A Component-Oriented Substitution Model

Bart George, Régis Fleurquin, Salah Sadou

A Component-Oriented Substitution Model

Bart George, Régis Fleurquin, and Salah Sadou

VALORIA Lab., University of South Brittany, France
{Bart.George,Regis.Fleurquin,Salah.Sadou}@univ-ubs.fr

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 its functional and non-functional properties. Then, the problem is to know which component satisfies a specific need in a specific composition context, during software conception or maintenance. We state that this is a substitution problem in any of the two cases. From this statement, we propose a need-aware substitution model that takes into account functional and non-functional properties.

1 Introduction

Component-oriented programming should allow us to build a software like a puzzle whose parts would be units "subjects to composition by a third party" [17]. Examples of such units are COTS (*Components-Off-The-Shelf*), which are commercial products from several constructors and origins. When one develops and maintains a component-based software, some problems occur, and we will notice two main ones: how to select, during conception of such a software, the most suitable component in order to satisfy an identified need ? And during a maintenance, if this need evolves, will the chosen component remain suitable, or shall we replace it ?

We think that these problems are related to a substitution problem. In fact, when one conceives or maintains an application, some 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 make concrete components substitute ideal ones.

However, composition doesn't concern only the functional aspect. Most components are "black boxes" which must describe not only functional, but also non-functional properties. As every software needs a certain quality, one can't think about composing components whose non-functional properties are unknown, and at the same time hope having its quality requirements satisfied anyway. 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 subtyping, as some object-oriented languages made it a general way of substitution. However, an ideal component describes more than general needs: it describes the application's

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, trying to substitute the old candidate by a new one would be a mistake, because the key notion isn't the candidate, but the need it is supposed to satisfy. Plus, if this need changes, a former candidate may no longer remain suitable. So substitution of an ideal component by a concrete one is performed only into the context of the need modeled by the ideal component. This is why a candidate component can replace another one without any subtyping relation between them, as every candidate is compared only to the ideal component.

In this paper, we consider a generic component model and a quality model (section 2), and into this framework we define a component-oriented substitution model, including substitutability rules for every functional and non-functional element of our model (section 3). In order to illustrate the possibilities of such a model, we describe the different substitution cases during the life cycle using a short application example (section 4). Then, before concluding, we describe some related works (section 5).

2 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 [12]. In this framework, we suppose the existence of metrics to measure non-functional properties (such as those defined in [19]), so that our contribution will focus only on the substitution model definition.

2.1 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.

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 of the paper, we refer to **candidate component** and **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.

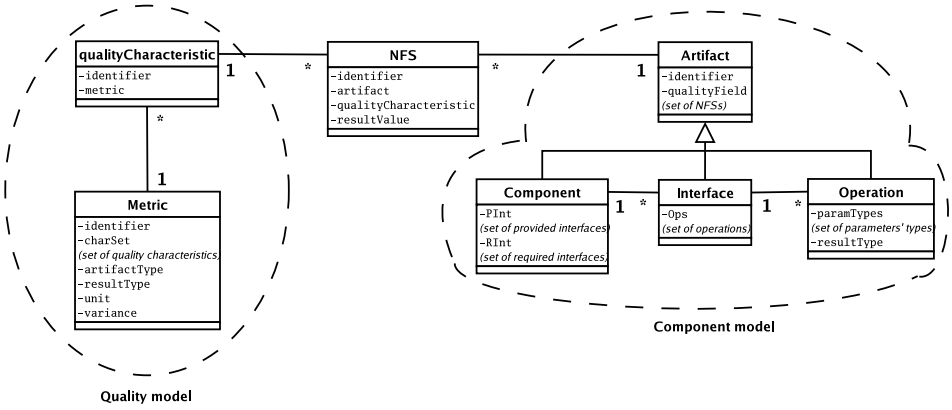

Fig. 1. Our generic model

Beside the component model, we define a generic quality model. Its elements are quality characteristics (such as those from ISO 9126 [12]), and metrics. We use existing metrics to evaluate and compare non-functional properties (see [9] 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.

2.2 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 of this paper, we consider that a metric may measure several quality characteristics (as proposed in the IEEE standard 1061-1998 [11]), but each characteristic is measured by only one metric. Elements of the quality model are defined as follows:

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

Metrics: A metric holds a set of quality characteristics it measures. It also holds 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 they are from the same metric. 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.

2.3 Non-functional specifications

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

In Figure 1, the *resultValue* attribute of an NFS 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.

Two NFSs are comparable if the artifacts they belong to are of the same kind and comparable (see next subsection for comparison definitions), and if they refer to the same characteristic. Two NFSs are equal if they are comparable and their *resultValue* attributes are equal.

2.4 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 NFSs. Two artifacts' quality fields are comparable if, for each NFS of one quality field, there is at least one comparable NFS in the other quality field. Two quality fields are equal if for at least one NFS of one quality field, there is an equal NFS in the other quality field, and *vice versa*.

Let us now describe the different kinds of artifacts:

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 (α_1, ... , α_n)[1] and its result's type β. It is noted (α_1, ... , α_n) $\longrightarrow \beta$.

Two operations are comparable if their signatures are comparable. Two operation signatures T and U are comparable if they are equal modulo the renaming of the type names, or if there exists a type substitution relationship V so that $V.T$ equals to U, or T equals to $V\ U$, modulo the renaming of the type names.

For example, $\alpha \longrightarrow \alpha$ equals to $\beta \longrightarrow \beta$ if we rename α by β, but $\alpha \longrightarrow \alpha$ is not equal to $\beta \longrightarrow \gamma$.

And 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 [20].

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

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*.

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 pretend to substitute C_0.

3 Our substitution model

For each NFS, we attach a **weight** (or comparison weight) noted $Comparison_S$, and a **penalty** noted $Penalty_S$ (S being the NFS). These two values define the NFS's importance for the artifact it belongs to. The higher these two values are, the more important this NFS is, in the whole substitutable component. If a substitutable artifact owns an NFS 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

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

candidate component may also bring his own new NFSs that the substitutable component doesn't have. These new elements will be evaluated by the ideal component designer.

The **substitution distance**, or distance, is defined using these weights, penalties, and NFS' *resultValue*s. This distance will inform on the substitutability of an NFS or an artifact. The best candidate for substitution is the one with the lowest distance. If the distance 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 distance is positive, then the candidate is worse. If the distance equals to 0, then the two compared elements are "equivalent" each to the other, but it doesn't mean that they are equal.

For each component, there is a **maximal distance** 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 distance between C_1 and C_0 is bigger than the maximal distance associated to C_0, then C_1 will be rejected.

3.1 Substitution distance between artifacts

Here, we will define a calculus that will give the distance separating a candidate component C_1 from a substitutable component C_0 in a given context. This context is defined by the weight and the penalty allocated to the NFSs of C_0's artifacts. So, before talking about distance between artifacts, let us present the distance between their quality fields.

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

Distance 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 distance between these quality fields (denoted QD) is defined as follows:

$$QD(Q_{A_1}, Q_{A_0}) = \sum_{S_0 \in Q_{A_0}} QDSpec(Q_{A_1}, S_0) - \sum_{S_1 \in Q_{A_1}} QDBonus(S_1, Q_{A_0})$$

with:

$$QDSpec(Q_{A_1}, S_0) = Comparison_{S_0} * (resultValue_{S_0} -_{Variance} resultValue_{S_1})$$
if $\exists\ S_1\ in\ A_1$ that is comparable to S_0; else, $Penalty_{S_0}$.

and:

$QFBonus(S_1, Q_{A_0}) =$ 0 if $\exists\ S_0 \in Q_{A_0}$ that is comparable to S_1; else, a value given by C_0's designer.

To measure the distance between the quality fields, we try to find for each S_0 a comparable NFS S_1 in A_1 (there can be only one, as NFSs of a same artifact cannot share the same characteristic). Substituable NFSs without any

comparable S_1 are taken into account through their penalty value $Penalty_{S_0}$. Candidate NFSs without any comparable S_0 are taken into account through a value given by C_0's designer.

$resultValue_{S_0} -_{Variance} resultValue_{S_1}$ is a subtraction between $resultValue_{S_0}$ and $resultValue_{S_1}$ depending on their metric's variance. For example, if its type is integer or float and variance is increasing, the measurement will equal to: $resultValue_{S_0}$ - $resultValue_{S_1}$. If variance is decreasing, it will equal to: $resultValue_{S_1}$ - $resultValue_{S_0}$.

Distance between incomparable artifacts. If two artifacts are incomparable, there will not be any substitution distance measurement between them.

Distance between comparable operations. Let us consider a substitutable operation O_0 and a comparable candidate operation O_1. The substitution distance 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 distance between them is in fact the distance between their quality fields.

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

Distance 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 distance 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 distance between required interfaces is the same as for provided ones, except that it is symmetrical. 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.

Distance 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 distance 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 distance between the components, we take into account only the lowest found distance 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.

4 Substitution in practice

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. This video camera example is taken from [3].

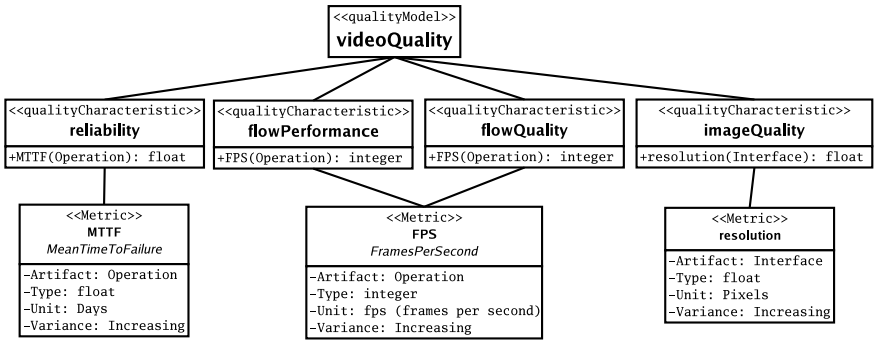

Fig. 2. Example of quality model.

4.1 Modeling an ideal component

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*[2]), and a required interface $DVFormat$ (with an operation *inputDVFlow* that asks for 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 NFSs. To each operation of the *cameraControl* interface, we attach an NFS using *reliability* characteristic (*onReliability* for *on* operation, *recordReliability* for *record* operation, and *ejectReliability* for *eject* operation). To *videoStream* interface, we attach the NFS *cameraResolution*, using the characteristic *imageQuality*.

Finally, the designer fixes expected $resultValue$s, weights and penalties for each NFS, and also fixes a maximal distance for the ideal component *videoCamera*. On Figure 3, we see that the expected value for *cameraResolution* is 1 million pixels, and the expected values for NFSs 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 *cameraResolution*, *recordReliability* and *ejectReliability* are very high in order to enforce candidate components to contain these NFSs. *cameraResolution*

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

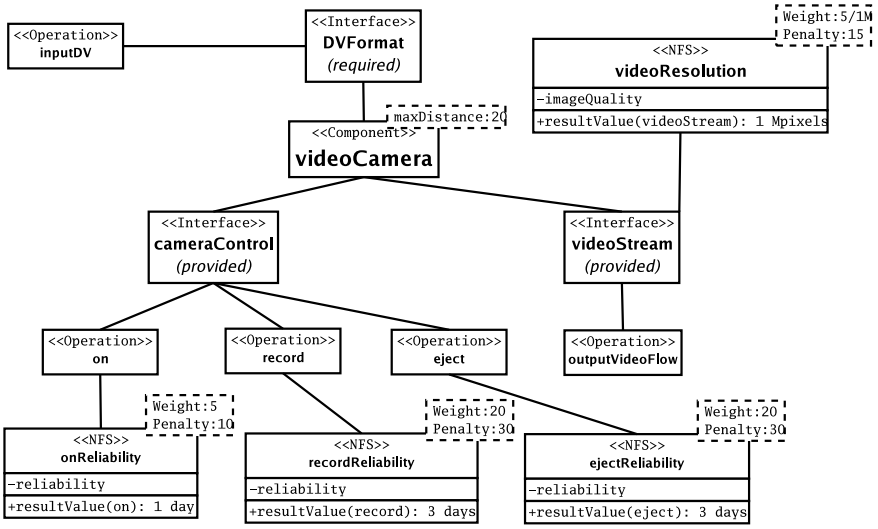

Fig. 3. Example of ideal component: *videoCamera*.

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 distance is fixed at a low level, so that the lack of one of these three NFSs in a candidate component will hardly be accepted.

4.2 Component lifecycle and substitution cases

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

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 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 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.

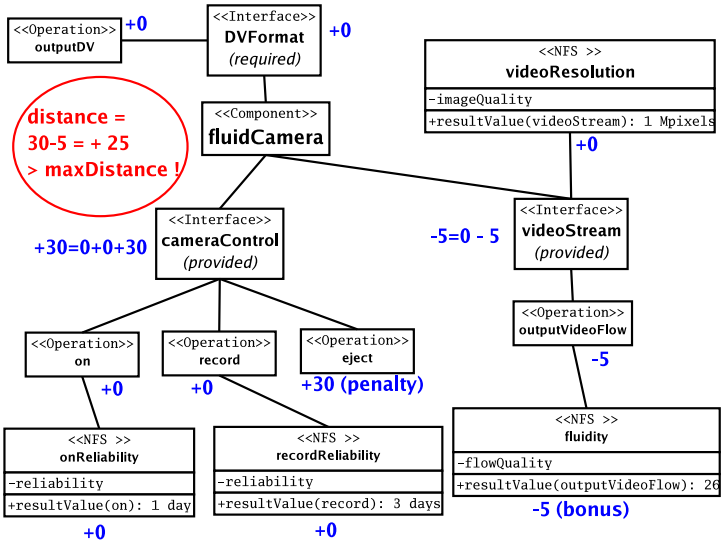

Fig. 4. Example of rejected candidate: $fluidCamera$.

Then, a candidate, like the $fluidCamera$ component on Figure 4, may add new NFSs 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 NFS using $flowQuality$ characteristic on the $outputVideoFlow$ operation, but the candidate ($fluidCamera$) lacks an important NFS. 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 NFSs. For example, let us take a candidate $goodImageCamera$ 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 5 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. If we consider this ideal component, and the distance obtained for each one of the candidates, we can say that both are acceptable (distance with candidate $goodImageCamera$ would equal to +15), but the $reliableCamera$ is the best one.

Maintenance. The application now has its camera component, but it could have a "better" one. If the needs are the same, the ideal component that models

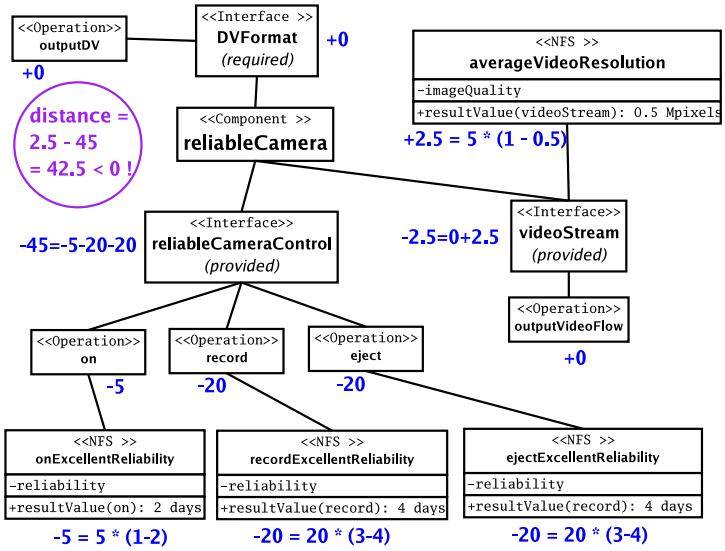

Fig. 5. Example of accepted : *reliableCamera*.

them 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. If the needs change, this implies that the ideal component changes too. Thus, we must compare each candidate (including previous accepted one) with the new ideal component. In both cases, we are back to the first composition schema.

5 Related work

We said in introduction that substitutability was a well-known problem in object-oriented languages which include typing [5] and subtyping [13]. It is also an industrial problem, as referred in [18], who asks how to make sure that changes on a component won't affect existing applications of a component, and try to answer by setting rules based on subtyping. It was tempting for us to base on subtyping too, in order to substitute components [16]. But we took critics of typing [15] and subtyping [17] 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 has 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 [20, 21]. 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 [14]. For example, our notion of weights can be compared to Scott Henninger's tools [10]. 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 [12] and metrics standards like IEEE-1061 [11]. Example of existing metrics that could be used with our model can be found in [9, 19]. 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 [7] and QoSCL [6]. 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.

6 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 distance, able 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. There are two reasons for such a limitation : i) in the actual research and industrial

schemes, composition concerns mainly components that come from a same component model; ii) the problem of comparing components from different models is orthogonal to the substitution problem. Both can be treated separately.

Right now, we have a tool [8] that allows us to check if a component can substitute another one according to our substitution distance measurement. This tool aims to help designers to find the best candidates for their needs.

References

1. J. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32 (7), 1999.
3. G. Blair and J.-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.
4. P. Brada. *Specification-Based Component Substituability and Revision Identification*. PhD thesis, Charles University in Pragues, 2003.
5. L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.
6. O. Defour, J.-M. Jézéquel, and N. Plouzeau. Extra-functional contract support in components. In *Proceedings of 7th International Symposium on Component-Based Software Engineering (CBSE 7)*, May 2004.
7. S. Frolund and J. Koistinen. Qml : A language for quality of service specification. Technical report, Hewlett-Packard Laboratories, Palo Alto, California, USA, 1998.
8. B. George. Substitute tool. http://www-valoria.univ-ubs.fr/SE/Substitute/, 2006.
9. M. Goulao and F. B. e Abreu. Software components evaluation : an overview. In *CAPSI 2004*, November 2004.
10. S. Henninger. Constructing effective software reuse repositories. In *ACM TOSEM 1997*, 1997.
11. IEEE. *IEEE Std. 1061-1998 : IEEE Standard for a Software Quality Metrics Methodology*, ieee computer society press edition, 1998.
12. ISO Int. Standards Organisation, Geneva, Switzerland. *ISO/IEC 9126-1:2001 Software Engineering - Product Quality - Part I : Quality model*, 2001.
13. B. Liskov and J. Wing. A behavioral notion of subtyping. In *ACM Transactions on Programming Languages and Systems 1994*, 1994.
14. D. Lucrédio, A. Prado, and E. S. D. Almeida. A survey on software components search and retrieval. In *EUROMICRO*, 2004.
15. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17 (4):40–52, October 1992.
16. J. C. Seco and L. Caires. A basic model for typed components. In *ECOOP*, 2000.
17. C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second edition, 2002.
18. R. Van Ommering. Software reuse in product populations. *IEEE Transactions on Software Engineering*, 31 (7):537–550, july 2005.
19. H. Washizaki, H. Yamamoto, and Y. Fukazawa. A metrics suite for measuring reusability of software components. In *Metrics 2003*, 2003.
20. A. Zaremski and J. Wing. Signature matching : a tool for using software libraries. In *ACM TOSEM 1995*, 1995.
21. A. M. Zaremski and J. Wing. Specification matching of software components. In *ACM TOSEM 1997*, 1997.