_{C_0})$

with:

$PIBonus(PI_1, C_0) =$ 0 if $\exists\ PI_0 \in PInt_{C_0}$ that is comparable to PI_1; else, a value given by C_0's designer.

and:

$RIBonus(C_1, RI_0) =$ 0 if $\exists\ RI_1 \in RInt_{C_1}$ that is comparable to RI_0; else, a value given by C_0's designer.

To measure the discrepancy between the components, we take into account only the lowest found discrepancy for each PI_0 and for each RI_1. Candidate provided

(resp. substitutable required) interfaces without any comparable PI_0 (resp. RI_1) are taken into account through a value given by C_0's designer.

5 Example

Now let us take the example of an application that requires a Digital Video ("DV") camera component, with an interface for video stream and another one for camera control. It must also conform to the DV standard. The video camera example for software components has already been treated in [3] and [1].

5.1 Modeling an ideal component

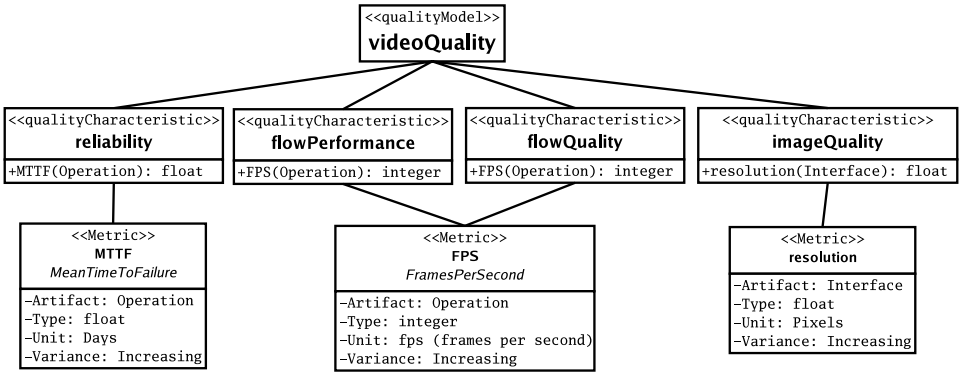

Fig. 2. Example of quality model.

The above requirements could be expressed by an ideal component called *videoCamera*. The latter contains a provided interface *videoStream* (with an operation *outputVideoFlow*), a provided interface *cameraControl* (with basic operations such as *on*, *record* and *eject* [5]), and a required interface *DVFormat* (with an operation *inputDVFlow* whose parameter is a DV tape).

The needs are not just about functional part, but also about non-functional properties and their respective importance. For example, we suppose that a high level of reliability for *record* and *eject* operations is required (so that the camera does not crash while recording, nor refuse to eject a video tape). We also assume that a high image quality, such as a 1 million pixels (1 MPixels) screen resolution, is required for *videoStream* interface. According to the quality model of Figure 2, we use the following characteristics: *reliability* and *imageQuality*. Their respective metrics are: *MeanTimeToFailure* (*MTTF*) and *screenResolution*. Then we attach to the ideal component several NFPs. To each operation of the *cameraControl* interface, we attach an NFP using *reliability* characteristic (*onReliability* for *on* operation, *recordReliability* for *record* operation, and *ejectReliability* for *eject* op-

[5] For simplicity and brevity reasons, we limit this provided interface to only three operations.

eration). To *videoStream* interface, we attach the NFP *cameraResolution*, using the characteristic *imageQuality*.

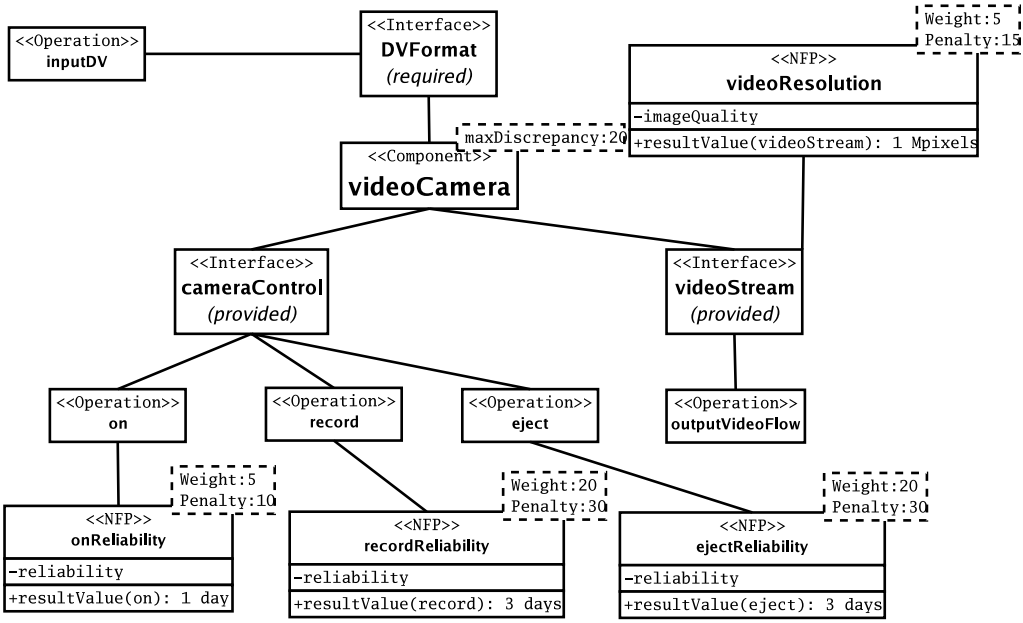

Fig. 3. Example of ideal component: *videoCamera*.

Finally, the designer fixes expected *resultValue*s, weights and penalties for each NFP, and also fixes a maximal discrepancy for the ideal component *videoCamera*. On Figure 3, we see that the expected value for *videoResolution* is 1 million pixels, and the expected values for NFPs using *reliability* characteristic vary from operation to operation. The values required for *recordReliability* and *ejectReliability* are higher than those for *onReliability*. The penalties attached to *videoResolution*, *recordReliability* and *ejectReliability* are very high in order to enforce candidate components to contain these NFPs. *videoResolution* has a low comparison weight, which means that a big difference on the image quality is not very important. However, *recordReliability* and *ejectReliability* have higher weights, which means that a big difference on the reliability measurements of *record* and *eject* is very important. The maximal discrepancy is fixed at a low level, so that the lack of one of these three NFPs in a candidate component will hardly be accepted.

5.2 Component lifecycle and substitution cases

Now that our ideal component is modeled, we can look for the best concrete candidate to substitute it. Here are the different substitution cases:

5.2.1 First composition.

Trying to plug a component into an application (in order to satisfy a given need) means trying to make this concrete component substitute the ideal one (corresponding to this need). Let us take the video camera example. Now that we modeled

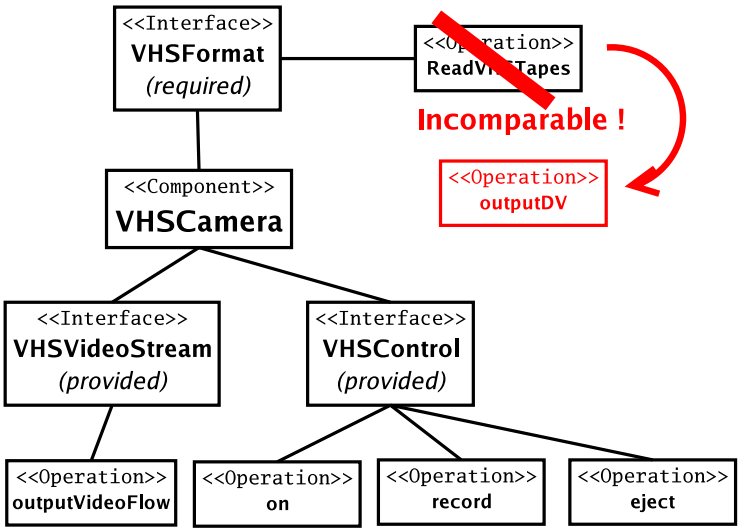

Fig. 4. Example of rejected candidate: $VHSCamera$.

an ideal camera component, we have to check which concrete camera is the best candidate to substitute it.

First, according to our substitution model, a candidate must meet all the functional requirements, i.e. it must have all the ideal component's provided services (interfaces and operations), and must not bring more required ones. Otherwise, it will be rejected even if it has a higher quality. For example, let us consider a $VHSCamera$ component (Figure 4) meeting all functional requirements, except one (it requires VHS tapes instead of DV ones). No matter its quality, we need a camera that requires only DV tapes, and this candidate adds a required interface, so it is rejected.

Then, a candidate, like the $fluidCamera$ component on Figure 5, may add new NFPs unanticipated by the ideal component designer. For example video flow's number of frames per second. That corresponds to the metric FPS (for Frames Per Second), which measures $flowPerformance$ and $flowQuality$ characteristics (all of them are shown in Figure 2). It may be interesting to have a new NFP using $flowQuality$ characteristic on the $outputVideoFlow$ operation, but the candidate ($fluidCamera$) misses an important NFP. The penalty is so high that it is rejected.

We can also have candidates providing at the same time some lower qualities, and other higher ones, than ideal component. In this case, a candidate component would rather have good "scores" in the most important NFPs. For example, let us take a candidate $goodImageCamera$ (Figure 6) which has an excellent image quality (2 million pixels instead of 1 million) and an average reliability (2.5 days instead of 3 for operations $record$ and $eject$), while candidate $reliableCamera$ shown in Figure 7 has an average image quality and an excellent reliability. We are not directly comparing them to find which one is "better" than the other. We are comparing each one of them, separately, with the ideal component, in order to find if it is an acceptable candidate. In the case of $goodImageCamera$, its syntactic index is -1, or -100%, so the designer can be very satisfied and give a semantic

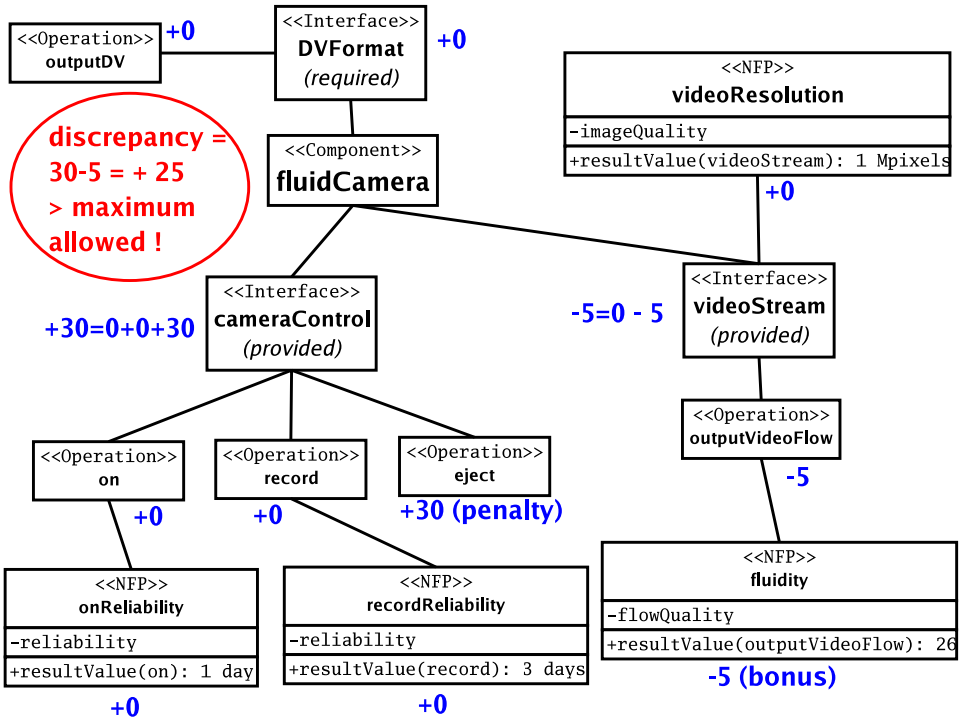

Fig. 5. Example of rejected candidate: *fluidCamera*.

index equal to -1. However, the candidate's values for reliability are a bit lower, so the designer can decide to sanction them by a 0.5 semantic index. In the case of *reliableCamera*, the opposite happens: the designer can be very unsatisfied with resolution, but reliability is much more important, and this candidate is much better than expected. So if we consider this ideal component, and the discrepancy obtained for each one of the candidates, we can say that both are acceptable, but the *reliableCamera* is the best one.

5.2.2 Maintenance with constant needs.

The application now has its camera component, but it could have a "better" one. The needs are the same, the context is the same, so the ideal component is exactly the same, but we can have new candidates. So we have to compare each one of them to this ideal component, ignoring the previous candidate. We are brought back to the first composition scheme.

Let us take the last candidate we showed, *reliableCamera*, and let us suppose it has been chosen in a first place. If we see a new candidate (let us call it *goodAndReliableCamera*), which proposes the same reliability as *reliableCamera*, but has a 1 Mpixels resolution. Once again, we do not compare the two concrete components' qualities directly, so we are not trying to check if *goodAndReliableCamera* is "better" than *reliableCamera*. But the designer can be more satisfied by this new candidate, which brings the expected resolution, and a much better reliability than expected. So *goodAndReliableCamera* is more likely

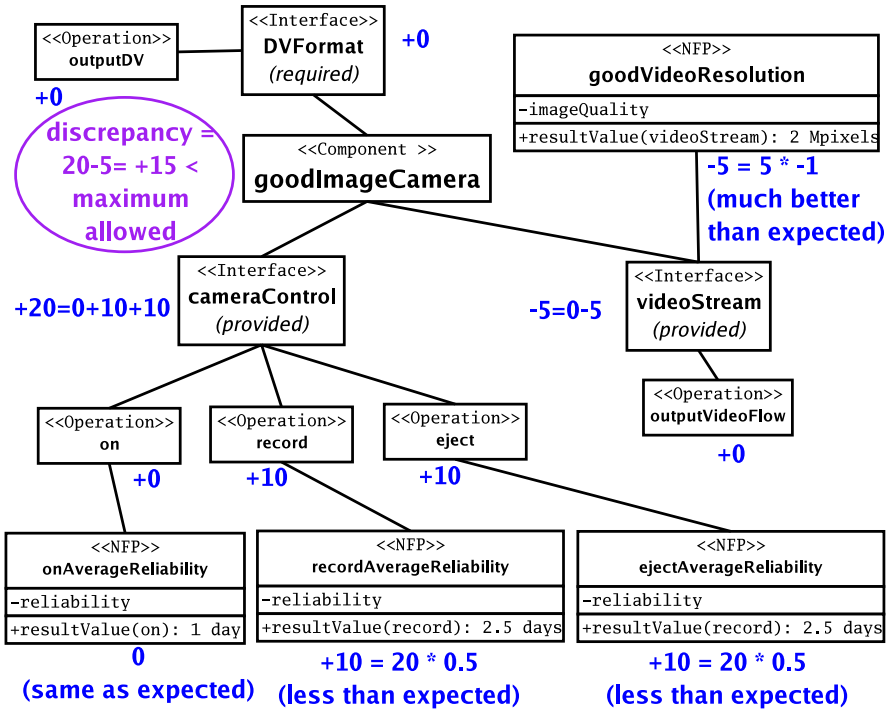

Fig. 6. Example of accepted : *goodImageCamera*.

to have a better "score" and correspond better to the ideal component as it is described.

5.2.3 Maintenance with evolving needs.

The application still has its camera component, but the needs have changed now. So we must ask the following question: is the currently chosen component still the best candidate for substituting the new ideal one, which models evolving needs ? And after all, is it still an acceptable component ? In fact, "evolving needs" may mean several different things.

First, it can be the need for a new functional service, such as a new artifact. For example, the ideal camera component may not need to read DV tapes any more, but VHS tapes only. Which means the removal of $DVFormat$ required interface and its replacement in the ideal component by a $VHSFormat$ required interface. In this case, the only candidate we have which functionally corresponds to this new ideal component is $VHSCamera$, previously rejected (Figure 4). But its acceptance will depend on its quality.

It can also be the need for new non-functional properties, such as a new NFP in the ideal component. For example, the ideal camera component may now need to evaluate the number of frames per second. This means that we need a new NFP using the FPS metric and measuring the $flowPerformance$ or the $flowQuality$ characteristic (see Figure 2). In this case, $fluidCamera$, previously rejected (Figure 5), may meet the new non-functional requirements, on some conditions. For

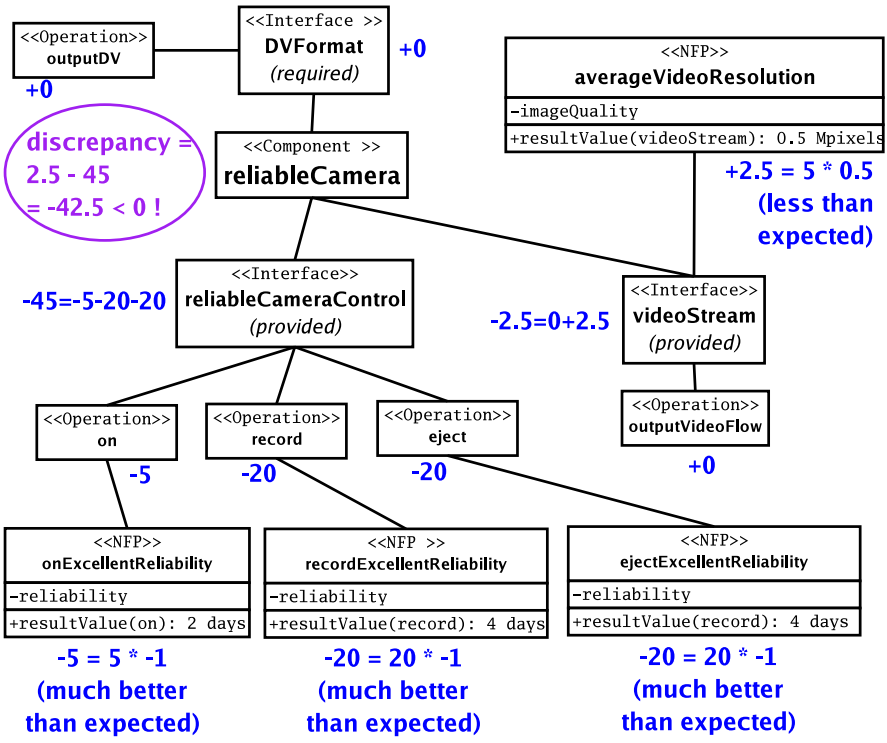

Fig. 7. Example of accepted : *reliableCamera*.

example, a low *resultValue* and a high comparison weight for the new NFP, and *flowQuality* as measured characteristic (else, the NFPs are not comparable).

Finally, it can be the need for re-evaluated qualities. The ideal camera component may have modified NFP values and/or modified weights and penalties. For example, in the case of resolution we can have a better expected value (such as 1.5 Mpixels) or a lighter weight (20 instead of 5), and in the case of reliability we can have a lighter weight for the NFPs that measures it (for example, 5 instead of 20). With these readjusted values, candidates *goodImageCamera* (Figure 6) and *reliableCamera* (Figure 7) would be accepted again. But this time, *goodImageCamera* would be more likely to be accepted than *reliableCamera*.

6 Related work

In introduction, we said that substitutability is a well-known problem in object-oriented languages which include typing [5] and subtyping [15]. It is also an industrial problem, as referred in [20], who asks how to make sure that changes on a component won't affect existing applications of a component, and tries to answer this question by setting rules based on subtyping. It was tempting to base our work on subtyping too, in order to substitute components [18]. But we took critics of typing [17] and subtyping [19] into account. Especially the one which said that they were too rigid and too restrictive for componentware, and couldn't deal with context. This is why we preferred to try a more flexible approach.

Premysl Brada explored the notions of deployment context and contextual substitutability [4]. A deployment context of a component is a sub-component that contains the used part of its services (provided and required services that are bound to other components). So Brada's contextual substitutability consists in comparing a candidate component with this sub-component, rather than the whole one. Although these notions seem close to ours, we work at a different level. Brada's approach consists in finding an "architecture-aware" form of substitutability, his context concerns a concrete component, and depends on its deployment in global architecture. Our approach is rather "need-aware", and our context considers an ideal component (modeling a need) and a concrete one which could substitute it.

As we said, our substitution model was inspired by Zaremski and Wing's specification and signature matching for library components [22,23]. Their matching takes into account some substitution schemes that subtyping doesn't include. We were close to this approach, but we went further, by taking context and non-functional properties into account, and applying our substitution rules on generic component models. Beside Zaremski's and Wing's approach, there are other notable works in software reuse and component retrieval [16]. For example, our notion of weights can be compared to Scott Henninger's tools [12]. These tools parse a source code, extract "components" from several keywords, then put them into a library where a valued network between words and components is created. So, when we search a word or a component in this library, a weight is calculated for each component with the nodes' values, and the selected candidate is the one which has the biggest weight. Our approach is at a different level, because we search and select candidates, not from keywords, but from components' structure. It can be used in such retrieval mechanisms in order to refine component search, and create more trustable libraries.

For our quality generic model, we were inspired by quality standards like ISO-9126 [14] and metrics standards like IEEE-1061 [13]. Example of existing metrics that could be used with our model can be found in [11,21]. But the quality part of our model can also be used with quality of service contracts languages (based on Antoine Beugnard's fourth level of component contracts [2]), such as the ones modeled in QML [9] and QoSCL [8]. In particular, our concern about substituting non-functional properties can be compared to Jan Aagedal's CQML language [1], that deals with the substitutability of QoS "profiles". However, contrary to CQML, which, like most QoS languages, doesn't take functional aspects into account, our model combines functional and non-functional ones. And while Aagedal separates primitive component substitutability and composite component one, we deal with contextual substitutability of two components, no matter their internal structure.

7 Conclusion and future work

We proposed a substitution model including several elements: i) a generic quality model, able to use existing quality metrics and QoS languages. ii) a generic component model, able to use existing research and industrial approaches. iii) a

substitution discrepancy, able to measure the substitutability of a candidate component. We also introduced the notion of ideal component, that models functional and non-functional conceptual needs and takes composition context into account.

In our current framework, we chose to consider one component model using existing quality characteristics and metrics from one quality model. The reason for such a limitation is that in the actual research and industrial schemes, composition concerns mainly components that come from the same component model.

Right now, we have a tool [10] that allows us to check if a component can substitute another one according to our substitution discrepancy measurement. This tool aims to help designers to find the best candidates for their needs.

References

[1] Aagedal, J. O., "Quality of Service Support in Development of Distributed Systems, " Ph.D. thesis, University of Oslo, Norway, 2001.

[2] Beugnard, A., Jean-Marc Jézéquel, Noël Plouzeau and Damien Watkins, *Making Components Contract Aware*, IEEE Computer **32 (7)** (1999), 38–45.

[3] Blair, G., and J.-B. Stefani, "Open Distributed Processing and Multimedia", Addison-Wesley, 1997.

[4] Brada, P., "Specification-Based Component Substituability and Revision Identification, " Ph.D. thesis, Charles University, Prague, Czech Republic, 2003.

[5] Cardelli, L., *Type Systems*, in : "The Computer Science and Engineering Handbook", chapt. 97, CRC Press, 2004.

[6] Crnkovic, I., *Component-based Software Engineering for Embedded Systems*, Proceedings of International Conference on Software engineering (ICSE), May 2005.

[7] Crnkovic, I., Stig Larsson and Michel Chaudron, *Component-based Development Process and Component Lifecycle*, Proceedings of the 27th International Conference Information Technology Interfaces (ITI), June 2005.

[8] Defour, O., Jean-Marc Jézéquel and Noël Plouzeau, *Extra-Functional contract support in components*, Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE7), May 2004.

[9] Frolund, S. and Jari Koistinen, "QML: A Language for Quality of Service Specification," technical report HPL-98-10, Hewlett-Packard Laboratories, California, USA, 1998.

[10] George, B., and Thomas Boggini, "The Substitute Tool", 2006, URL: http://www-valoria.univ-ubs.fr/SE/Substitute/

[11] Goulao, M., and Fernando Brito e Abreu, *Software Components Evaluation: an Overview*, Proceedings of the 5th Conferencia da Associaçao Portuguesa de Sistemas de Informaçao (CAPSI), November 2004.

[12] Henninger, S., *An Evolutionary Approach to Constructing Effective Software Reuse Repositories*, ACM Transactions On Software Engineering and Methodology (TOSEM) **6 (2)** (1997), 111–140.

[13] "IEEE Std. 1061-1998:IEEE Standard for a Software Quality Metrics Methodology", IEEE Computer Society Press, 1998.

[14] "ISO/IEC 9126-1:2001 Software Engineering - Product Quality - Part I : Quality model", ISO International Standards Organisation, 2001.

[15] Liskov, B., and Jeannette Wing, *A Behavioral Notion of Subtyping*, ACM Transactions On Programming Languages and Systems (TOPLAS) **16 (6)** (1994), 1811–1841.

[16] Lucrédio, D., A. F. D. Prado and E. S. D. Almeida, *A Survey on Software Components Search and Retrieval*, Proceedings of the 30th EUROMICRO Conference, September 2004.

[17] Perry, D., and Alexander Wolf, *Foundations for the Study of Software Architecture*, ACM SIGSOFT Software Engineering Notes **17 (4)** (1992), 40–52.

[18] Seco, J. C., and Luis Caires, *A Basic Model of Typed Components*, Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP), June 2000.

[19] Szyperski, C., "Component Software : Beyond Object-Oriented Programming," 2nd Ed., Addison-Wesley/ACM Press, 2002.

[20] Van Ommering, R., *Software Reuse in Product Populations*, IEEE Transactions on Software Engineering **31 (7)** (2005), 537–550.

[21] Washizaki, H., Hirokazu Yamamoto and Yoshiaki Fukazawa, *A Metrics Suite for Measuring Reusability of Software Components*, Proceedings of the 9th International Symposium on Software Metrics (METRICS'03), September 2003.

[22] Zaremski, A. M., and Jeannette Wing, *Signature Matching : A Tool for Using Software Libraries*, ACM Transactions On Software Engineering and Methodology (TOSEM) **4 (2)** (1995), 146–170.

[23] Zaremski, A. M., and Jeannette Wing, *Specification Matching for Software Components*, ACM Transactions On Software Engineering and Methodology (TOSEM) **6 (4)** (1997), 333–369.