A Formal Framework for Analyzing Reusability Complexity in Component-Based Systems

Ismael Rodríguez, Manuel Núñez, Fernando Rubio

Dept. Sistemas Informáticos y Programación
Universidad Complutense de Madrid, E-28040 Madrid. Spain.
e-mail:{isrodrig,mn,fernando}@sip.ucm.es

Abstract

In this paper we present a methodology to estimate the impact of modifying a given software system design. In addition, we will be able to evaluate its reusability as well as the coupling of its components. In order to do that, the designer defines the system in terms of its components, their dependencies, the properties they fulfill, and the properties each component requires to other components. Besides, some auxiliary functions are used to define the relations between properties and the cost associated with their modification. Putting together all this information the different ways to perform a modification can be systematically generated and studied. We have applied our methodology to a medium-size system. Specifically, we dealt with an on-line intelligent tutoring system allowing users to learn the programming language Haskell.

1 Introduction

In the last years there has been a great interest in the development and implementation of tools, as well as in the definition of well-founded methodologies to facilitate the task of system designers. Clear examples of this current are the work around UML [18] and the application of design patterns [6]. In this line, Component-Based Software Engineering, in short CBSE, is a very active field of research and practice (see e.g. [1,3,5] for a description of issues in CBSE). Given the fact that this is a relatively new area of study, and that concepts are taken from other fields as object-oriented programming, there is not yet a standard interpretation for the terminology (in [21,4] two interesting presentations of the involved concepts are provided). Actually, there is not even a standard notion of component, and the concept is usually misunderstood. In terms of [20], the characteristic properties of a component are that it is a unit

of *independent deployment*, a unit of *third party composition*, and it has *no persistent state*.

Taking into account that the field of CBSE is still in a premature phase, it is not a surprise that one of the main concerns consists in the study of the properties related to component composition (e.g. [13,14,19,7]). In this paper we develop a cost model related to component composition when components are reused either to create new designs or to modify existing ones. It is worth to point out that a very relevant information about a concrete design is the *tightness* of the coupling among different components. In other words, how the modification of an existing component affects other components. Actually, if a software system has to be modified/updated then it is very important to analyze the different costs associated with each possible way to develop the new system. This is not a trivial task since modifications in one of the components may induce the propagation of this change to other components. This process may be repeated until a new *coherent* design is achieved. Thus, it would be very desirable to automatically perform an analysis of the *impact* of the modification of each component. Unfortunately, such an analysis cannot be completely automatized due to the complex relations that may exist among different components. For instance, a straightforward analysis of the source code of an application can alert us that the modification of a given class *may* affect all the classes that depend on this class. Nevertheless, this analysis will be unable to detect which of them will *actually* be affected, as this finer analysis depends on the semantics of each class. However, in this paper we show that by considering a suitable abstraction of the design, the impact of the propagation of such a modification can be studied in a systematic and semi-automatic way. Our model is based on the following points. First, we consider, as usual, that the complete design includes a directed graph specifying dependencies among components. Besides, following the division described in [17], each component contains the following information:

- The set of properties fulfilled by the component.
- The set of properties that the component expects from the components related to it.
- A function computing the cost of modifying the component according to some future configuration. This function takes into account the properties specified in the previous items.

It is important to take into account that, if a component must be modified, there are (in general) several possibilities to reach a new coherent design. Nevertheless, they may have different associated costs. For instance, we may modify a component in such a way that the requirements of the neighbor components are kept. In this case, the propagation of changes is contained. Another possibility is to modify the requirements so that the change is propagated to other components. If we only consider the cost associated with the

modified component then the first possibility will be, in general, more expensive. However, let us note that the cost associated with the rest of components will be reduced. Thus, it is the goal of our model to decide which solution is *globally* less expensive. Besides, our methodology chooses a solution such that each component is modified at most once. Otherwise we could find loops in the graph so that we would never reach a coherent design. Finally, it may happen that a change in a component induces a modification in the dependencies graph. In this case, the new graph will be considered for the remaining changes.

In addition to the cost associated with the induced modifications, we will consider other factors. For example, it may be interesting to take into account whether the new design will be itself easy to modify/upgrade. Actually, there are a lot of situations in which the possible future modifications of a given design can be anticipated in early stages, so that they can be taken into account. For example, an incremental development methodology based on the regular creation of prototypes/versions may take advantage of our model so that forthcoming versions will be easier to create. Our model can be used to evaluate this issue with respect to a set of possible (desirable) future modifications. The method consists in computing the cost of these modifications for each of the possible designs. Let us remark that an approach like ours, that takes into account not only the current cost of modifications but also the possible future ones, will be helpful to choose between easy-to-develop modifications in the short term or in the long term.

Moreover, a useful measure for the reusability of a design (or its parts) is given by the degree of coupling among its components. Actually, a measure of coupling can be used to determine how components are affected by the modification of one of them. If we have a low degree of coupling then the current design will be easier to modify. In addition, a component with a low degree of coupling can be more easily reused in a completely different system. According to our model, we may evaluate the degree of coupling with respect to a set of hypothetic future modifications. This fact helps to provide more *rational* designs. Indeed, designs can be split into those groups of components presenting a higher degree of coupling, so that *higher order* components can be considered.

After presenting the main issues and goals, we briefly sketch the methodology proposed in this paper. First, the studied system is split into components and their dependencies are set so that the dependencies graph is generated. Let us remark that components are identified as deployment units which must be atomically used in the reusing process. Second, the relevant properties of the components (from a reusing point of view) are determined. We have to identify both the own properties of the components and the properties each component demands from the components they depend on. Next, we define

the objectives of the desired modifications in terms of new properties to be fulfilled by the components. The *correct* modifications will be identified, that is, those modifications producing a new system where both the objective of the modification and the internal coherence among components are fulfilled. Let us note that in the modification process we have to consider factors as the possibility of reusing old versions of components or purchasing binary commercial components that cannot be modified. Once these correct modifications are found and their costs are computed, costs of possible future modifications will also be computed in order to measure the *reusability* level of the resulting system. In short, we advocate that the relation among components must be modeled so that validity of components is considered as a factor depending on the other components configuring the whole system.

In conclusion, and using terminology introduced in [20], the main goal of our approach consists in migrating from *introvert* component structure models, that is, components are thought as *isolated* units, to *extrovert* component structures, where also the context in which components are used is taken into account. Only in the second case it is feasible to develop software as a *plug-and-play* methodology. The methodology that we present in this paper has been already used in the design of the `WHAT` system [12]. So, it can be properly said that our work represents a real example of theory guided by practice. `WHAT` is a tutoring system to help students who are learning Haskell. It is worth to point out that components where implemented by different programmers (from students to ourselves). In particular, the different teams were responsible for providing the adequate properties and cost functions associated with the components. Moreover, some of the components were created much before we even had in mind this methodology (e.g. we were *reusing* some modules from [15,16]). Thus, even though we did not purchase any component, these last set of components were considered as binary components since we could neither influence in their design nor modify them. Finally, two major modifications of the system were carried out. First, we had to evolve from the first running prototype into the first *real* version of WHAT. This step did not represent a big change in the high level conceptual design but involved most of the *former* components. Second, the system had to be extended in order to include a collaborating system for the team-development of Haskell programs [11].

Regarding related work, let us remark that most metrics proposed as cost models are based on the study of some specific characteristics of the source code of the components. This is the case of the lines of code, the function points, or the inheritance trees in object oriented frameworks [8,9,2]. Another proposal consists in performing black-box metrics so that it is not needed to study the source code [22]. In this framework, metrics are based on the external interfaces of the classes of the components. All of these analyses are based on the study of the *syntactic structure* of the components, that is, they do

not study what the components do but the structure of its code or the structure of the interfaces of the components with the external environment. On the contrary, our metric to estimate development costs uses abstractions that model the *semantical relation* existing among the different properties of the components. This abstraction is performed through the simple design functions that we will present in section 4, which are automatically composed to create a single global cost function. In fact, we claim that a successful analysis of modification costs or reusability requires that the semantical dependencies existing among the properties of components are both taken into account and represented in the cost model. Nevertheless, let us remark that our methodology profits from the ideas underlying the metrics commented above. The influence of a property on some other property can be measured by taking into account the syntactic factors used by the other metrics. For instance, the lines of code could be used to estimate the effort needed to include some new property in our design.

The rest of the paper is organized as follows. In Section 2 we present the system being the driving force of our methodology. In Section 3 we introduce our formalization of component and component-based system. Then, in Section 4 we give the definition of the functions computing the cost associated with a modification of a design. In Section 5 we describe how a system is modified step by step to reach a *coherent* design. Finally, in Section 6 we present our conclusions.

2 An Application of the Methodology

Before we introduce our formalism we consider more appropriate to present the system whose development inspired our methodology. WHAT [11,12] (Web-based Haskell Adaptive Tutor) is an on-line intelligent tutoring system whose main purpose is to teach Haskell. After implementing a first restricted version of this tutor, we considered extending it to deal with new capabilities. While extending its capabilities, we also decided to modify part of the technologies underlying the development of the system. The redesign of our application was done by using the methodology presented in this paper. Some of the concepts appearing in this section will be formally presented in the forthcoming Sections 3 and 4. However, we will give intuitive explanations of their meaning.

2.1 A Brief Introduction to WHAT

WHAT is an on-line system developed to help students learning the functional language Haskell. Its main duty consists in proposing exercises to stu-

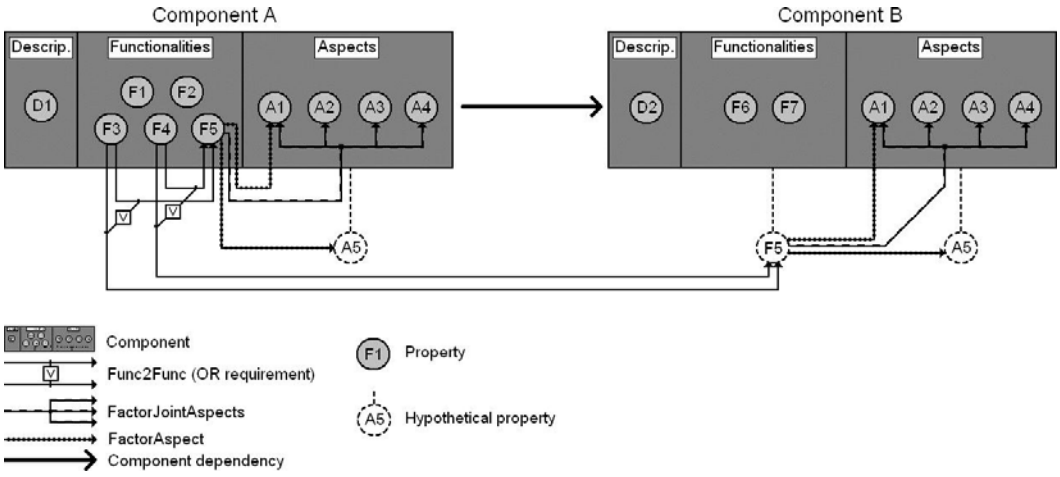

Fig. 1. Diagram of configuration of components

dents according to their current command on the language. WHAT provides a personalized monitoring of students by creating individualized profiles that allow to offer better assistance to them. In order to do that, the system takes into account both the information about the current student and also all the information about previous *similar* students. Students are similar when they belong to the same *classes*. WHAT manages both *static* and *dynamic* classes. The former considers attributes as whether the student is taking the course for the first time, whether an imperative language is already known, etc. Dynamic classes handle information that may vary along the academic year. For example, dynamic classes consider whether a student learns fast or whether he has good memory. In contrast with static classes, as dynamic attributes change (on the basis of interactions with the system) students can be automatically relocated into different dynamic classes.

2.2 Analyzing Modifications

Due to the big size of the complete model of components of WHAT, with lots of properties, conditions, etc., we will concentrate on showing a very small portion of it. Specifically, we will present only two components. The first one describes the logic of the management of student's data and the second one describes how the access to persistent data is provided. The properties shown in this example are just a small subset of the whole set of properties, so that we may focus on this concrete example. As we will explain in Section 3, a component has three kinds of properties associated with it: *Descriptions* (describing what the component is), *functionalities* (describing what the component does), and *aspects* (describing how the component performs its tasks).

Component A	Descriptions	D1) Manager of accesses to student's data.
	Functionalities	F1) Consult student's data. F2) Modify student's data. F3) Find all the (current year) failing students. F4) Find all the repeat students. F5) Range students fulfilling special condition.
	Aspects	A1) Persistent data in files. A2) Students data include name. A3) Students data include marks. A4) Students data include whether they are repeat students or not.
Component B	Descriptions	D2) Manager to access persistent data.
	Functionalities	F6) Read student's data. F7) Write student's data.
	Aspects	A1) Persistent data in files. A2) Students data include name. A3) Students data include marks. A4) Students data include whether they are repeat students or not.

Fig. 2. Initial configuration of the components

Let us consider the two components A and B depicted in Figures 1 and 2. The component A provides functionalities to search information about students, but it has to use B in order to access persistent data. In our first rudimentary prototype, we handled the information about students by using independent files for each student. However, when developing the current version of WHAT, we decided to use an Oracle database. As it can be expected, the decision of using a different method to access persistent data forced a modification in the design of the components.

In the rest of this section we present how we dealt with the previous modification. We will informally introduce some of the functions where relations between components as well as modification costs are encoded (formal definitions of these functions will be given in Section 4).

In the initial version, as data were stored in different files, ranging repeat students was tricky, as it forced handling the files directly, searching for students fulfilling the required conditions. Therefore, the functionalities F3 and F4 of

the component A forced F5 to be fulfilled. In fact, one of the requirements of A is that a functionality for ranging students must be available, no matter whether this functionality is implemented by A or delegated to B. This requirement is included in the function *Func2Func*, which for each functionality states the other related functionalities (both from the component and from the related components) that must be fulfilled. Of course, the costs of performing these searches were strongly influenced by the aspect saying that persistent data were stored in files. This fact is included in the definition of the function *FactorAspect*. This function returns, for each functionality, the cost associated with adapting the functionality to the corresponding aspect. Besides, as a lot of information about students was stored, ranging files was much more complicated than needed. As a result, while including students data in persistent files, *FactorJointAspects* penalized the costs associated with ranging students. This function returns, for a given functionality, the overhead generated because a set of aspects must be simultaneously fulfilled.

Given the previous configuration, we proposed to modify it so that persistent data were stored in an Oracle database. Such a modification would imply the replacement of the aspect A1 by a new aspect A5, meaning that persistent data are stored in a database. Obviously, looking for students with special characteristics would be easier with the new configuration, since SQL queries were available. In fact, we found two main methods to implement the modification, and we computed the cost functions for each of them, using programmer-hours as a unit to measure the costs. We used this cost unit instead of money since the modification was going to be performed by our students and us. Actually, there was not money available to buy external components. The first alternative consisted in keeping the low-level search mechanism in the component A while modifying B to substitute file management by database queries. The functionalities of B would change, and their costs would be estimated by using *FactorAspect* and *FactorJointAspects*. Nevertheless, this approach was easily shown to be inefficient, as database languages allow to perform this kind of queries easily. Therefore, a better way to implement the modification consisted in moving functionality F5 from component A to the component B. By doing so, not only the cost of the modification was smaller (in fact, it was nearly four times smaller) but it also made cheaper to develop possible future modifications consisting in adding more students data (such as *addresses* and *phone numbers*). The reason was that *FactorJointAspects* almost did not penalize the cost of modifying these functionalities in B, while in the first approach the cost of modifying these functionalities on A was dramatically bigger.

Finally, let us remark that the second alternative was also more *appropriate* in the sense that the coupling of the components was smaller. That is, future modifications in one of them would have less impact in the other one, as the access to persistent data was isolated in a cleaner way.

3 Basic Definitions

In this section we introduce some concepts that we will use along the paper. Specifically, we give our notion of component as well as some concepts related to the composition of them. First, we specify the different kinds of properties that components may hold. As usual, we have both *functional* and *extrafunctional* properties (see e.g. [4]). *Functionalities* are those properties defining *what* the component *does*. For example, "recording user data", "generating queries for the balance", "save persistent data", etc. We denote by F the set of functional properties. However, we will consider a finer division because we split extrafunctional properties into *descriptions* and *aspects*. By *descriptions* we consider those properties defining *what* the component *is*. Typical examples are: "a user interface", "a layer in a communication protocol", "a database query module", etc. We denote by D the set of description properties. *Aspects* define *how* the component performs its tasks. They approximately correspond to the classical definition of aspect in aspect-oriented programming [10] in the sense that they cross-cut basic functionalities. For example, "data from clients include name, ID, and balance", "the communications protocol is Ethernet", "persistent data are saved in a file", etc. We denote by A the set of aspect properties.

Definition 1 A *component* C is a tuple (P, P_o, τ), where $P, P_o \subseteq D \cup F \cup A$ are the set of *own properties* and of *properties required to neighbors*, respectively, and τ is the *group of design functions* for this component. We denote by ξ the set of all components. A *cost-evaluated component* E is a tuple $(P, P_o, ModifCost)$, where P and P_o are as before and

$$ModifCost : \mathcal{P}(P) \times \mathcal{P}(P_o) \times \mathcal{P}(P) \times \mathcal{P}(P_o) \to \mathbb{R}^+ \cup \{\infty\}$$

is the *function of modification cost*. We denote by ϱ the set of all cost-evaluated components. □

As we indicated in the introduction of this paper, the description of a component (P, P_o, τ) is given by the properties fulfilled by the component (i.e. P), the properties expected from the components related to it (i.e. P_o), and a tuple of design functions (i.e. τ). In the next section we formally present the kind of functions appearing in these tuples. They give some basic values over the different kinds of properties. A *cost-evaluated component* represents a different *view* of a component. Intuitively, the group of design functions is replaced by a *cost* function computing the cost associated with the modification of the component by taking into account the initial properties (first two arguments of *ModifCost*) and the properties to be fulfilled after the modification is performed (third and fourth arguments of *ModifCost*). In the next definition we introduce two notions related to systems of components.

Definition 2 A *simple system* of components S is a pair (K, G), where we have that $K \subseteq \varrho \times \mathbb{N}$ is a set of indexed cost-evaluated components, and $G : \mathbb{N} \to \mathbb{N}$ is the directed *dependencies graph* of the system. A *global system* of components GS is a tuple (S, Av, M), where S is a simple system, Av is a set of pairs (E, Mod) denoting the available cost-evaluated components $E \in \varrho$ while $Mod \in \{true, false\}$ is a boolean denoting whether E can be modified, and $M : \varrho \to \mathbb{R}^+ \cup \{\infty\}$ is a *market* denoting the price of purchasing binary components, measured in units of effort. □

In the definition of simple systems we consider that each component has a unique identification number. Moreover, for any $i \in \mathbb{N}$ we have that $j \in G(i)$, with $(E', i), (E, j) \in K$, means that E' depends on E. Let us remark that a simple system contains the main information that we require, as presented in the introduction of the paper, about a component based system (e.g. the dependencies graph). Global systems extend the notion of simple system to consider that components can be bought. Let us remark that these components cannot be modified because we will not (usually) receive the source code. Besides, global systems also include information regarding the components which are freely available to the designer. These components originate in either current and previous versions of used components or in previous purchases.

4 Generation of the Cost Functions

In this section we present the definition of the functions computing the cost of performing a modification in a component-based system. First, we give a set of simple functions, each of them dealing with different kinds of properties. Afterwards, we show how these simple functions are combined to compute the overall cost function.

4.1 Generating Design Functions

Given a component and its current properties, the *function of modification cost* computes the costs required to modify its current properties so that it fulfills some new desired properties. Obviously, the number of different combinations of current and desired properties that could be taken into account is huge. Thus, it is not feasible for a designer to define all these cases. In contrast, a method allowing the designer to specify a default pattern should be provided, but the definition of as many special cases as needed must be allowed. Following these ideas, in our framework, the designer will only need to declare some rules for each property (based on heuristics). For example, we consider that a change into a configuration where the properties of the

component to be modified and those of the related components are similar will be less expensive than a change into a configuration where these properties diverge. In addition, the relations among the different properties must be defined. After that, an automatic process will compute the appropriate costs description function.

The key point to make feasible the previous approach consists in finding an adequate set of basic rules. On the one hand, this set should be simple enough to allow the developer to easily use the rules. On the other hand, it should be expressive enough to provide the user with a useful tool. The rest of this section presents a set of basic definitions that will represent such a compromise between expressivity and feasibility. We will classify them into two main categories: those related to the modification of the properties of the component itself and those related to the modification of the properties that the component requires from other components.

Let us start by describing the information required to express the modification of the properties of the component. We consider three kinds of properties depending on what must be modified: Descriptions D, functionalities F, and aspects A.

When considering descriptions we need to take into account three points. First of all, sometimes different descriptions will share some common functionalities. In these cases we can reduce the costs associated with modifying a description by taking advantage of the *similarities* of the functionalities. Thus, we are interested in the ratio of the costs that we will *save* due to similarities. We define the function

$$Similarity : D \times D \rightarrow [0, 1]$$

where $Similarity(d_1, d_2)$ returns the proportion of the functionality of d_1 which can be applied to d_2. The default value is 0, while a value 1 means that d_1 is a generalization of the description d_2. As a second cost, depending on the complexity of a description, there will be a fix *base cost* to develop any concrete functionality. We define this cost

$$DescripBaseCost : D \rightarrow \mathbb{R}^+$$

where $DescripBaseCost(d)$ denotes the base cost associated with the application of a functionality to the description d. The default value is 0. As a final issue related to descriptions, the cost to develop a description is multiplied by a certain factor when it is being developed together with other descriptions. Thus, we consider a function

$$Multiplicity : \mathcal{P}(D) \rightarrow \mathbb{R}^+$$

where 1 is the value by default.

In the case of functionalities we also have to consider several issues. First, and analogously to the previous case, each functionality has a base cost. Thus, we have the function

$$BaseCost : F \to \mathbb{R}^+$$

where $BaseCost(f)$ denotes the cost needed to perform functionality f without any other consideration, being 0 the default value. Second, we can reduce the costs of implementing a functionality by using similarities with other functionalities already implemented. This fact is considered by the function

$$ReusingSavings : F \times F \to [0, 1]$$

where $ReusingSavings(f_1, f_2)$ defines the ratio of the functionality f_2 that can be saved by using functionality f_1. In this case, the default value is 0. Alternatively, the value 1 indicates again that f_1 is a generalization of the functionality f_2. Third, a functionality can modify its cost by applying a given aspect to it, multiplying its base cost by a certain factor. This is reflected in the function

$$FactorAspect : F \times A \to \mathbb{R}^+$$

where the default value is 0. This value means that the aspect a has no influence on the functionality f when applying $FactorAspect(f, a)$. Finally, the occurrence of several aspects while developing a functionality can also introduce a multiplicative factor in its cost. We consider the function

$$FactorJointAspects : F \times \mathcal{P}(A) \to \mathbb{R}^+$$

where 1 is the default value and represents that there is no influence.

Let us consider now the influence of *aspects*. In this case, we will have to consider three issues. First, the base cost factor due to applying an aspect is given by

$$AspectBaseCost : A \to \mathbb{R}^+$$

being 0 the default value. Second, for each functionality, the cost needed to migrate from an aspect to another is given by the function

$$Migration : A \times A \to [0, 1]$$

where $Migration(a_1, a_2)$ defines the cost saved when migrating from aspect a_1 to aspect a_2, or when adding a_2 if a_1 already exists. Also in this case we have that a value 1 means that a_1 is a generalization of aspect a_2, while 0 is the default value. Finally, when including several aspects simultaneously, a multiplicative factor must be considered, representing the compatibility of such aspects:

$$Compatibility : \mathcal{P}(A) \to \mathbb{R}^+$$

being 1 the default value.

Next we present the functions related to the modification of the properties required to neighbor components. In addition to the own properties of the component (represented by D, F, and A) we will consider three more categories denoting the properties the component requires to the components it is related with. These categories are D_o, F_o, and A_o, representing descriptions, functionalities, and aspects respectively. In order to formalize the conditions that must be fulfilled, we will use formulas of propositional logic in which the proposition symbols of the signature represent both own and required properties. We will denote by $L(\Sigma)$ the set of propositional formulas which can be built from the signature Σ. Besides, we will assume that a valuation $\nu : \Sigma \rightarrow \{true, false\}$ is available such that $\nu(p)$ evaluates to *true* iff the property p holds in the component. Let us note that p can be either an own property of the component or a property the component requires to its neighbors. We denote by $\varpi(\Sigma)$ the set of all possible valuations over Σ. Besides, we consider an evaluation function

$$Eval : L(\Sigma) \times \varpi(\Sigma) \rightarrow \{true, false\}$$

such that $Eval(\varphi, \nu)$ represents the evaluation of the propositional formula φ with valuation ν.

Each basic function related to requirements of neighbor properties provides a propositional formula which must hold in a *coherent* design. With respect to descriptions, two issues must be considered. The first one is related to the implications that a description of a component has on other descriptions. We have the function

$$Desc2Desc : D \rightarrow L(D \cup D_o)$$

where $Desc2Desc(d)$ denotes the conditions (both internal and external) that must be fulfilled when the description d holds. The default value is the formula *true*, denoting that nothing is required. The second issue is related to the implications that a description of the component have on the functionalities, both of the own component and of those of the neighbors:

$$Desc2Func : D \rightarrow L(F \cup F_o)$$

In the case of functionalities and aspects, the functions needed are analogous. Implications of a functionality on other functionalities, of the own component and of those of the neighbors, are encoded in the function

$$Func2Func : F \rightarrow L(F \cup F_o)$$

Besides, the implications of a functionality on the aspects (both internal and external) are given by

$$Func2Asp : F \rightarrow L(A \cup A_o)$$

$$
\begin{aligned}
CondHolds = \bigwedge & \{Eval(Desc2Desc(d), d_2 \cup d_{o2}) \wedge \\
& \quad Eval(Desc2Func(d), f_2 \cup f_{o2}) \mid d \in d_2\} \wedge \\
\bigwedge & \{Eval(Func2Func(f), f_2 \cup f_{o2}) \wedge \\
& \quad Eval(Desc2Asp(f), a_2 \cup a_{o2}) \mid f \in f_2\} \wedge \\
\bigwedge & \{Eval(Asp2Asp(a), a_2 \cup a_{o2}) \mid a \in a_2\}
\end{aligned}
$$

where $d_i = \{p \mid p \in \alpha_i \cap D\}$, $f_i = \{p \mid p \in \alpha_i \cap F\}$, and $a_i = \{p \mid p \in \alpha_i \cap A\}$

$d_{oi} = \{p \mid p \in \gamma_i \cap D_o\}$, $f_{oi} = \{p \mid p \in \gamma_i \cap F_o\}$, and $a_{oi} = \{p \mid p \in \gamma_i \cap A_o\}$

Fig. 3. Conditions to be satisfied in order to perform a modification

Finally, the implications of an aspect on other aspects of both the own component and of those of the neighbors appear in the function

$$Desc2Func : A \rightarrow L(A \cup A_o)$$

4.2 Generation of the Global Cost Function

Once we have defined the basic functions to compute costs we automatically obtain the global cost function. Let us remark that the user will not need to define complex functions like the ones that will be shown in this section: It will be enough to define the simple functions presented in the previous section and the whole cost function will be automatically obtained.

Let τ denote a group of design functions, where each of them is denoted by its sort (e.g. *Similarity* denotes the Similarity function). The cost function generated from τ, denoted by $Generated(\tau)$, is a function

$$ModifCost : \mathcal{P}(P) \times \mathcal{P}(P_o) \times \mathcal{P}(P) \times \mathcal{P}(P_o) \rightarrow \mathbb{R}^+ \cup \{\infty\}$$

where $P = D \cup F \cup A$ and $P_o = D_o \cup F_o \cup A_o$. Intuitively, $ModifCost(\alpha_1, \gamma_1, \alpha_2, \gamma_2)$ denotes the cost of modifying a component having certain own and demanded properties (α_1 and γ_1 respectively), to reach the future own and required properties (α_2 and γ_2 respectively). In this case, a value equal to ∞ denotes that the modification is impossible. The function *ModifCost* is defined as follows:

$$
ModifCost(\alpha_1, \gamma_1, \alpha_2, \gamma_2) = \begin{cases} Old + New & \texttt{if } CondHolds = true \\ \infty & \texttt{if } CondHolds = false \end{cases}
$$

$$
\begin{aligned}
Old &= \sum_{f \in f_1 \cap f_2} ModifAspectOld(f) + ModifDescripOld(f) \\
ModifAspectOld(f) &= COM \cdot FJA(f) \cdot \\
&\quad \sum_{a \in a_2 \setminus a_1} AspectBaseCost(a) \cdot CM(a) \cdot FactorAspect(f, a) \\
CM(a) &= min\{1 - Migration(a', a) \mid a' \in a_1\} \\
COM &= \prod_{\beta \in \mathcal{P}(a_2) \setminus \mathcal{P}(a_1)} Compatibility(\beta) \\
FJA(f) &= \prod_{\beta \in \mathcal{P}(a_2) \setminus \mathcal{P}(a_1)} FactorJointAspects(f, \beta) \\
ModifDescripOld(f) &= MUL \cdot \sum_{d \in d_2 \setminus d_1} DescripBaseCost(d) \cdot CS(d) \\
CS(d) &= min\{1 - Similarity(d', d) \mid d' \in d_1\} \\
MUL &= \prod_{\beta \in \mathcal{P}(d_2) \setminus \mathcal{P}(d_1)} Multiplicity(\beta)
\end{aligned}
$$

where $d_i = \{p \mid p \in \gamma_i \cap D\}$, $f_i = \{p \mid p \in \gamma_i \cap F\}$, and $a_i = \{p \mid p \in \gamma_i \cap A\}$

Fig. 4. Adaptation of old functionalities

In the previous expression, *Old* denotes the cost due to the adaptation of old functionalities, *New* is the cost due to the creation of new functionalities, and *CondHolds* indicates that all of the condition functions hold. These values are formally defined in Figures 3, 4 and 5 respectively.

Next we give an intuitive explanation of the meaning of the definitions appearing in these figures. In the definition of $CondHolds$ in Figure 3 the terms d_2, f_2, and a_2 denote the set of own properties after the modification, and d_{o2}, f_{o2}, and a_{o2} denote the sets of properties demanded to neighbors after the modification. Thus, $CondHolds$ requires that the specific conditions of any description, functionality, and aspect existing in the new component hold after the modification is performed.

In Figure 4 we show the costs needed to adapt old functionalities. These costs are split into those due to modifications in the aspects and those incurred due to modifications in the descriptions. When aspects change, the main cost for adapting the functionality is computed by adding all the *AspectBaseCost*'s, that is, by adding the base costs of all of the new aspects. However, this cost must be adjusted by taking into account both *migration* costs and *compatibility* of the aspects. These factors have to be adapted to the concrete functionality by using the functions *FactorJointAspects* and *FactorAspect*. Analogously to the case of modifying aspects, the costs due to changing the descriptions are

$$
\begin{aligned}
New &= \sum_{f \in f_2 \setminus f_1} Creation(f) + AspectNew(f) + DescripNew(f) \\
Creation(f) &= BaseCost(f) \cdot RS(f) \\
RS(f) &= min\{1 - ReusingSavings(f', f) \mid f' \in f_1\} \\
AspectNew(f) &= COMNEW \cdot FJANEW(f) \cdot \\
&\quad \sum_{a \in a_2} AspectBaseCost(a) \cdot FactorAspect(f, a) \\
COMNEW &= \prod_{\beta \in \mathcal{P}(a_2)} Compatibility(\beta) \\
FJANEW(f) &= \prod_{\beta \in \mathcal{P}(a_2)} FactorJointAspects(f, \beta) \\
DescripNew(f) &= MULNEW \cdot \sum_{d \in d_2} DescripBaseCost(d) \\
MULNEW &= \prod_{\beta \in \mathcal{P}(d_2)} Multiplicity(\beta)
\end{aligned}
$$

where $d_i = \{p \mid p \in \gamma_i \cap D\}$, $f_i = \{p \mid p \in \gamma_i \cap F\}$, and $a_i = \{p \mid p \in \gamma_i \cap A\}$

Fig. 5. Creation of new functionalities

computed by adding the *DescripBaseCost*'s reduced by the *similarity* factor, and increased by the *multiplicity* factor.

Figure 5 contains the costs related to the addition of new functionalities. We can split these costs into three parts. The cost due to aspects is the sum of the *AspectBaseCost*'s, increased by the *Compatibility* factor, and adapted by using the factors of the concrete functionality (*FactorAspect* and *FactorJointAspects*). The cost due to descriptions is computed from the sum of the *DescripBaseCost*'s, by increasing them with the *multiplicity* factor. Finally, reutilization costs are obtained by applying the *ReusingSaving* factor to *BaseCost*.

The generation of the cost function from the design functions allows to use cost-evaluated components instead of components. Actually, by applying the previous definition, a component $C = (P, P_o, \tau)$, where τ is a group of design functions, will be transformed into the cost-evaluated component $E = (P, P_o, Generated(\tau))$. ¿From now on, only cost-evaluated components will be used, as we will focus on modification costs to develop our methodology.

5 Transformation of Global Systems

In the previous section we showed how the *global cost* induced by a modification can be computed. Next we present a mechanism to transform a global system where a modification must be introduced. A *valid transition* represents a single step in the process of eliminating incoherence by replacing/modifying/purchasing a component.

Definition 3 Let $GS = (S, Av, M)$ and $GS' = (S', Av', M)$ be global systems such that $S = (K, G)$ and $S' = (K', G)$. Intuitively, given two global systems GS and GS', *transition* $GS \Rightarrow_{c,b} GS'$ denotes that the cost of passing from GS to GS' is given by c. The second parameter of the transition, that is b, is a boolean indicating whether a new component is purchased or not.

Let $c \in \mathbb{R}$ and $b \in Bool$. We say that $GS \Rightarrow_{c,b} GS'$ is a *valid* transition if $c \neq \infty$ and there exist two cost-evaluated components $E = (P, P_o, CostModif)$ and $E' = (P', P'_o, CostModif')$ such that $(E, i) \in K$ for some $i \in \mathbb{N}$, and $K' = (K \backslash \{(E, i)\}) \cup \{(E', i)\}$. In addition, the conditions of one of the following cases hold:

- *Modification of a component*: In this case, $c = CostModif(P, P_o, P', P'_o)$, $b = false$, $(E, true) \in Av$, $Av' = Av \cup \{(E', true)\}$, and $CostModif = CostModif'$.
- *Substitution of a component by an existing component*. The conditions to be fulfilled are $c = 0$, $b = false$, $(E', X) \in Av$ with $X \in \{true, false\}$, and $Av' = Av$.
- *Substitution of a component by the modification of an existing component*: There exits $(E'', true) \in Av$ with $E'' = (P'', P''_o, CostModif')$, $b = false$, $c = CostModif'(P'', P''_o, P', P'_o)$, and $Av' = Av \cup \{(E', true)\}$.
- *Purchase of a binary component*: $c = M(E')$, $b = true$, and $Av' = Av \cup \{(E', false)\}$.

□

Let us remind that $M(E')$ denotes the cost (in units of effort) of purchasing the component E'. Besides, let us remark that the third case generalizes the first two ones. However, for the sake of clarity, we prefer to keep redundancy in the previous definition. Among all the valid transitions leading to a desired global system, we have to consider the *cheapest* transition such that no component is purchased. Moreover, we also take into account those transitions where a component is purchased but its cost is cheaper than the previous one. So, we will have a set of *interesting* transitions to make any transformation of the system. The reason why we require purchased components to be cheaper that handmade components is that they are, in general, binary and cannot be modified, which is a drawback against those components that are directly developed by the team.

Definition 4 Let $GS = (S, Av, M)$ be a global system such that $S = (K, G)$. Let $A(GS, S_1) = \{GS \Rightarrow_{c,b} GS' \mid GS' = (S_1, Av', M')\}$ denote the set of transitions leading to the same (final) simple system S_1 and let $c_{min} \in \mathbb{R}$ be such that $c_{min} = min\{c \mid (GS \Rightarrow_{c,b} GS') \in A(GS, S_1) \ \wedge \ b = false\}$. We define the set of *interesting transitions* for GS and S_1, denoted by $Interesting(GS, S_1)$, as $X_1 \cup X_2$, where

$$X_1 = \{(GS \Rightarrow_{c,b} GS') \in A(GS, S_1) \mid b = false \ \wedge \ c = c_{min}\}$$

$$X_2 = \{(GS \Rightarrow_{c,b} GS') \in A(GS, S_1) \mid b = true \ \wedge \ c < c_{min}\}$$

□

Next we introduce the notion of *coherent system*, that is, a system where all the requirements demanded by components to neighbor components are fulfilled.

Definition 5 Let $GS = (S, Av, M)$ be a global system such that $S = (K, G)$. Let $E = (P, P_o, ModifCost)$ be a cost-evaluated component such that $(E, i) \in K$, for some $i \in \mathbb{N}$. We say that the conditions of the component E are *satisfied* in GS, denoted by $Satisfied(E, GS)$, if we have

$$P_o \subseteq \{P' \mid (E', i') \in K \ \wedge \ i' \in G(i) \ \wedge \ E' = (P', P'_o, ModifCost')\}$$

We say that GS is *coherent*, denoted by $Coherent(GS)$, if for any E such that $(E, i) \in K$, for some $i \in \mathbb{N}$, we have $Satisfied(E, GS)$. □

In order to reach coherent systems we will use *traces*, that is, sequences of concatenated interesting transitions.

Definition 6 A *trace* σ is a finite sequence

$$[GS_1 \Rightarrow_{c_1,b_1} GS_2, GS_2 \Rightarrow_{c_2,b_2} GS_3, \ldots, GS_{n-1} \Rightarrow_{c_{n-1},b_{n-1}} GS_n]$$

of interesting transitions. The *cost* of the trace σ, denoted by $cost(\sigma)$, is defined as $\sum_{i=1}^{n-1} c_i$. Besides, we say that σ is a *coherent modification* of GS_1, denoted by $CoheMod(GS_1, \sigma)$, if $Coherent(GS_n) = true$. □

Next we define the *objectives* of the modifications. These objectives are given by the set of new properties that each component has to fulfill in the global system. We will also characterize those systems fulfilling these new properties as well as the traces allowing to reach them. Moreover, we can consider those traces reaching a global coherent system GS such that the expected requirements are fulfilled.

Definition 7 Let $GS = (S, Av, M)$ be a global system such that $S = (K, G)$. A *set of new requirements* is a set of pairs $(i, Prop)$ where i is the (unique) identification number of a cost-evaluated component E associated with GS (that

is, $(E, i) \in K$) and *Prop* is a set of new (own) properties required to E. We say that GS *fulfills the set of new requirements* R, denoted by *fulfill*(GS, R), if for any $(i, Prop) \in R$ we have that there exists $E = (P, P_o, CostModif)$ such that $(E, i) \in K$ and $Prop \subseteq P$.

A trace $\sigma = [GS_1 \Rightarrow_{c_1,b_1} GS_2, \ldots, GS_{n-1} \Rightarrow_{c_{n-1},b_{n-1}} GS_n]$ is an *initial inclusion* of the set of new requirements R and the global system GS_1, denoted by $inclusion(R, GS_1, \sigma)$, if $fulfill(GS_n, R) = true$.

Let GS_1 be a global system and R be a set of new requirements. Let us consider a trace $\sigma = [GS_1 \Rightarrow_{c_1,b_1} GS_2, \ldots, GS_{n-1} \Rightarrow_{c_{n-1},b_{n-1}} GS_n]$ such that $inclusion(R, GS_1, \sigma)$ and $CoheMod(GS_1, \sigma)$. In this case we say that the trace σ is an *appropriate modification* of GS_1 according to R and we denote it by $ApproMod(GS_1, R, \sigma)$. □

Now that we have a method to reach a coherent design fulfilling the new requirements, we can choose among the possible solutions the most suitable one for our purposes. However, we may also have in mind a list of modifications/additions to be included in future releases of our system. We consider that *possible future modifications* are described as a set Q such that each of its elements is a set of new requirements. In this case, we may use the following notion.

Definition 8 Let GS_1 be a global system, R be a set of new requirements, and Q be a set of possible future modifications. Let

$$\sigma_1 = [GS_1 \Rightarrow_{c_1,b_1} GS_2, \ldots, GS_{n-1} \Rightarrow_{c_{n-1},b_{n-1}} GS_n]$$

$$\sigma_2 = [GS_n \Rightarrow_{c_n,b_n} GS_{n+1}, \ldots, GS_{n+m-1} \Rightarrow_{c_{n+m-1},b_{n+m-1}} GS_{n+m}]$$

be two traces such that $ApproMod(GS_1, R, \sigma_1)$ and there exits $R' \in Q$ such that $ApproMod(GS_n, R', \sigma_2)$. In this case we say that the trace $\sigma = \sigma_1 \circ \sigma_2$ is an *appropriate predictable modification* of GS_1 according to R and Q and we denote it by $ApproPredMod(GS_1, R, Q, \sigma)$. □

In the previous definition $\sigma_1 \circ \sigma_2$ denotes the concatenation of the traces σ_1 and σ_2. Let us remark that $cost(\sigma) = cost(\sigma_1) + cost(\sigma_2)$. We finish this section by showing an interesting characterization of the degree of coupling among components. Intuitively, if we have a new requirement for a component E and an appropriate modification changes another component E' then we consider that E and E' are coupled.

Definition 9 Let $GS_1 = (S, Av, M)$ be a global system such that $S = (K, G)$ and let $R = \{(j, Prop)\}$ be a (unitary) set of new requirements. Let E, E' be cost-evaluated components such that $(E, j) \in K$ and $E' = (P', P'_o, CostModif')$,

with $(E', m) \in K$, for some $m \in \mathbb{N}$. Let us consider a trace

$$\sigma = [GS_1 \Rightarrow_{c_1,b_1} GS_2, \ldots, GS_{n-1} \Rightarrow_{c_{n-1},b_{n-1}} GS_n]$$

such that $GS_n = (S', Av', M')$ with $S' = (K', G')$. We say that the trace σ is a *coupled modification* of E and E' with respect to R in GS, denoted by $coupled(GS, E, E', R, \sigma)$, if $ApproMod(GS_1, R, \sigma)$ and there exists a component $E'' = (P'', P''_o, CostModif'')$ such that $(E'', m) \in K'$ and $P'' \neq P'$. □

Once we have presented concepts as *costs*, *appropriate modifications*, and *appropriate predictable modifications*, it is possible to automatically apply them to real systems. Thus, they can guide us while searching for the cheapest modifications, the designs easier to modify, or the designs with the smallest coupling factors. This can be done in such a way that the designer does not need to perform by hand a planning of all the changes that are needed to solve the dependency issues. Actually, once the designer specifies the system including the description of the components (own properties, properties demanded to other components, simple design functions), the dependencies among the components (that is, the dependencies graph), and (optionally) a set of market components with their sale prices, then any possible modification of the system can be analyzed in an automatic way, returning data about modification costs, degree of reusability, and degree of coupling. So, a designer can use our methodology to study a set of possible design alternatives, comparing the estimated results returned automatically and choosing manually the solution which better fits into her necessities according to her experience.

6 Conclusions

In this paper we have presented a methodology to assist in the process of modifying the design of a component-based system. In particular, it provides analyses of the impact of each modification. The method takes as input the specification of the components, including the dependencies structure, the properties that each component holds, and the properties that each component requires from other components. In addition to that, some functions are provided in order to specify relations among properties. Those functions facilitate the estimation of costs. All these simple functions are used to automatically generate a single global cost function. Taken into account this function, the cost of each possible modification can be automatically computed, enabling the systematic study of the different alternatives to perform the desired modifications. This systematic study includes information about costs, reusability, and coupling of the different alternatives.

We are aware that the use of our methodology requires a certain degree of

familiarity with the process, starting from the identification of properties, and going through the definition of the functions expressing these properties. Nevertheless, despite the apparent complexity, the experiments that we have performed in the framework of WHAT are very encouraging. It is worth to point out that in the case of WHAT there were designers/developers who did not participate in the definition of our methodology.

Acknowledgements

We would like to thank the students who collaborated in the application of our methodology during the development of WHAT.

References

[1] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 1998:37–46, 1998.

[2] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE TRansactions of Software Engineering*, 20(6):476–493, 1994.

[3] I. Crnkovic. Component-based software engineering: New challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[4] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Specification, implementation, and deployment of components. *Communications of the ACM*, 45(10):35–40, 2002.

[5] I. Crnkovic and M. Larsson. Challenges of component-based development. *Journal of Systems and Software*, 61:201–212, 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[7] S.A. Hissam, G.A. Moreno, J. Stafford, and K.C. Wallnau. Packaging predictable assembly. In *IFIP/ACM Working Conference on Component Deployment, LNCS 2370*, pages 108–124. Springer, 2002.

[8] T. Capers Jones. *Programming Productivity.* McGraw-Hill, 1986.

[9] T. Capers Jones. *Applied Software Measurement: Assuring Productivity and Quality.* McGraw-Hill, 1991.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming, LNCS 1241*, pages 220–242. Springer, 1997.

[11] N. López, M. Núñez, I. Rodríguez, and F. Rubio. Including malicious agents into a collaborative learning environment. In *8th Intelligent Tutoring Systems, LNCS 2363*, pages 51–60. Springer, 2002.

[12] N. López, M. Núñez, I. Rodríguez, and F. Rubio. WHAT: Web-based Haskell Adaptive Tutor. In *10th Artificial Intelligence: Methodologies, Systems, and Applications, LNAI 2443*, pages 71–80. Springer, 2002.

[13] D. Mason. Probabilistic analysis for component reliability composition. In *5th ICSE Workshop on Component-Based Software Engineering*, 2002.

[14] G.A. Moreno, S.A. Hissam, and K.C. Wallnau. Statistical models for empirical component properties and assembly-level property predictions: Toward standard labelling. In *5th ICSE Workshop on Component-Based Software Engineering*, 2002.

[15] M. Núñez, P. Palao, and R. Peña. A second year course on data structures based on functional programming. In *Functional Programming Languages in Education, LNCS 1022*, pages 65–84. Springer, 1995.

[16] R. Peña, Y. Ortega-Mallén, and F. Rubio. Teaching monadic algorithms to first-year students. In *Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99*, pages 33–45, 1999.

[17] I. Rodríguez and F. Rubio. A framework for selecting components automatically: A first approach. In *TACOS'03 - Test and Analysis of Component Based Systems*. Electronic Notes in Theoretical Computer Science 82, 2003.

[18] J. Rumbaugh, I. Jacobsen, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1997.

[19] J. Stafford and J.D. McGregor. Issues in predicting the reliability of composed components. In *5th ICSE Workshop on Component-Based Software Engineering*, 2002.

[20] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[21] C. Szyperski. Components and the way ahead. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 1. Cambridge University Press, 2000.

[22] H. Washizaki, H.Yamamoto, and Y.Fukazawa. A metrics suite for measuring reusability for software components. In *Proc. of the 9th International Symposium on Software Metrics (Metrics 2003)*. IEEE CS, 2003.