A Framework for Selecting Components Automatically: A First Approach

Ismael Rodríguez and Fernando Rubio

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

Abstract

CBSE (Component Based Software Engineering) is devoted to develop software projects in such a way that the final applications can be created by using plug and play generic components. It requires to properly choose the needed components from those available. However, that task is not simple at all. In this paper we present a way to automatically search, find and choose generic commercial components in a market of components. The method will be based on the specification of components in a standardized way, so that both customers and manufacturers can refer to them by using the same notation. Besides, both customers and manufacturers will give heuristics to estimate the costs of modifying the desired and sold components respectively. Then, the total cost needed to adapt a purchased component will be computed using both heuristics jointly, taking into account their confidence level. Hence, in order to choose the component that better fits his requirements, the customer will be able to take into account not only the sales price of the component, the confidence on the vendor, or the expectations of future versions to be developed, but also the adaptation costs.

1 Introduction

During the last years there has been a great effort in the implementation of tools and in the definition of well-founded methodologies to facilitate the task of system designers. In this line, Component-Based Software Engineering, in short CBSE, is a very active field of research and practice (see e.g. [1,2,4,11] for a description of issues in CBSE).

One of the main concerns in CBSE consists in the study of properties, as reliability, related to component composition (e.g. [7,10,5]). Actually, CBSE encourages the development of projects as a plug and play process where

---

1 Work supported in part by the CICYT projects MASTER and AMEVA.

©2003 Published by Elsevier Science B. V.
generic components are selected and connected to build up the final product. So, it is a critical issue to find the *cheapest* components which fit in our design from a wide repository of market components. In fact, *cheapest* is a complex term in CBSE, as many factors must be considered: The sales price of the component, the confidence on the vendor, the expectations of future versions to be developed, and the effort needed to adapt the purchased component to our actual necessities. While all of these four factors must be considered, we will mainly focus on the latter, the *adaptation costs*. Actually, this is a critical point. Providing a practical *semiautomatic* selection methodology is necessary to cope with the huge repositories of available components, which are in continuous expansion. However, it is an extremely hard issue, as it requires both defining an appropriate standardization of properties of components for customers and vendors, and including accurate cost models for the components. So, we propose a first approach to the critical topic of the automatic estimation of adaptation costs in CBSE. We concentrate on the study of a formal framework for structuring components in such a way that, given a market, it allows to obtain the cheapest component for a given requirement from the market of components, focusing on the adaptation costs of components.

CBSE should be based on combining the use of generic components bought to a certain manufacturer, other generic components own by the user, and some extra source code developed to serve as the *glue*. That is, this glue should adapt the generic components to the exact peculiarities of the current software project. Obviously, when creating a components catalog, it is not feasible to try to create all the possible components that could be required. In contrast, a wide repository of generic components documented by using a common notation is needed. Moreover, an appropriate structure of the repository must be used to be able to perform systematic searches. However, we claim that this kind of repositories are not enough for CBSE, as the decision of which component to select must depend on the cost of adapting it to our necessities. Thus, it does not matter whether we have access to the source code of the component or we just have a binary distribution: In any case we will usually have to adapt it. In the first case we can modify the component itself, while in the second we have to create a new layer on top of it.

Traditionally, the estimation of these modification costs used to be handled by the purchaser of the component, as he knows what he wants to obtain. However, the manufacturer is who really knows the complexity needed to modify a component in order to obtain other functionalities similar to those already provided. Thus, we claim that he should specify not only the functionalities, but also the costs needed to modify them. By doing so, the computation of the costs will be split into two steps. Firstly, the manufacturer will have to deal with the costs due to modifying his components to obtain some other typical functionalities. Secondly, the purchaser will deal with the costs due to adapting common functionalities to his actual necessities. In both cases, the same kind of heuristics could be used to compute approximations to such costs. Let
us remark that the heuristics defined by the manufacturer and the customer will take as reference different components. The domain of the heuristics of the manufacturer is located near the component he sells, so that the confidence level will be lost as the component gets more different (obviously, the heuristic cannot take into account any possible modification). Besides, the heuristic of the customer will increase its precision as the characteristics of the component are closer to those of the component he is interested in. Hence, to compute the modification costs for each of the transitions during the modification, we will need to take into account the varying precision degree of both heuristics.

All of these computations require a powerful abstraction which allows modeling the real components by using a formal notation. Obviously, such abstraction must consider the relevant properties that hold in the component. As we will deal both with open-source and binary unmodifiable components, and the adaptation mechanism lies on different kinds of modifications, we will distinguish between the set of unmodifiable parts of a component, (e.g. a binary purchased component), and the modifiable interface/adaptation layer. Actually, if the component bought was open source, the former set will be empty, and the latter will include both the original component and the modifications introduced by the customer. Obviously, if there is any unmodifiable property then modifiable properties play the role of the component interface, while otherwise they play the role of all the properties. Thus, we will also assume that they play the role of the actual properties that the component, as a whole, fulfills. So, for any unmodifiable property, the corresponding modifiable property should also hold, as otherwise it will not be considered an actual property.

Besides, as it is in pointed out in [11], components cannot be considered as isolated entities (introvert components), as there are elements that are to be connected to other components (extrovert components). The relevant information of a component will not be complete without the set of properties that it requires to neighbors components. This information must be specified to accurately analyze the usefulness of these components for a given design.

Finally, component abstractions will require a function which estimates the modification costs needed to obtain a set of properties assuming that another set of properties holds initially. This is not a trivial task, as there is a huge number of cases that must be considered. So, other simpler functions will be provided to the designer, making easier the task of finding and specifying relevant cost information. The goal is to be able to automatically infer the whole information taking as basis these simple functions.

The aim of this paper is to present a framework where generic commercial components can be semiautomatically searched, found and chosen in a market of components. To do so, estimations of the costs needed to adapt the purchased components will be computed. These estimations will be computed according to some modification cost heuristics given both by manufacturers and customers. It requires the development of a common standardized structure of the characteristics the components could have. This structure will
allow both the unified usage of the two heuristics and a way to extrapolate costs for undefined modifications. So, from a huge amount of available components, the methodology will automatically find a reduced set including the most suitable components according to the heuristics. Then, the designer will manually choose the component to purchase from this reduced set.

The rest of the paper is structured as follows. In Section 2 we present the basic definitions needed in the rest of the paper. Then, Section 3 presents how to compute the cost of a modification to a component. Afterwards, Section 4 shows the mechanism to obtain the best components from the market. Finally, in Section 5 we present our conclusions. Due to lack of space, we have not included an illustrative example, but it can be found in [9].

2 Basic Elements

In this section we define the basic concepts that will be used in the rest of the paper. We start by specifying the different kinds of properties that components may fulfill. As usual, we have both functional and extrafunctional properties (see e.g. [3]). 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. Functionalities are those properties defining what the component does. For example, “recording user data”, “generating queries for the balance”, “saving persistent data”, etc. Aspects define how the component performs its tasks. They approximately correspond to the classical definition of aspect in aspect-oriented programming [6] in the sense that they cross-cut the basic functionalities. For example, “data of clients include name, ID, and balance”, “the communications protocol is Ethernet”, etc.

In general, we will denote by $D_1$, $F_1$ and $A_1$ the sets of descriptions, functionalities, and aspects respectively. These kinds of properties will be used in components to denote its set of modifiable properties, its set of unmodifiable properties, and its set of properties it requires to neighbors. Nevertheless, as sometimes we will need to refer properties of all of these three sets in the same expression, we will need to rename them. Hence, while these sets will be called $D_1$, $F_1$ and $A_1$ in the case of modifiable properties, they will be called $D_2$, $F_2$ and $A_2$ and $D_0$, $F_0$ and $A_0$ in the cases of sets of unmodifiable and required properties, respectively. Besides, $D = D_1 \cup D_2$, $F = F_1 \cup F_2$ and $A = A_1 \cup A_2$ will denote the sets of own properties of the component.

**Definition 2.1** A component $C$ is a tuple $(P_1, P_2, P_o, \tau)$, where $P_1 \subseteq D_1 \cup F_1 \cup A_1$, $P_2 \subseteq D_2 \cup F_2 \cup A_2$, and $P_o \subseteq D_o \cup F_o \cup A_o$ are the set of own properties, modifiable and unmodifiable, and the set of properties required to neighbors, respectively, and $\tau$ is the group of basic cost functions for this component. We denote by $\xi$ the set of all the components.

A cost-evaluated component $E$ is a tuple $(P_1, P_2, P_o, \text{ModifCost})$, where $P_1$,
As we have already indicated, the description of a component \((P_1, P_2, P_0, \tau)\) is given by the modifiable properties fulfilled by the component (i.e. \(P_1\)), which are also considered as the actual properties of the component, the unmodifiable properties of the component (i.e. \(P_2\)), the properties expected from the components related to it (i.e. \(P_0\)), and a tuple of basic cost functions (i.e. \(\tau\)). The next section formally presents the kind of functions appearing in these tuples. A cost-evaluated component represents a different view of a component. Intuitively, the group of basic cost functions is replaced by a cost function computing the cost of modifying the component by taking into account the initial properties (first tuple of three arguments of \(\text{ModifCost}\)) and the properties after the modification is performed (second tuple of three arguments of \(\text{ModifCost}\)). Actually, this function returns a pair of values: the cost of the modification; and the confidence degree of the prediction, a value in \([0..1]\).

To be able to search for the components with the appropriate properties, a classification of such properties is required. Thus, we will use a hierarchy for each of the three properties: Descriptions, functionalities, and aspects. For each hierarchy, its top will be denoted by undefined (⊥). For each description (or functionality or aspect), it will be possible to access both to its subclasses and to its superclass. Hierarchies are the basis to extrapolate the modification costs. We will use subclasses and superclasses to estimate the modification costs of those entries of the modification function that are undefined. However, this will make the confidence on the results to be reduced. See [9] for details.

### 3 Computing the Cost of Adapting a Component

In this section we present the definition of the functions computing the cost of performing a modification in a component. First, we give a set of simple functions, each of them dealing with an independent concept. Then, we show how these simple functions are combined to compute the overall cost function.

#### 3.1 Generating the Basic Cost Functions

Given a component and its current properties, the global cost function 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 one by one. In contrast, a method allowing the designer to specify general rules should be provided, but the definition of as many special cases as needed must be permitted. Following these ideas, in our framework, the designer will only need to declare some rules for each property (based on heuristics). An example of
such a heuristic is that, by default, changing into a configuration where the properties of the component to be modified and those required to neighbor components are similar will be cheaper than changing 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 this 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. Our set of basic functions represent such a compromise between expressiveness 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 to other components.

For each of these functions, there will be some special outputs. In particular, each function has associated a default value that is applied in case the given modification does not add any new cost (e.g. multiplicative factors equal to 1). More importantly, each function will be allowed to return the undefined value (⊥). In fact, we will assume that ⊥ belongs to the domain of all the types we will use. This will be required when the input received is specified with so many details that it exceeds what it was taken into account in the definition of the function. In these cases, the estimated costs will be computed by using the inputs corresponding both to its superclass and to its subclasses. Obviously, the confidence on the value returned by the heuristic functions will be smaller when we need to use information about the subclasses or superclasses. In fact, the confidence will get smaller as the antecessor (or predecessor) classes to be used are further in the hierarchy of classes.

The ideas underlying the definition of all the functions are similar: They compute the impact on the modification costs due to including together some combination of properties. All of them are defined in [9]. Due to lack of space, here we will only describe one of these functions. We can reduce the costs of implementing a functionality by taking into account similarities with other functionalities already implemented. This fact is considered by the function $ReusingSavings : F \times F \rightarrow [0,1]$, where $ReusingSavings(f_1, f_2)$ defines the proportion of the cost of functionality $f_2$ that can be saved by using functionality $f_1$. Default value is 0. A value 1 means that $f_1$ is a generalization of the functionality $f_2$. Let us remark that $F$ denotes the set of own functionalities (including both modifiable and unmodifiable properties). The other functions are $Similarity$, $DescripBaseCost$, $Multiplicity$, $BaseCost$, $FactorAspect$, $FactorJointAspects$, $AspectBaseCost$, $Migration$, and $Compatibility$.

These functions will be defined both by the manufacturer and by the purchaser. The first one will define them for the properties provided by his components, while the second one will provide them for the properties that he actually needs. In any case, the functions will allow estimating the influence
of the properties of a component on the cost needed to modify it. As extrapolating the costs using superclasses and/or subclasses reduces the precision of the results, we define new versions of these functions which consider this fact. When the result of a function is undefined for some parameters, we search for sets of related parameters for which the function is defined. These related parameters will be obtained by searching the subclasses and superclasses of the original parameters, and the confidence degree will be reduced as the related parameters are further in the hierarchy. So, from now on we will use modified versions of the functions that return pairs of elements: The estimated cost of the modification and the confidence degree on this cost. See [9] for details.

We will now present the functions denoting the conditions about own properties and properties demanded to neighbor components. Let us remind that $D_o$, $F_o$, and $A_o$ are the properties demanded to neighbors. In [9] the precise notation needed to tackle with these functions is presented. Basically, each property states a logical formula which must hold among own and external properties. Undefined entries are estimated from descendants in the hierarchy. One of these functions is \( \text{Func}^2_{\text{Asp}}: F_1 \rightarrow L(A_1 \cup A_o) \), where \( L \) denotes the set of logical formulas, and \( \text{Func}^2_{\text{Asp}}(f) \) denotes the conditions (both internal and external) that must be fulfilled by aspects when functionality \( f \) holds. Default value is true, denoting that nothing is required. The other functions are \( \text{Desc}^2_{\text{Desc}}, \text{Desc}^2_{\text{Func}}, \text{Func}^2_{\text{Func}}, \) and \( \text{Func}^2_{\text{Asp}} \).

### 3.2 Generation of the Global Cost Function

Once we have defined the basic functions to compute costs, we will obtain the global cost function by using an automatic method. Let us remark that the user will not need to define complex functions: It will be enough to define simple functions as the ones presented in the previous subsection, and the whole cost function will be automatically obtained.

Let \( \tau \) denote a group of basic cost functions, where each function is denoted by its sort (e.g. \( \text{Similarity} \) is the Similarity function). Let \( \theta \) denote a hierarchy of component properties. The cost function generated from \( \tau \), denoted by \( \text{Generated}(\tau) \), is a function \( \text{ModifCost} : (P(P_1) \times P(P_2) \times P(P_o)) \times (P(P_1) \times P(P_2) \times P(P_o)) \rightarrow (\mathbb{R}^+ \cup \{\infty\}) \times [0,1] \), with \( P_1 = D_1 \cup F_1 \cup A_1 \), \( P_2 = D_2 \cup F_2 \cup A_2 \), and \( P_o = D_o \cup F_o \cup A_o \). Intuitively, if \( (c_1, c_2) = \text{ModifCost}((\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2)) \), we have that, when modifying a component from the current own properties (modifiable and unmodifiable) and the current demanded properties \( (\alpha_1, \beta_1, \gamma_1) \) respectively to reach the future own and required properties \( (\alpha_2, \beta_2, \gamma_2) \), \( c_1 \) specifies the cost of this modification, while \( c_2 \) denotes the confidence on the previous value. A value of \( c_1 \) equal to \( \infty \) denotes that the modification is impossible, and a value of \( c_2 \) equal to 0 denotes that confidence on \( c_1 \) is null. Function \( \text{ModifCost} \) is defined as:

\[
\text{ModifCost}(((\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2)) = \begin{cases} 
\text{Old} + \text{New} & \text{if } \text{CondHolds} \land \beta_1 = \beta_2 \\
(\infty, 1) & \text{otherwise}
\end{cases}
\]
In the previous expression, \( \text{Old} \) denotes the cost due to the adaptation of old functionalities, \( \text{New} \) is the cost due to the creation of new functionalities, and \( \text{CondHolds} \) indicates that all of the condition functions of the component over own properties and properties demanded to neighbors hold. \( \text{CondHolds} \), \( \text{Old} \), and \( \text{New} \) are formally defined in [9]. Intuitively, \( \text{Old} \) and \( \text{New} \) compose the effects of each basic cost function to generate the global cost of the whole modification. The costs due to the adaptation of old functionalities and the inclusion of new ones will depend on the old and new descriptions and aspects.

Let us remark that a cost of \( \infty \) will be returned by \( \text{ModifCost} \) either if the sets denoting the unmodifiable properties \( (\beta_1, \beta_2) \) change, which is obviously forbidden, or if the functions denoting the own properties and the properties demanded to neighbor components \( \text{(CondHolds)} \) do not hold. Besides, let us note that \( \text{Old} \) and \( \text{New} \) are pairs of values \( \text{(cost, confidence)} \). Thus, we suppose that their sum is overloaded so that the first component of the result is the addition of costs and the second is the weighted average of confidences.

The generation of the cost function from the simpler basic cost functions allows using cost-evaluated components instead of components. Actually, by applying the previous definition, a component \( C = (P_1, P_2, P_o, \tau) \), where \( \tau \) is a group of basic cost functions, will be transformed into the cost-evaluated component \( E = (P_1, P_2, P_o, \text{Generated}(\tau)) \). From now on, only cost-evaluated components will be used, as we will focus on modification costs.

4 Modifying a Component

Let us remark that in the process of adapting an existing component to a desired one, we will need to perform several intermediate transitions, where each of them will have attached a cost. Let us also remark that the confidence on the heuristics given by the manufacturer is better when we are dealing with components closer to those provided by him, while the heuristics of the purchaser are better for components with properties closer to those required by him. Thus, the heuristic to be used in each transition will depend on the distance from the current component to both the existing and the desired ones. Notice that it is not reasonable to perform the whole transformation in a single transition, because in that case we would lose precision, as we would not be able to make good use of the different distances to the existing and desired components. The next definition is used to deal with the transitions required to transform a component. From now on \( MC \) denotes \( \text{ModifCost} \).

**Definition 4.1** Let \( E_1 = (\alpha_1, \beta_1, \gamma_1, MC) \) and \( E_2 = (\alpha_2, \beta_2, \gamma_2, MC) \) be two cost-evaluated components. Then, the *transition* from \( E_1 \) to \( E_2 \) is denoted as \( E_1 \Rightarrow_{(c_1,c_2)} E_2 \), where \( (c_1,c_2) = MC\left((\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2)\right) \).

The following two definitions denote the transitions of *double* components, i.e., those taking into account the information gathered from the specifications
of manufacturer and customer. In the first definition, this kind of double component is defined. This component takes into account the general confidence on the two cost functions provided by manufacturer and customer. In the second, transitions for this kind of component are defined.

**Definition 4.2** Let $E_1 = (\alpha, \beta, \gamma, MC_1)$ and $E_2 = (\alpha, \beta, \gamma, MC_2)$ be two cost-evaluated components. Let $\conf_1$ and $\conf_2$ be the confidence levels of the cost functions $MC_1$ and $MC_2$. The doubly interpreted cost-evaluated component (DICC) created from $E_1$ and $E_2$ is $(\alpha, \beta, \gamma, (MC_1, MC_2), (\conf_1, \conf_2))$. □

**Definition 4.3** Let us consider two DICCs $D_1$ and $D_2$, where $D_1$ is defined as $D_1 = (\alpha_1, \beta_1, \gamma_1, (MC_1, MC_2), (\conf_1, \conf_2))$ and $D_2$ is defined as $D_2 = (\alpha_2, \beta_2, \gamma_2, (MC_1, MC_2), (\conf_1, \conf_2))$. Besides, let us define $(a_1, a_2) = MC_1((\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2))$ and $(b_1, b_2) = MC_2((\alpha_1, \beta_1, \gamma_1), (\alpha_2, \beta_2, \gamma_2))$, and let $w = \frac{(a_2 \cdot \conf_1)^{\delta}}{(a_2 \cdot \conf_1)^{\delta} + (b_2 \cdot \conf_2)^{\delta}}$, where $\delta$ is a given constant. The transition from $D_1$ to $D_2$ is denoted as $D_1 \Rightarrow (c_1, c_2) \Rightarrow D_2$, where $c_1 = w \cdot a_1 + (1 - w) \cdot b_1$ and $c_2 = w \cdot \conf_1 \cdot a_2 + (1 - w) \cdot \conf_2 \cdot b_2$. □

Let us remark that, while computing the cost of the transition, the estimation with the biggest confidence degree will increase geometrically its weight with respect to the other. Besides, two different kinds of confidence are considered. First, the confidence related to extrapolations of the cost functions ($a_2$ and $b_2$). Second, the global confidence on the cost functions ($\conf_1$ and $\conf_2$). Now, we can deal with the whole chain of transitions needed to transform the original component into the desired one. It is only necessary to record the costs of each of the transitions, as shown in the next definition.

**Definition 4.4** Let $D_1 \Rightarrow (a_1, b_1) \Rightarrow D_2$, ..., $D_{n-1} \Rightarrow (a_{n-1}, b_{n-1}) \Rightarrow D_n$ be $n - 1$ transitions of DICCs, where for all $1 \leq i \leq n - 1$ we have that $a_i \neq \infty$. We say that $\sigma = [D_1, \ldots, D_n]_{c_1, c_2}$ is a modification trace, where $c_1$ and $c_2$, the cost and confidence of the trace respectively, are defined as $(c_1, c_2) = \sum_{i=1}^{n-1} (a_i, b_i)$. □

In the previous definition, the $\sum$ operator is based on the overloaded addition that uses the weighted average to compute the second element of the pair. Now, we can specify the whole mechanism. Firstly, we have to define what the customer desires to obtain, and then we define how to obtain it.

**Definition 4.5** A set of desired components of the customer, in short SDCC, is a set of tuples $(E, Ad, conf)$ where $E$ is a cost-evaluated component, $Ad$ is the adaptation cost of the component $E$ to the customer design, and $\conf$ is the confidence degree on the modification costs estimated for $E$. □

In order to compute $Ad$, we can use the methodology proposed in [8], whose description is out of the scope of this paper. Finally, we are in a position to describe the process of searching components in a market.

**Definition 4.6** A market of components $M$ is a set of pairs $(E, Pr, conf)$ where $E$ is a cost-evaluated component, $Pr$ is the market price to purchase
component $E$, and $conf$ is the confidence degree on the modification costs estimated for $E$.

Let us note that the confidence on the cost estimations given by the vendor will be set up by the customer. Besides, let us now describe the way in which the properties are classified. Let us suppose that $E = (P_1, P_2, P_o, MC)$ is a component that can be sold. If $E$ is a binary component, then all of its properties are unmodifiable, so they belong to $P_2$, and $P_1 = \emptyset$. On the contrary, in case it is open-source, $P_1$ will contain the properties, while $P_2 = \emptyset$.

Assuming that the economic cost of buying a component can be given in the same measure units as the cost of modifying a component, we can trivially compute the best solution not only in terms of programming effort, but also in terms of economical issues.

**Definition 4.7** Let $M$ be a market of components and $S$ be an SDCC. Let $(E_1, Pr, conf_1) \in M$ and $(E_2, Ad, conf_2) \in S$, with $E_1 = (\alpha_1, \beta_1, \gamma_1, MC_1)$ and $E_2 = (\alpha_2, \beta_2, \gamma_2, MC_2)$. Let $D_1$ and $D_2$ be two DICCs where $D_1$ is defined as $(\alpha_1, \beta_1, \gamma_1, (MC_1, MC_2), (conf_1, conf_2))$, while we define $D_2$ as the tuple $(\alpha_2, \beta_2, \gamma_2, (MC_1, MC_2), (conf_1, conf_2))$. Finally, let $\sigma = [D_1, \ldots, D_2]_{a_1,a_2}$ be a modification trace. A purchasing process from component $E_1$ to $E_2$ is a tuple $(E_1, E_2, c_1, c_2)$, where $c_1 = Pr + a_1 + Ad$ and $c_2 = \frac{a_1 \cdot a_2 + Pr + Ad \cdot conf_2}{c_1}$. \qed

In the previous definition, in order to compute the global cost, we take into account the economical price of the component ($Pr$), the adaptation cost to create the desired component from the purchased component ($a_1$), and the cost to adapt the final component to the customer design ($Ad$). The confidence of the final cost is calculated from the confidence of the individual costs, weighting each confidence according to the cost added, and taking into account that the confidence on the vendor price is 1.

## 5 Conclusions

In this paper we have presented a semiautomatic methodology to help in the process of finding the best commercial components available in the market. Searching for a component is reduced to the following steps:

- Abstracting the components by analyzing their main properties.
- Creating two (manufacturer and customer) heuristic functions computing the cost for creating a new component on the basis of similar components. Such a function depends on the characteristics where the components differ.
- Searching in the market in order to find the components whose characteristics are closer to what it is needed. This search requires the market to be appropriately structured and standardized.
- Given the components found in the previous step, computing the costs needed to adapt them, and selecting those with the lowest costs. This will require using the heuristics given by the manufacturer and by the purchaser.
Choosing the appropriate component by considering both the costs of adapting it, the costs of buying it, and the confidence on these previous values. Let us remark that, by using our framework, the purchaser has an automatic method to search inside markets, and to find the components that better fit his necessities. Obviously, the expert designer should analyze not only one component, but a small subset corresponding to the best components found by the methodology. By doing so, he will be able to apply his experience to refine the results. Nevertheless, the point is that a small number of interesting components can be found automatically from a huge number of available components. Besides, this search will not only obtain the cheapest solutions, but also the confidence degree on the predictions, so that it is possible to choose which is the best solution by finding a tradeoff between low cost and low risk.

References


