Automated Selection of Software Components Based on Cost/Reliability Tradeoff*

Vittorio Cortellessa[1], Fabrizio Marinelli[1], and Pasqualina Potena[2]

[1] Dipartimento di Informatica
Università dell'Aquila
Via Vetoio, 1, Coppito (AQ), 67010 Italy
{cortelle,marinelli}@di.univaq.it
[2] Dipartimento di Scienze
Università "G.D'Annunzio"
Viale Pindaro, 42, Pescara, 65127 Italy
potena@sci.unich.it

Abstract. Functional criteria often drive the component selection in the assembly of a software system. Minimal distance strategies are frequently adopted to select the components that require minimal adaptation effort. This type of approach hides to developers the non-functional characteristics of components, although they may play a crucial role to meet the system specifications. In this paper we introduce the CODER framework, based on an optimization model, that supports "build-or-buy" decisions in selecting components. The selection criterion is based on cost minimization of the whole assembly subject to constraints on system reliability and delivery time. The CODER framework is composed by: an UML case tool, a model builder, and a model solver. The output of CODER indicates the components to buy and the ones to build, and the amount of testing to be performed on the latter in order to achieve the desired level of reliability.

1 Introduction

When the design of a software architecture reaches a good level of maturity, software engineers have to undertake selection decisions about software components. COTS have deeply changed the approach to software design and implementation. A software system is ever more rarely built "from scratch", as part of the system comes from buying/reusing existing components.

Even though in the last years numerous tools have been introduced to support decisions in different phases of the software lifecycle, the selection of the appropriate set of components remains a hard task to accomplish, very often left to the developers' experience. Without the support of automation, the selection is frequently driven from functional criteria related to the distance of the characteristics of available components from those specified in the architectural description. This is due to the deep understanding that software designers have developed on functional issues, as well as to the

* This work has been partially supported by the PLASTIC project: Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication. EC - 6th Framework Programme. http://www.ist-plastic.org

introduction of sophisticated compositional operators (e.g. connectors with complex internal logics) that help to assembly systems satisfying the functional requirements.

As opposite, limited contributions have been brought to support the selection of components on the basis of their non-functional characteristics. As a consequence, software developers have no automated tools to support the analysis "aimed at characterizing the performance and reliability behavior of software applications based on the behavior of the "components" and the "architecture" of the application" [6]. This analysis might be used to answer questions such as: (i) which components are critical to the performance and reliability of the application? and (ii) how are the application performance and reliability influenced by the performance and reliabilities of individual components? If the software application is to be assembled from a collection of components, then answer to such questions can help the designers to make decisions such as which components should be picked off the shelf, and which components should be developed in-house [6].

A recent empirical study on COTS-based software development [17] shows that component selection is part of new activities integrating the traditional development process, and it is always based on the experience of project members. The same study evidences a similar practice that bases the COTS component selection either on the developer familiarity or on license issues and vendor reputation. None of the studied projects uses decision-making algorithms.

Beside all the above considerations, real software projects ever more suffer from limited budgets, and the decisions taken from software developers are heavily affected by cost issues. The best solutions might not be feasible due to high costs, and wrong cost estimations may have a critical impact for the project success. Therefore tools that support decisions strictly related to meet functional and non-functional requirements, while keeping the costs within a predicted budget, would be very helpful to the software developer's tasks.

In this paper we introduce CODER (Cost Optimization under DElivery and Reliability constraints), a framework that helps developers to decide whether buying or building components of a certain software architecture. Once built a software architecture, each component can be either bought, and probably adapted to the new software system, or it can be developed in-house. This is a "build-or-buy" decision that affects the software cost as well as the ability of the system to meet its requirements.

CODER supports the component selection basing on cost, delivery time and reliability characteristics of the components. We assume that several instances of each software component may be available as COTS. Basically, the instances differ with respect to cost, reliability and delivery time. Besides, we assume that several in-house instances of each software component may be built. In fact, the developers of a system could build an in-house component by adopting different strategies of development. Therefore, the values of cost, reliability and delivery time of an in-house developed component could vary due to the values of the development process parameters (e.g. experience and skills of the developing team). CODER indicates the assembly of (in-house and COTS) components that minimizes the cost under constraints on delivery time and reliability of the whole system. In addition, for each in-house developed component CODER suggests the amount of testing to perform in order to achieve the required level of reliability.

The paper is organized as follows: in section 2 we provide the formulation of the optimization model that represents the CODER core; in section 3 we introduce the CODER structure and underlying mechanisms; in section 4 we illustrate the usage of CODER in the development of a mobile application; in section 5 we summarize recent work in software cost estimation vs. quality attributes and outline the novelty of our approach with respect to the existing literature; conclusions are presented in section 6. In [5] we have collected all the details that are not strictly necessary for this paper understanding.

2 The Optimization Model Formulation

In this section, we introduce the mathematical formulation of the optimization model that CODER generates and solves, and that represents the core of our approach ([1]).

Since our framework may support different lifecycle phases, we adopt a general definition of component: a component is a self-contained deployable software module containing data and operations, which provides/requires services to/from other components. A component instance is a specific implementation of a component.

The solution of the optimization model determines the instance to choose for each component (either one of the available COTS products or an in-house developed one) in order to minimize the software costs under the delivery time and reliability constraints. Obviously when no COTS products are available the in-house development of a component is a mandatory decision whatever being the cost incurred. Viceversa for components that cannot be in-house built (e.g. for lack of expertise) one of the available COTS products must be chosen.

Due to our additional decision variables, the model solution also provides the amount of testing to be performed on each in-house component in order to achieve a certain reliability level.

2.1 The Problem Formulation

Let S be a software architecture made of n components. Let J_i ($\bar{J}_i$) be the set of COTS (in-house developed) instances available for the i-th component, and $m = \max_i |J_i \cup \bar{J}_i|$.

Let us suppose to be committed to assemble the system by the time T while ensuring a minimum reliability level R and spending the minimum amount of money.

COTS Component Model Parameters

The parameters that we define for a COTS product $C_{ij} \in J_i$ are:

- the cost c_{ij};
- the delivery time d_{ij};
- the average number s_i of invocations;
- the probability μ_{ij} of failure on demand.

[1] For sake of readability, we report model details in [5], thus we ask readers interested to a deeper understanding of the model construction to refer to [5].

The estimate of the cost c_{ij} is outside the scope of this paper, however, the following expression can be used to estimate it:

$$c_{ij} = c_{ij}^{buy} + c_{ij}^{adapt}$$

where c_{ij}^{buy} is the purchase cost, and c_{ij}^{adapt} is the adaptation cost. The adaptation cost takes into account the fact that, in order to integrate a software component into a system, the component must support the style of the interactions of the system's architecture to correctly work together with other components. If a COTS product has another style of interaction, developers have to introduce glueware to allow correct interactions.

Yakimovich et al. in [24] suggest a procedure for estimating the adaptation cost. They list some architectural styles and outline their features with respect to a set of architectural assumptions. They define a vector of variables, namely the interaction vector, where each variable represents a certain assumption. An interaction vector can be associated either to a single COTS or to a whole software architecture. To estimate the adaptation cost of a COTS they suggest to compare its interaction vector with the software architecture one.

Furthermore, c_{ij}^{adapt} could include the cost needed to handle mismatches between the functionalities offered by alternative COSTs and the functional requirements of the system. In fact, it may be necessary to perform a careful balancing between requirements and COTS features, as claimed in [2].

The purchase cost c_{ij}^{buy} is typically provided by the vendor of the COTS component.

The delivery time d_{ij} might be decomposed in the sum of the time needed to the vendor to deliver the component, and the adaptation time.

For sake of model formulation, in this paper we do not explicitly preserve the above decompositions of cost and delivery time parameters, although we implicitly take into account them in the example of Section 4.

The parameter s_i represents the average number of invocations of a component within the execution scenarios considered for the software architecture. Note that this value does not depend on the component instance, because we assume that the pattern of interactions within each scenario does not change by changing the component instance. This value is obtained by processing the execution scenarios that, in the CODER framework, are represented by UML Sequence Diagrams (see Section 3). The number of invocations is averaged overall the scenarios by using the probability of each scenario to be executed. The latter is part of the operational profile of the application.

The parameter μ_{ij} represents the probability for the instance j of component i to fail in one execution [20]. A rough upper bound $1/N_{nf}$ of μ_{ij} can be obtained upon observing the component being executed for a N_{nf} number of times with no failures. However, several empirical methods to estimate COTS failure rates can be found in [17].

In-House Component Model Parameters

The parameters that we define for an in-house developed instance $C_{ij} \in \bar{J}_i$ are:

- the unitary development cost $\bar{c}_{ij}$;
- the estimated development time t_{ij};

- the average time τ_{ij} required to perform a test case;
- the average number s_i of invocations;
- the probability p_{ij} that the instance is faulty;
- the testability $Testab_{ij}$.

The unitary cost $\bar{c}_{ij}$ is intended as the per-day cost of a software developer, that may depend on the skills and experience required to develop C_{ij}. Well-assessed cost/time models are available to estimate the first three parameters (e.g. COCOMO [4]).

The parameter p_{ij} is an intrinsic property of the instance that depends on its internal complexity. The more complex the internal dynamics of the component instance is, the higher is the probability that a bug has been introduced during its development. In [18] an expression for p_{ij} has been proposed as a function of the component instance internal reachability.

The definition of testability that we adopt in our approach is the one given in [21], that is:

$$Testab_{ij} = P(failure|prob.\ distribution\ \ of\ \ inputs) \tag{1}$$

Their definition of testability expresses the conditional probability that a single execution of a software fails on a test case following a certain input distribution. In [5] we suggest a procedure to estimate it.

Model Variables. In general, a "build-or-buy" decisional strategy can be described as a set of 0/1 variables defined as follows ($\forall i = 1 \ldots n$):

$$x_{ij} = \begin{cases} 1 \text{ if the } C_{ij} \text{ instance is chosen } (j \in \bar{J}_i \text{ or } j \in J_i) \\ 0 \text{ otherwise} \end{cases}$$

Obviously, if the i-th component has only $\bar{m} < m$ instances then the x_{ij}'s are defined for $1 \leq j \leq \bar{m}$.

For each component i, exactly one instance is either bought as COTS or in-house developed. The following equation represents this constraint:

$$\sum_{j \in J_i \cup \bar{J}_i} x_{ij} = 1, \qquad \forall i = 1 \ldots n \tag{2}$$

Finally, let N_{ij}^{tot} be an additional integer decision variable of the optimization model that represents the total number of tests performed on the in-house developed instance j of the i-th component ([2]).

Basing on the testability definition, we can assume that the number N_{ij}^{suc} of successful (i.e. failure-free) tests performed on the same component can be obtained as:

$$N_{ij}^{suc} = (1 - Testab_{ij})N_{ij}^{tot}, \qquad \forall i = 1 \ldots n, j \in \bar{J}_i \tag{3}$$

[2] The effect of testing on cost, reliability and delivery time of COTS products is instead assumed to be accounted in the COTS parameters.

Cost Objective Function (COF). The development cost of the in-house instance C_{ij} can be expressed as: $\bar{c}_{ij}(t_{ij} + \tau_{ij} N_{ij}^{tot})$. The objective function to be minimized, as the sum of the costs of all the component instances selected from the "build-or-buy" strategy, is given by:

$$COF = \sum_{i=1}^{n} \left(\sum_{j \in \bar{J}_i} \bar{c}_{ij}(t_{ij} + \tau_{ij} N_{ij}^{tot}) x_{ij} + \sum_{j \in J_i} c_{ij} x_{ij} \right) \tag{4}$$

Delivery Time Constraint (DT). A maximum threshold T has been given on the delivery time of the whole system. In case of a COTS product the delivery time is given by d_{ij}, whereas for an in-house developed instance C_{ij} the delivery time shall be expressed as $t_{ij} + \tau_{ij} N_{ij}^{tot}$. Therefore the following expression represents the delivery time DT_i of the component i:

$$DT_i = \sum_{j \in \bar{J}_i} (t_{ij} + \tau_{ij} N_{ij}^{tot}) x_{ij} + \sum_{j \in J_i} d_{ij} x_{ij} \tag{5}$$

Without loss of generality, we assume that sufficient manpower is available to independently develop in-house component instances. Therefore the delivery constraint can be reformulated as follows:

$$\max_{i=1 \ldots n} (DT_i) \leq T \tag{6}$$

which can be decomposed in the set of constraints $DT_1 \leq T, \ldots, DT_n \leq T$.

Reliability Constraint (REL). We consider systems that may incur only in crash failures, that are failures that (immediately and irreversibly) compromise the behaviour of the whole system ([3]).

A minimum threshold R has been given on the reliability on demand [20] of the whole system. The reliability of the whole system can be obtained as a function of the probability of failure on demand of its components, as we show in this section.

The probability of failure on demand $\mu_{ij}, j \in J_i$, for COTS components has been discussed in section 2.1.

The probability of failure on demand θ_{ij} of the in-house developed instance $C_{ij}, j \in \bar{J}_i$, can be formulated as follows:

$$\theta_{ij} = \frac{Testab_{ij} \cdot p_{ij}(1 - Testab_{ij})^{N_{ij}^{suc}}}{(1 - p_{ij}) + p_{ij}(1 - Testab_{ij})^{N_{ij}^{suc}}} \tag{7}$$

A proof of this formulation is given in [5].

Now we can write the average number of failures $fnum_i$ of the component i as follows:

$$fnum_i = \sum_{j \in \bar{J}_i} \theta_{ij} s_i x_{ij} + \sum_{j \in J_i} \mu_{ij} s_i x_{ij} \tag{8}$$

[3] Note that, although promising formulations of the component capability of propagating errors have been devised (see for example [1]), no closed form expression for system reliability embedding error propagation has yet been found.

In agreement with [11], the probability that no failure occurs during the execution of the i-th component is given by $\phi_i = e^{-fnum_i}$, which represents the probability of no failures occurring in a Poisson distribution with parameter $fnum_i$.

Therefore the probability of a failure-free execution of the system is given by $\prod_{i=1}^{n} \phi_i$. The reliability constraint is then given by:

$$\prod_{i=1}^{n} \phi_i \geq R \tag{9}$$

Model Summary. The objective function (4), under the main constraints (2), (6) and (9), plus the obvious integrality and non-negativity constraints on the model variables, represent the optimization model adopted within the CODER framework.

The model solution provides the optimal "build-or-buy" strategy for component selection, as well as the number of tests to be performed on each in-house developed component. The solution guarantees a system reliability on demand over the threshold R, a system delivery time under the threshold T while minimizing the whole system cost. The applied reliability model is a light-weighted one, as we work in favor of model solvability. However, it can be replaced by a profound reliability growth model from literature [7] to increase the result accuracy. This can be done without essentially changing the overall model structure, with the side effect of increasing complexity. In [5] we report the mathematical formulation of the whole model.

With regard to the accuracy of the model, there are some input parameters (e.g. the probability of failure on demand, the cost) that may be characterized by a not negligible uncertainty (i.e. only a range for the costs may be available [14]). The propagation of this uncertainty should be analyzed, but it is outside the scope of this paper. However, several methods to perform this type of analysis can be found, e.g. it has been done in [8] for a reliability model.

3 The CODER Framework

In Figure 1 the CODER framework is shown within its working environment.

The input to the framework is an UML model constituted by: (i) a Component Diagram representing the software architecture, (ii) a set of Sequence Diagrams representing the possible execution scenarios.

CODER accepts UML models in XMI format [25]. In theory, any tool exporting diagrams to XMI can be used to generate input models for CODER. In practice this is not the case because XMI exporting formats may sensibly differ from each other. For this paper example (in Section 4) we have used ArgoUML [26] to build and export UML diagrams.

The CODER framework is made of two components, which are a model builder and a model solver.

The model builder first allows users to annotate the UML diagrams with additional data that represent the optimization model parameters (see Section 2), such as failure probabilities of software components. Then it transforms the annotated model into an optimization model in the format accepted from the solver.

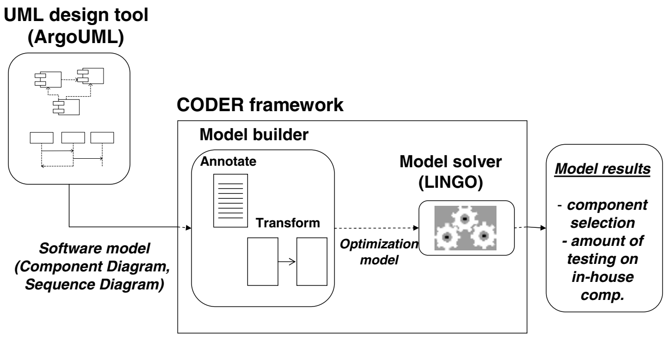


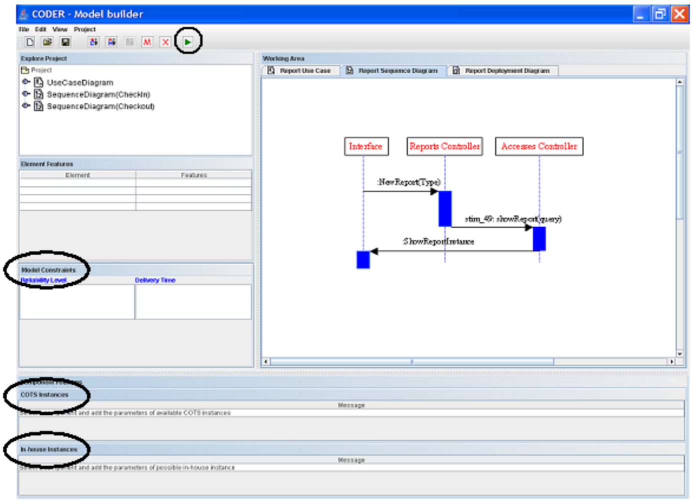
Fig. 1. The CODER framework and its environment

The model solver processes the optimization model received from the builder and produces the results, that consist in the selection of components and the amount of testing to perform on in-house components.

The optimization model solver that we have adopted in CODER is LINGO [27]. The integration between the model builder and the model solver has been achieved as follows. LINGO makes use of a callable Dynamic Link Library (DLL) to provide a way to bundle its functionalities into a custom application. In particular, the DLL gives the ability to run a command script containing an optimization model and a series of commands that allows to gather data, to populate and solve the model. The integration of data between the calling application (i.e. the model builder in our case) and the solver can be obtained by means of the *Pointer* functions in the data section of the script. These functions act as a direct memory links and permit direct and fast memory transfers of data in and out of the solver memory area.

As a result of this integration, LINGO can be directly run from the main interface of the model builder, as shown in Figure 2. The main interface can be partitioned in 3 areas: (i) the working area (upper right side of Figure 2), where the imported UML diagrams are shown, and where components and lifelines can be selected for annotations; (ii) the annotation area, where the model parameters related to software components can be entered (lower side of Figure 2); (iii) the model constraint area, where values of model constraint bounds can be assigned (upper left side of Figure 2). The four ellipses of Figure 2 highlight, respectively, from the top to the bottom of the figure: the button to run the model solver LINGO, the title of the area where constraint bounds can be entered and the titles of areas where COTS and in-house component parameters can be entered.

Summing up, CODER allows to specify ranges for model parameters and sets of alternative optimization models can be automatically generated (by sampling parameters in the given ranges) and solved. The output of CODER, for each model, is a suggested selection of available components and the suggested amount of testing to perform on each in-house developed component. In the next section, we apply the model to an example.



and add the parameters of available COTS instances

and add the parameters of possible in-house instance

Fig. 2. A screenshot of the CODER model builder

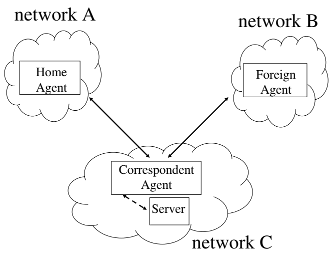


Fig. 3. The example architecture

4 Using CODER in the Development of a Mobile Application

We have considered an application that allows the mobility of a user without loosing its network connection based on Mobile IP [19].

Figure 3 shows the architecture of the application. A user of the network A can exchange data with users of the network C through a server located in C. A user of network A can also move to network B and continue to interact with a user of network C without generating a new connection. Four software components are deployed: *Home Agent* (running on network A), *Foreign Agent* (running on network B), *Correspond Agent* and *Server* (running on network C).

The scenario that we consider can be described as follows: interactions between users in A and users in C change only when a user moves from A to B; the effect of this move is that *Foreign Agent* provides the user's new address to *Home Agent*; as soon as users in C attempt to interact with the moving user through her/his old address in

Table 1. First Configuration : parameters for COTS products

	Component name	COTS alternatives	Cost c_{ij}	Average delivery time d_{ij}	Average no. of invocations s_i	Prob. of fail. on demand μ_{ij}
C_0	*Correspond Agent*	C_{01}	12	4	200	0.0005
		C_{02}	14	3		0.00015
		C_{03}	15	3		0.0001
C_1	*Server*	C_{11}	6	4	40	0.0003
		C_{12}	12	3		0.0001
C_2	*Home Agent*	C_{21}	12	2	80	0.00015
C_3	*Foreign Agent*	C_{31}	7	4	25	0.0002
		C_{32}	10	3		0.00015
		C_{33}	8	7		0.00015

Table 2. First Configuration : parameters for in-house development of components

	Component name	Development Time t_{i0}	Testing Time τ_{i0}	Unitary development cost $\bar{c}_{i0}$	Average no. of invocations s_i	Faulty Probability p_{i0}	Testability $Testab_{i0}$
C_0	*Correspond Agent*	10	0.007	1	200	0.03	0.00001
C_1	Server	5	0.007	1	40	0.01	0.001
C_2	*Home Agent*	6	0.007	1	80	0.04	0.001
C_3	*Foreign Agent*	5	0.007	1	25	0.05	0.002

A, *Home Agent* provides the user's new address to *Correspond Agent* so that, without interruption, users in C switch their interactions towards network B [19].

We show the support that the CODER framework can provide to select components during the development of this application. We apply our approach on two different configurations. In order to keep our model as simple as possible, in both configurations we assume that only one in-house instance for each component can be developed.

The number of COTS instances does not change across configurations, but each configuration is based on a different set of component parameters. We have solved the optimization model in both configurations for a set of values of reliability and delivery time bounds.

4.1 First Configuration

Table 1 shows the parameter values for the COTS available instances, likewise Table 2 does for in-house developed ones, where $\bar{J}_i = \{0\}(i = 0, ..., 3)$, $J_0 = \{1, 2, 3\}$, $J_1 = \{1, 2\}$, $J_2 = \{1\}$ and $J_3 = \{1, 2, 3\}$.

The third column of Table 1 lists, for each component, the set of COTS alternatives available at the time of system development. For each alternative: the buying cost c_{ij} (in KiloEuros, KE) is given in the fourth column, the average delivery time d_{ij} (in days) is given in the fifth column, the average number of invocations of the component in the system s_i is given in the sixth column, finally the probability of failure on demand μ_{ij} is given in the seventh column.

For each component in Table 2: the average development time t_{i0} (in days) is given in the third column and the average time required to perform a single test τ_{i0} (in days) is given in the fourth column, the unitary development cost $\bar{c}_{i0}$ (in KE per day) is given in the fifth column, the average number of invocations s_i is given in the sixth column,

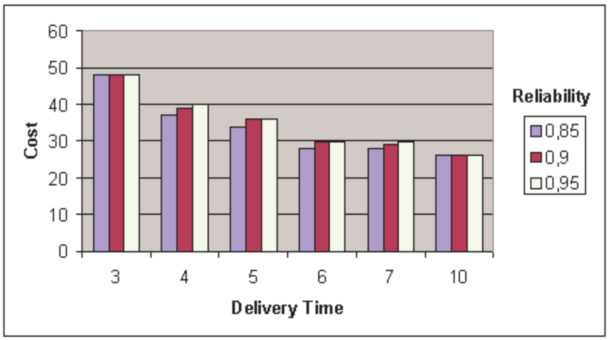


Fig. 4. Model solutions for first configuration

the probability p_{i0} that the component instance is faulty at the execution time is given in the seventh column, and finally the component testability $Testab_{i0}$ is given in the last column.

Note that each component can be in-house built. This configuration is characterized by the fact that the in-house instance of each component is less reliable than all the COTS available components, but it is less expensive than all the latter ones.

In Figure 4 we report the results obtained from solving the optimization model for multiple values of bounds T and R. Each bar represents the minimum cost for a given value of the delivery time bound T and a given value of the reliability bound R. The former spans from 3 to 10 whereas the latter from 0.85 to 0.95.

As expected, for the same value of the reliability bound R, the total cost of the application decreases while increasing the delivery time bound T (i.e. more time to achieve the same goal). On the other hand, for the same value of T the total cost almost always decreases while decreasing the reliability bound R (i.e. less reliable application required) and, in two cases, it does not increase.

With regard to the component selection: for $T < 10$, the model solution proposes different combinations of COTS and in-house instances almost always without test on the latter ones; for $T = 10$, the model proposes, for all R values, the same solution made of all in-house instances without test. In [5] we report the solution vectors for each pair of bounds (T,R).

4.2 Second Configuration

Similarly to the first configuration, Table 3 shows the parameter values for the COTS available components, whereas Table 4 shows the ones for the in-house instances.

Again note that each component can be in-house built. The component parameters in this configuration have been set to induce a certain amount of testing on in-house instances. In particular: the in-house instance of C_0 is less reliable, but earlier available and less expensive than all the available COTS instances for this component; the C_1 and C_2 in-house instances are less reliable than all the corresponding COTS available instances, but are less expensive than these last ones; the in-house instance of C_3 is

Table 3. Second Configuration : parameters for COTS products

	Component name	COTS alternatives	Cost c_{ij}	Average delivery time d_{ij}	Average no. of invocations s_i	Prob. of fail. on demand μ_{ij}
C_0	*Correspond Agent*	C_{01}	12	4	200	0.00015
		C_{02}	14	3		0.00015
		C_{03}	15	3		0.00001
C_1	*Server*	C_{11}	18	4	40	0.0001
		C_{12}	18	3		0.00003
C_2	*Home Agent*	C_{21}	15	2	80	0.00001
C_3	*Foreign Agent*	C_{31}	9	4	25	0.0002
		C_{32}	14	3		0.00015
		C_{33}	9	7		0.00002

Table 4. Second Configuration: parameters for in-house development of components

	Component name	Development Time t_{i0}	Testing Time τ_{i0}	Unitary development cost $\bar{c}_{i0}$	Average no. of invocations s_i	Faulty Probability p_{i0}	Testability $Testab_{i0}$
C_0	*Correspond Agent*	1	0.007	1	200	0.08	0.008
C_1	Server	10	0.007	1	40	0.08	0.009
C_2	*Home Agent*	10	0.007	1	80	0.08	0.007
C_3	*Foreign Agent*	6	0.007	1	25	0.05	0.004

as reliable as (but more expensive than) the first COTS instance, whereas all the other COTS instances are more reliable than it.

In Figure 5 we report again the results obtained from solving the optimization model for multiple values of bounds T and R. Here the former spans from 3 to 15 whereas the latter from 0.90 to 0.98. In [5] we report the solution vectors for each pair of bounds (T,R).

Similarly to the first configuration, for the same value of the reliability bound R, the total cost of the application decreases while increasing the delivery time bound T. On the other hand, for the same value of T the total cost decreases while decreasing the reliability bound R.

As shown in [5], the component selection for this configuration is more various. While T increases, the model tends to select in-house components because they are cheaper than the available COTS instances. This phenomenon can be observed even for low values of T. The total cost decreases while T increases because ever more in-house instances can be embedded into the solution vectors. The in-house instances remain cheaper than the corresponding COTS instances even in cases where a non negligible amount of testing is necessary to make them more reliable with respect to the available COTS.

5 Related Work

The correlation between costs and non-functional attributes of software systems has always been of high interest in the software development community. After a phase of experimental assessment, in the last years new methodologies and tools have been introduced to systematically model and evaluate issues related to this aspect from the architectural phase.

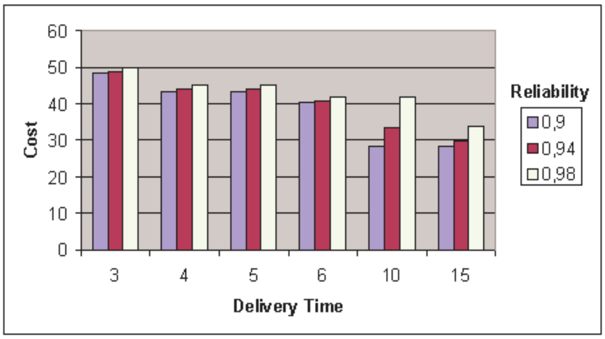


Fig. 5. Model solutions for second configuration

The Architecture Tradeoff Analysis Method (ATAM) [13] provides to software developers a framework to reason about the software tradeoffs at the architectural level. The Attribute-based Architectural Styles (ABAS) [16], used within ATAM, help software architects to reason (quantitatively and qualitatively) about the quality attributes and the stimulus/response characteristics of the system. ATAM/ABAS framework is based on roughly approximated cost-characteristic curves, usually elicited from experience, that show how costs will behave with respect to each architectural decision. Our context differs from ATAM/ABAS because it is model-based, as opposed to experience-based, and the model we propose focuses on component selection decisions.

A significant breakthrough in this area has been the Costs Benefit Analysis Method (CBAM) [14]. CBAM, laying on the artifacts produced from ATAM, estimates costs, (short-term and long-term) benefits and uncertainty of every potentially problematic architectural design decision devised from ATAM. The estimates come out from information collected from stakeholders in a well assessed elicitation process. Architectural decisions are represented in a space whose dimensions are costs, benefits and (some measure of) uncertainty. The graphical representation of decisions is an excellent mean to support the developers' choices. Architectural strategies (ASs) typically have effects on several quality attributes (QAs). In order to evaluate the benefits of ASs on the whole software system, CBAM framework proposes that stakeholders assign contributions of ASs to QAs, and quality attribute score to QAs. The benefit of an AS is then computed as the sum of its contributions weighted on the QAs quality attribute scores. CBAM framework, however, deals neither with the elicitation of such contributions and scores nor with the assessment of ASs implementation costs. Actually, cost estimation often has to take into account some critical time-to-market goals such as delivery times and shared use of resources.

Although CBAM is a very promising technique to support software developers giving priorities among architectural decisions on the basis of their costs and benefits, it requires to stakeholders to estimate a large number of scores, contributions and costs by resorting to qualitative judgements based of their own expertise. In this context an analytical approach taking into account all architectural alternatives and tradeoffs among qualitative attributes is extremely suitable. A key issue is to capture the relationships among costs and quality attributes, as well as across different quality attributes.

An optimization model may play the role of *decision support* in the early development phases, where the decisions are usually based on stakeholders' estimates. Later on, it can be an actual *decision-making* tool, when the software architecture and the bounds on the quality attributes have been devised, and implementation choices (such as resource allocation and amount of testing) may heavily affect costs and quality of the system. A classical approach for cost management is the portfolio-optimization, based on a knapsack-like integer linear programming model [15]. The models and techniques that we refer to in the remainder of this section follow this approach.

Optimization techniques appeared first in the area of software development in [10], where a variant of the 0-1 knapsack model is introduced to select the set of software requirements that yields the maximum value while minimizing costs. The concept of value is kept quite general and may be interpreted as an implementation priority. The knapsack model is first used to maximize the total value of requirements under budget constraints, thereafter to minimize the total cost without loosing requirement value.

The same authors in [12] introduce an interesting generalization of the model assumptions. The idea is that each COTS has a generic quality attribute, the objective function is the system quality as the weighted sum of COTS qualities, and the maximization is budget-constrained.

In the reliability domain, an interesting formulation of a cost minimization model has been given in [11]. Again 0-1 variables allow to select alternative COTS components, under a constraint on the failure rate of the whole system. The latter quantity is modeled as a combination of the failure rates of single components, their execution times, and a rough measure of the system workload. This is the closest model formulation to the one that we propose here.

In [22,23] the reliability constraints also cope with hardware failures, but the non-linear complexities of the models impose heuristic solutions.

An extensive optimization analysis of the tradeoff between costs and reliability of component-based software systems has been presented in [9]. A reliability constrained cost minimization problem is formulated, where the decision variables represent the component failure intensities. Three different types of cost functions (i.e., linear, logarithmic exponential, inverse power) have been considered to represent the dependency of the component cost on the component failure intensity, that is the cost to attain a certain failure intensity. An exponential function has been used to model the system reliability as a combination of component failure intensities, operational profile (i.e. probability of component invocation) and time to execute the invoked service. The goal of this type of analysis is quite different from the one of this paper. The model in [9] works after the components have been chosen, as its solution provides insights about the failure intensities that the (selected) components have to attain to minimize the system cost.

The formulation of our model that we proposed in section 2 is close to the one in [11] with an additional constraint on the system delivery time. However, none of the existing approaches, supports "build-or-buy" decisions.

The following major aspects characterize the novelty of our approach:

- From an automation viewpoint, CODER is (at the best of our knowledge) the first thorough framework that supports a process of component selection based on cost, reliability and delivery time factors.
- CODER is not tied to any particular architectural style or to any particular component-based development process. Values of cost and reliability of in-house developed components can be based on parameters of the development process (e.g. a component cost may depend on a measure of developer skills).
- From a modeling viewpoint, we introduce decision variables that represent the amount of testing performed on each in-house component. The cost objective function, the reliability and delivery time constraints depend on these variables, therefore our model solution not only provides the optimal combination of COTS/in-house components, but also suggests the amount of testing to be performed on in-house components in order to attain the required reliability.

6 Conclusions

We have presented a framework supporting "build-or-buy" decisions in component selection based on cost, reliability and delivery time factors. The framework not only helps to select the best assembly of COTS components but also indicates the components that can be conveniently developed in-house. For the latter ones, the amount of testing to perform is also provided.

The integration of an UML tool (like ArgoUML), a model builder, and a model solver (like LINGO) has been quite easy to achieve due to XML interchange formats on one side, and to the Dynamic Link Library of LINGO on the other side. The CODER framework has been conceived to be easily usable from developers, and it indeed shows two crucial usability properties: transparency and automation. The software is annotated without modifying the original UML model, but producing a new annotated model, thus attaining transparency with respect to software modeling activities. Besides, the model building and solving is a completely automated tool supported process.

The results that we have obtained on the example shown in this paper provides evidence of the viability of such approach to the component selection. The components selected from the framework evidently constitute an optimal set under the existing constraints. It would be hard to obtain the same results without tool and modeling support. In addition, the tool also provides the amount of testing to perform, thus addressing the classical problem of: "How many tests are enough?" [18].

We are investigating the possibility of embedding in CODER other types of optimization models that may allow to minimize costs under different non-functional constraints (e.g. under security constraints). In general, these types of models are well suited to study the tradeoffs between different non-functional attributes, that are usually very hard to model and study in current (distributed, mobile) software systems. Furthermore, we intend to enhance CODER by introducing the multi-objective optimization [3] to provide the configuration of components that minimizes, for example, both the cost of construction of the system and its probability of failure on demand.

References

1. W. Abdelmoez et al., “Error Propagation in Software Architectures”, *Proc. of METRICS*, 2004.
2. Alves, C. and Finkelstein, A. “Challenges in COTS decision-making: a goal-driven requirements engineering perspective”, *Proc. of SEKE 2002*, 789-794, 2002.
3. Censor, Y., “Pareto Optimality in Multiobjective Problems”,*Appl. Math. Optimiz.*, vol. 4, 41-59, 1977.
4. Boehm, B. “Software Engineering Economics”, *Prentice-Hall*, 1981.
5. Cortellessa, V., Marinelli, F., Potena, P. “Appendix of the paper: Automated selection of software components based on cost/reliability tradeoff”, *Technical Report*, Dip. Informatica, Università de L’Aquila, http://www.di.univaq.it/cortelle/docs/TECHNICALREPORT.pdf.
6. Gokhale, S.S., Wong, W.E, Horgan, J.R., Trivedi, K.S. “An analytical approach to architecture-based software performance and reliability prediction”, *Performance Evaluation*, vol. 58 (2004), 391-412.
7. Goseva-Popstojanova, K. and Trivedi, K.S. “Architecture based-approach to reliability assessment of software systems” , *Performance Evaluation*, vol. 45 (2001),179-204.
8. Goseva-Popstojanova, K. and Kamavaram, S. “Uncertainty Analysis of Software Reliability Based on Method of Moments” , *FastAbstract ISSRE 2002*.
9. Helander, M.E., Zhao, M., Ohlsson, N. “Planning Models for Software Reliability and Cost”, *IEEE Trans. in Software Engineering*, vol. 24, no. 6, June 1998.
10. Jung H.W. “Optimizing Value and Cost in Requirement Analysis”, *IEEE Software*, July/August 1998, 74-78.
11. Jung H.W. et al. “Selecting Optimal COTS Products Considering Cost and Failure Rate”, *Fast Abstracts of ISSRE*, 1999.
12. Jung H.W. and Choi B. “Optimization Models for Quality and Cost of Modular Software Systems”, *European Journal of Operational Research*, 112 (1999), 613-619.
13. Kazman, R. et al. “Experience with Performing Architecture Tradeoff Analysis”, *Proc. of ICSE99*, 1999.
14. Kazman, R. et al. “Quantifying the Costs and Benefits of Architectural Decisions”, *Proc. of ICSE01*, 2001.
15. Kellerer, H. et al. “Knapsack Problems”, *Springer-Verlag*, 2004.
16. Klein, M. and Kazman, R. “Attribute-based Architectural Styles”, *CMU-SEI-99-TR-22, SEI, CMU*, 1999.
17. Li, J. et al. “An Empirical Study of Variations in COTS-based Software Development Processes in Norwegian IT Industry”, *Proc. of METRICS’04*, 2004.
18. Menzies, T. and Cukic, B. “How Many Tests are Enough”, *Handbook of Software Engineering and Knowledge Engineering*, Volume 2, 2001.
19. Perkins, C. “IP Mobility Support”, *RCF 3344 in IETF*, 2002.
20. Trivedi K., “Probability and Statistics with Reliability, Queuing, and Computer Science Applications”, J. Wiley and S., 2001.
21. Voas, J. M. and Miller, K. W. “Software testability: The new verification”, *IEEE Software*, pages 17-28, May 1995.
22. Wattanapongsakorn N. “Reliability Optimization for Software Systems With Multiple Applications”, *Fast Abstracts of ISSRE01*, 2001.
23. Wattanapongsakorn N. and Levitan S. “Reliability Optimization Models for Fault-Tolerant Distributed Systems”, *Proc. of Annual Reliability and Maintainability Symposium*, 2001.
24. Yakimovich, D., Bieman, J. M., Basili, V. R.“Software architecture classification for estimating the cost of COTS integration.”, *Proc. of ICSE*, pages 296-302, June 15-21, 1999.
25. OMG, “MOF 2.0/XMI Mapping Specification”, v2.1 formal/05-09-01.
26. argouml.tigris.org
27. www.lindo.com