Resolving Platform Specific Models at Runtime Using an MDE-Based Trading Approach

Javier Criado, Luis Iribarne, and Nicolás Padilla

Applied Computing Group, University of Almería, Spain
{javi.criado,luis.iribarne,npadilla}@ual.es

Abstract. Dynamic service composition provides versatility and flexibility features for those component-based software systems which need to self-adapt themselves at runtime. For this purpose, services must be located in repositories from which they will be selected to compose the final software architecture. Modern component-based software engineering and model-driven engineering techniques are being used in this field to design the repositories and the other elements (such as component specifications), and to implement the processes which manage them at runtime. In this article, we present an approach for the runtime generation of Platform Specific Models (PSMs) from abstract definitions contained in their corresponding Platform Independent Models (PIMs). The process of generating the PSM models is inspired by the selection processes of Commercial Off-The-Shelf (COTS) components, but incorporating a heuristic for ranking the architectural configurations. This trading process has been applied in the domain of component-based graphical user interfaces that need to be reconfigured at runtime.

Keywords: MDE, Trading, Components, Adaptation, Web Services.

1 Introduction

One of the traditional goals of software engineering (SE) has been the need to develop systems by assembling independent modules. The *Component-Based Software Development* (CBSD) is one of the traditional disciplines of SE that is characterized by describing, developing and using components based on techniques for building open, distributed systems such as distributed information systems. This approach makes software engineering faces to new challenges and problems, since this kind of systems requires a "bottom-up" development instead to a traditional "top-down" development. The CBSD process begins with the definition of the architecture, which sets out the specifications of the components at an *abstract* level. In the "bottom-up" perspective, most of the architectural requirements may be covered by other components stored in repositories. These repositories contain "concrete" specifications of components which are required by automated searching processes trying to locate those exact component specifications that meet with those abstract restrictions of the software architecture.

In addition, it is also important to take into account some automated processes which consider dependencies between the components of the architecture, and

Y.T. Demey and H. Panetto (Eds.): OTM 2013 Workshops, LNCS 8186, pp. 274–283, 2013.

generate possible combinations of the components. These combinations could provide a partial or complete solution of the architecture to be rebuilt at runtime. In this paper we propose a solution for systems that use a component-based architecture that can change at runtime, and therefore requiring an automatic reconfiguration. Examples of systems with this type of architecture may be seen for instance in smart home applications [1], robotics [2], communication network infrastructures [3], user interfaces [4], etc.

On the other hand, *Model-Driven Engineering* (MDE) has provided many solutions for software development. In the particular domain of *component-based software systems*, the use of MDE techniques can facilitate design and development of architectures, for example, for defining their structure, the behavior of their components and their relationships, their interaction or functional and non-functional properties [5].

Our research work intends to find a solution to the problem of adapting software systems at runtime, but focuses only on component-based systems. In our methodology, the life cycle for developing component-based architectures is structured on *abstract architectural model*, which corresponds to a PIM (*Platform Independent Model*) level of MDE and represents the architecture in terms of what sort of components it contains and their relationships; and *concrete architectural model*, which corresponds to the PSM (*Platform Specific Model*) level and describes what concrete components comply with the abstract definition of the architecture. The proposal presented in this paper concerns the adaptation process of concrete architectural models that solves a platform specific model fulfilling the component architectural requirements of the system at runtime. Given an starting abstract architecture, the concrete architectural models are realized by a semantical trading process [6], calculating the configurations of concrete components that meet the abstract definitions best, which provides the possibility of generating different software architectures based on the same abstract definition, for example, so it can be executed on different platforms.

The rest of the paper is organized as follows: Section 2 presents the domain for resolving our PSM models. Then, Section 3 explains the trading process. Section 4 shows a case study and describes validation performed. Section 5 reviews the related work and finally, conclusions and future works are given in Section 6.

2 Resolving Platform Specific Models

As outlined in the introduction, our research work focuses on the adaptation of component-based systems. We have chosen the domain of graphical user interfaces as part of research projects of the Spanish Ministry and the Andalusian Government that require adaptation of user interfaces at runtime. Our interest in this domain is due to the trend toward *Social Semantic Web* or *Web 3.0*.

In this context, we understand that it could be useful to have component-based user interfaces that can adapt their functionality depending on the circumstances. Under this scenario, the user interfaces have to be defined as architectural models in which each component represents a user interface

component. We therefore assume that there is a repository containing such user interface components that could be fed by components developed by third-parties, as long as they comply with our specifications.

These components are represented in two levels of abstraction: (1) *abstract level*, corresponding to the PIM level of MDE, and (2) *concrete level*, which corresponds to the PSM level of MDE. Both types of component models will be constructed conform to the same metamodel, which is shown in Figure 1. This metamodel represents the main parts of a COTS component. Each component has a `Functional` part, and optionally, a `NonFunctional` part. The first contains the functional component properties, which are distinguished between provided and required. `Provided` properties are related with the services offered by the component, and `Required` properties represent the services necessary for the component to work properly.

The `NonFunctional` part contains non-functional properties of the component. For each non-functional property, it is possible to define the name, the value, the priority, if is observable (which means the component has mechanisms to offer its value) and if is editable (indicating the component has mechanisms to edit its value). The difference between abstract and concrete models lies in the latter have the property `uri` which indicates the location of the final component, while the `property_priority` attribute is only defined in the first. The fulfillment of this condition as well as verification that the model is correctly defined, is checked using OCL rules [7].

Moreover, for the remainder of the article, we will also use $\mathcal{R}$ to refer to the provided interfaces of a component and $\overline{\mathcal{R}}$ to represent its required interfaces. Similary, $\{MapInfo\}$ denotes an interface providing the $MapInfo$ service, and $\{\overline{UserInfo}\}$ represents an interface requiring the $UserInfo$ service.

From these component specifications, we build our abstract and concrete architectural models. They will be used in an adaptation process that is executed on both levels of abstraction. On the abstract level, M2M transformation processes are executed to adapt the abstract architectural models to the changes in

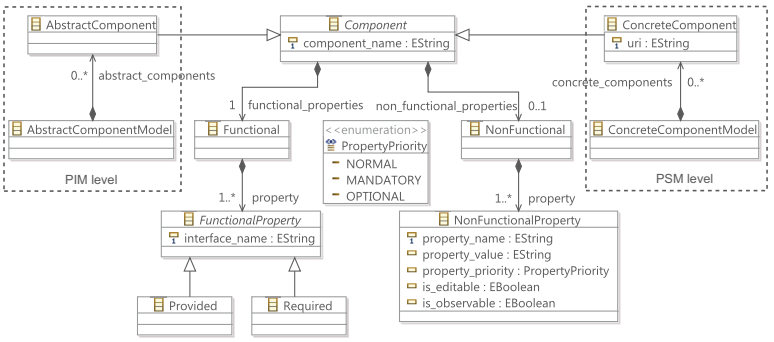

Fig. 1. Component Metamodel

the context [8]. Furthermore, the concrete architectural models are realized by a trading process, which is the goal of this paper, calculating the configurations of concrete components that best fulfill the abstract definitions (Figure 2).

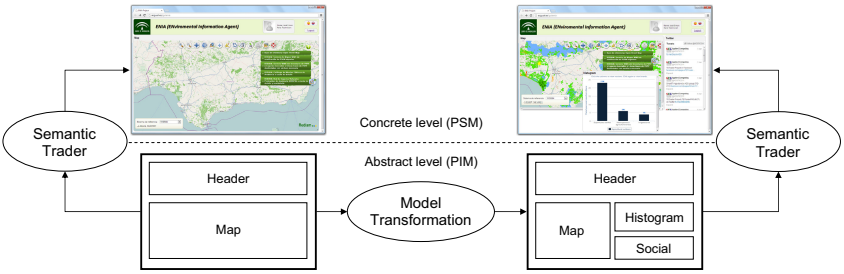

Fig. 2. Adaptation process within our GUI domain

Our trading process uses the information about the meaning given by combining the functional and non-functional properties of the components. Therefore, this mediation service, responsible for managing and selecting the best configuration of COTS-type concrete components, has been named *Semantic Trader service* and will be explained in detail in the next section.

3 Defining the Trading Process

Although there is a lot of work on the selection and evaluation of COTS components [9], our process for the selection of components is based on the proposal of [6]. In this approach, and also in the work presented in this paper, a trading service is developed for the management of the components and the configurations of the system architectures.

The main purpose of this service is to calculate the best possible configurations of available concrete components from the abstract definition of an architecture. These configurations are searched in order to fulfill the requirements described by the abstract definition. From the resolved configurations, the service generates a concrete architectural model as output. The steps of the trading process are shown in Figure 3 that will be explained above.

#1. Selection of candidates: As stated in the article from which this work arises [10], the selection of candidate components (CC) consists of filtering from the repository of concrete component specifications (CCR) those ones that could be part of the architecture. For this purpose, the process checks which are the concrete components of the repository that have at least one common provided interface with the definition given in the abstract architectural model. The filtering does not take into account the required interfaces of each concrete component, and their suitability to be part or not of the resulting configuration will be checked in the following steps.

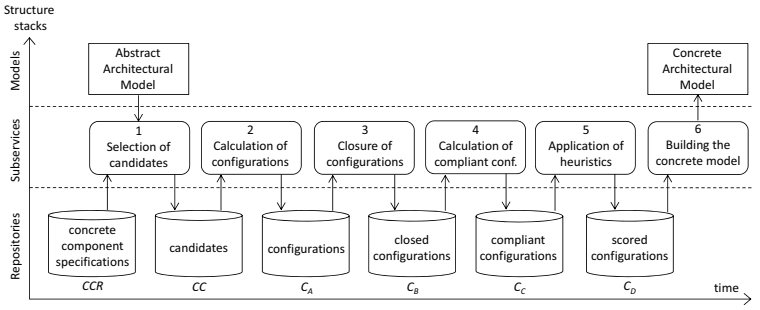

Fig. 3. Semantic Trader service

#2. Calculation of configurations: The next step is to perform the calculation of all the possible configurations of concrete components that could be constructed from the candidates. This process is inspired in the *backtracking* algorithm described in [10]. The aforementioned algorithm was developed for searching and matching components at design time. However, in our case, the adaptation of architectural models should be carried out at runtime, and the time it takes to run the *backtracking* algorithm does not meet our requirements. Furthermore, in our proposal, the configurations must satisfy a one-to-one matching between the components of the abstract architecture and the components of the concrete architecture. In this regard, the *backtracking* algorithm explores and generates a large number of configurations that will not be taken into account for the resulting concrete architecture. Consequently, and based on this premise, the algorithm for the calculation of the configurations has been implemented using a recursive algorithm whose stop condition is that a concrete component had been found for each abstract component existing in the input abstract architectural model. The pseudocode of the algorithm is shown in Table 1.

#3. Closure of configurations: The third step executed in our *Semantic Trader service* is the closure of all the possible configurations previously calculated. This phase is necessary because some of the configurations could generate solutions that are not complete, in the case that any of the concrete components requires some additional component so that the abstract definition runs correctly. The pseudocode of the algorithm is shown in Table 1. In this algorithm, all the candidates that are found for each configuration are checked. If any of these candidates has any additional required interface to the abstract architectural model, that configuration will be considered as unclosed. In contrast, if none of the candidates meet this condition, the configuration will be added to the repository of closed configurations C_B.

#4. Calculation of compliant configurations: As has been described in our approach the configurations must satisfy an one-to-one matching between the components of the abstract architectural model and the components of the concrete architectural model. This matching is pursued in order to facilitate adaptation and synchronization operations of our system, so that the actions

Table 1. Calculation and closure of the configurations

```
function configurations(i, sol, C_A)
  if i ≥ card(AAM) then
    CC(i) = getCandidates(AAM_i)
    for j = 1 → card(CC(i))
      sol := sol ∪ {CC(i)_j.R ∩ AAM_i.R}
      configurations(i + 1, sol, C_A)
      sol := sol − {CC(i)_j.R ∩ AAM_i.R}
    endfor
  else
    C_A := C_A ∪ {sol}
  endif
endfunction
```

```
function closeConfigurations(C_A, C_B)
  for i = 1 → card(C_A)
    closed := true, conf := (C_A)_i
    for j = 1 → card(conf)
      candidate := conf_j
      if (candidate.R̄ ∩ AAM.R̄) <> ∅ then
        closed := false
      endif endfor
  endfor
  if closed == true then C_B := C_B ∪ {conf}
  endif
endfunction
```

performed on the components of concrete and abstract levels have a direct correspondence with the components of the other level. Therefore, the behavior of this subservice is to discard those configurations that do not comply with the internal structure of the abstract architecture, *i.e.*, those configurations that have divisions or groupings of services that differ from those established in the abstract architectural model. Thus, it is generated the repository of compliant configurations C_C.

#5. Application of heuristics: Once the configurations have been filtered by choosing only those that match with the internal structure of the architecture, it is necessary to define a process that evaluates each configuration to establish a criteria for the selection of the best one. With this aim, a ranking process based on heuristics has been defined. This heuristics process checks the matching between each of the abstract components and their corresponding concrete components that resolve them. The total score of the configuration will be the sum of the partial scores for each component. The heuristics rules could be separated in two categories, each one referring to the main parts of the component definition that has been described in Figure 1. The scores are shown in Table 2. The defined heuristics is based on the analysis performed in which we study what are the most important criteria for the configurations to meet the system requirements. It is also based on a "backwards" analysis from the configurations of concrete components that we know that are better (that resolve an abstract definition

Table 2. Scoring the configurations

Functional Properties	**Score**
Direct matching between the $\mathcal{R}$ of the abstract component and the $\mathcal{R}$ of the concrete component	+0.5
Direct matching between the $\overline{\mathcal{R}}$ (if the condition of direct matching between provided is met)	+0.3
Each additional provided interface regarding those defined in the abstract component	−0.1
Each additional required interface regarding those defined in the abstract component	−0.2
Each additional binding needed in the architecture	−0.3
Non-Functional Properties	**Score**
Each 'mandatory' property that matches the abstract definition	+0.1
Each 'normal' property that matches the abstract definition	+0.05
Each 'optional' property that matches the abstract definition	+0.02
Each property defined as 'observable'	+0.01
Each property defined as 'editable'	+0.01
Each additional property regarding those defined in the abstract component'	+0.01

better) for our purpose. As a result of applying this process, the repository of scored configurations, C_D, is generated.

#6. Building the concrete model: Once the repository of scored configurations, C_D, has been calculated, the Semantic Trader service builds the concrete architectural model offering it as its output. This model is constructed from the configuration that has a higher score, that is, it is the configuration that best fulfills the abstract definition.

4 Case Study: Implementation and Validation

With the aim of illustrate the presented approach, a case study is shown below. Let us suppose that the current software architecture managed by the system has an abstract definition at the PIM level (*i.e.*, an abstract architectural model) containing the following four abstract components: *Map*, *Histogram*, *Header* and *Social*. This architecture could represent the right GUI of the Figure 2. From this abstract architectural model, the *Semantic Trader service* will be called at runtime for resolving the PSM model (*i.e.*, the concrete architectural model). Looking into the repository of concrete component specifications CCR, the concrete candidates components CC shown in Table 3 are selected. Due to space reasons, the table only shows information related to the functional properties, since they are involved in the most of trader subservices.

Once the candidates have been selected, the remaining subservices are called. First, all the possible configurations are calculated. Second, these configurations are filtered with the described closure process. Then, the closed configurations that comply with the architecture are selected. These configurations are scored by applying the defined heuristics and, therefore, the ordered set of configurations shown in the Table 3 is generated.

Table 3. Abstract components, concrete candidates, and resulting configurations

<table>
<tr><th colspan="15">Abstract components</th></tr>
<tr><td colspan="15">$Map = \{MapInfo\}, Histogram = \{HistInfo, \overline{MapInfo}\}$,
$Header = \{UserInfo\}, Social = \{SocialInfo, \overline{UserInfo}\}$</td></tr>
<tr><th colspan="15">Concrete candidate components (CC)</th></tr>
<tr><td colspan="7">(1)$MapCom1 = \{MapInfo\}$
(2)$MapCom2 = \{MapInfo, Feedback\}$
(3)$MapCom3 = \{MapInfo, \overline{OGCServices}\}$
(4)$MapCom4 = \{MapInfo, HistIfo\}$
(5)$MapCom5 = \{MapInfo, \overline{UserInfo}\}$
(6)$HistCom1 = \{HistInfo, Palette, \overline{MapInfo}\}$</td><td colspan="8">(7)$HistCom2 = \{HistInfo, \overline{MapInfo}\}$
(8)$HeaderCom1 = \{UserInfo, Language, Logout\}$
(9)$HeaderCom2 = \{UserInfo, Logout, \overline{Weather}\}$
(10)$HeaderCom3 = \{UserInfo, Logout\}$
(11)$SocialCom1 = \{SocialInfo\}$
(12)$SocialCom2 = \{SocialInfo, \overline{UserInfo}\}$</td></tr>
<tr><th colspan="15">Resulting configurations (C_D)</th></tr>
<tr><th>#</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>Score</th><th>Configuration</th></tr>
<tr><td>1</td><td>M</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>H</td><td>-</td><td>-</td><td>U</td><td>-</td><td>S</td><td>2.41</td><td>$CC_1, CC_7, CC_{10} - \{L\}, CC_{12}$</td></tr>
<tr><td>2</td><td>M</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>H</td><td>U</td><td>-</td><td>-</td><td>-</td><td>S</td><td>2.31</td><td>$CC_1, CC_7, CC_8 - \{L, I\}, CC_{12}$</td></tr>
<tr><td>3</td><td>M</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>H</td><td>-</td><td>-</td><td>U</td><td>S</td><td>-</td><td>2.11</td><td>$CC_1, CC_7, CC_{10} - \{L\}, CC_{11}$</td></tr>
<tr><td>4</td><td>M</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td><td>H</td><td>U</td><td>-</td><td>-</td><td>S</td><td>-</td><td>2.01</td><td>$CC_1, CC_7, CC_8 - \{L, I\}, CC_{11}$</td></tr>
<tr><td>5</td><td>M</td><td>-</td><td>-</td><td>-</td><td>-</td><td>H</td><td>-</td><td>-</td><td>-</td><td>U</td><td>-</td><td>S</td><td>1.51</td><td>$CC_1, CC_6 - \{P\}, CC_{10} - \{L\}, CC_{12}$</td></tr>
<tr><td>6</td><td>-</td><td>-</td><td>-</td><td>-</td><td>M</td><td>-</td><td>H</td><td>-</td><td>-</td><td>U</td><td>-</td><td>S</td><td>1.5</td><td>$CC_5, CC_7, CC_{10} - \{L\}, CC_{12}$</td></tr>
<tr><td>7</td><td>-</td><td>-</td><td>-</td><td>-</td><td>M</td><td>-</td><td>H</td><td>U</td><td>-</td><td>-</td><td>-</td><td>S</td><td>1.4</td><td>$CC_5, CC_7, CC_8 - \{L, I\}, CC_{12}$</td></tr>
<tr><td>...</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>...</td><td>...</td></tr>
<tr><td>43</td><td>-</td><td>M</td><td>-</td><td>-</td><td>-</td><td>H</td><td>-</td><td>U</td><td>-</td><td>-</td><td>S</td><td>-</td><td>0.1</td><td>$CC_2 - \{F\}, CC_6 - \{P\}, CC_8 - \{L, I\}, CC_{11}$</td></tr>
<tr><td>44</td><td>-</td><td>-</td><td>M</td><td>-</td><td>H</td><td>-</td><td>U</td><td>-</td><td>-</td><td>S</td><td>-</td><td></td><td>0.1</td><td>$CC_4 - \{M\}, CC_6 - \{P\}, CC_8 - \{L, I\}, CC_{11}$</td></tr>
<tr><td colspan="15">Key → CC_i: i-th candidate, M: $MapInfo$, H: $HistInfo$, U: $UserInfo$, S: $SocialInfo$,
L: $Logout$, I: $Language$, P: $Palette$, F: $Feedback$, P: $Palette$</td></tr>
</table>

As a result, the best scored configuration is the one containing the candidates components CC_1, CC_7, CC_{10} and CC_{12}. The concrete component CC_{10} has an additional provided interface (*Logout*) to the abstract definition, so it is necessary to perform a hiding operation of such interface. The given score is due to: (a) the direct matching between provided and required interfaces of CC_1, CC_7 and CC_{12} components ($(0.5+0.3)*3 = 2.4$), (b) the additional provided interface of CC_{10} component (-0.1) and (c) a 'mandatory' non-functional property that matches the abstract definition and which is also 'observable' ($0.1 + 0.01 = 0.11$), making a total of 2.41. This configuration is used to build the concrete architectural model that will be generated as output of the process.

For the **validation** of our proposal, each of the subservices which are part of the Semantic Web service (see Figure 3) have been implemented and deployed on a web server. These services are described through the corresponding WSDL (*Web Service Description Language*) file to be used in combination with SOAP. Regarding execution times for the case study, total process time is about 54 *ms* which is an acceptable time for resolving the platform specific model at runtime.

5 Related Work

Even though our proposal covers different aspects on adaptation of component-based software systems, this paper is focused on resolving platform specific models of this kind of systems by using a trading service. Therefore, the reviewed papers describe approaches related to the selection and evaluation of COTS components [9]. There are many works dealing with the issue of COTS evaluation. In [11] authors proposed the *Off-The-Shelf Option* (OTSO) method that relies on *Analytic Hierarchy Process* (AHP) [12] with the aim of stablish an evaluation criteria for the components based on their features and the system requirements. In addition, the AHP method is used in a wide variety of COTS evaluation and selection approaches [13–15]. The problem is that this methodology only analyses unidirectional relationships among the different levels that are involved in the evaluation process. However, components are not always independent from each other [16], but often require other components or interact in a way that should be taken into account.

The authors in [17] present a model for the evaluation of the COTS combining TOPSIS (*Technique for Order Performance by Similarity to Idea Solutuion*) and ANP (*Analytic Network Process*) techniques. The describe framework is practical for ranking COTS in terms of their performance regarding multiple criteria. Otherwise, the work in [18] proposed a methodology of six steps based on a quality model that decompose the component characteristics for their evaluation, and it describes a selection process with the DesCOTS system. In contrast, our component evaluation proposal is more simple and only takes into account the component specification in terms of functional and non-functional properties.

In [6] the COTStrader approach which inspired this paper is presented. It describes a trading service for the management of COTS components and how it can be used for build component-based systems at design time. However, our trading service is intended to generate at runtime PSM models from their PIM

specification, due to which, the algorithms have been modified. An approach for the selection of COTS based on cots and system requirements is described in [19]. In this paper, the authors present DEER, a framework for COTS selection in the requirement phase of a component-based development process.

None of the processes described in these papers occur during the execution of the system, nor aim to adapt the structure of the component-based architecture at runtime. Instead, our proposal resolves at runtime the configurations of concrete components to achieve the adaptation of architectural models.

6 Conclusions and Future Work

In this paper we have presented an approach for resolving Platform Specific Models at runtime. These PSM models fulfill the specifications of an abstract architectural model, which represents the software architecture at the PIM level. Our architectures are made up of COTS-type components, so we were inspired by the proposals of selection and search for these components to generate our processes and our models. As a consequence, a MDE-based Trading service has been developed for realizing the concrete architectural models from the abstract architectures, calculating the configurations of concrete components that best meet the abstract definitions.

Our trading process provides the possibility of generating different software architectures based on the same abstract definition, for example, so it can be executed on different platforms. It uses the information about the meaning given by combining the functional and non-functional properties of the components for the calculation and the ranking of the configurations, so our trading process has been named *Semantic Trader*. It involves six stages to accomplish the requirements of the abstract model: selection of candidate components, calculation of configurations, closure of configurations, calculation of compliant configurations, application of heuristics, and calculation of the resulting concrete model. Each process has been implemented as a web service to be called from the main Semantic Trader service. This main service has been tested and validated on a case study in which acceptable execution times have been generated for resolving the platform specific models at runtime.

The developed trading service assumes that at least one configuration of concrete components that meets the abstract definition will be resolved. This limitation is intended to be improved in future work from the resolution of mismatches [20], and we want to provide some alternative resolution mechanism if there is a situation that can not calculate a valid configuration. Moreover, we also intend to implement searching algorithms (for calculating the configurations) that will be based on this heuristics to perform tree pruning for not explore possible configurations that do not lead to the optimal solution.

Acknowledgments. This work is supported by the Spanish MINECO under grant of the project TIN2010-15588, and the Spanish MECP under a FPU grant (AP2010-3259), and also by the Junta Andalucía under grant of the project P10-TIC-6114.

References

1. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. Computer (IEEE Computer Society) 42(10), 37–43 (2009)
2. Edwards, G., Garcia, J., Tajalli, H., Popescu, D., Medvidovic, N., Sukhatme, G., Petrus, B.: Architecture-driven self-adaptation and self-management in robotics systems. In: SEAMS 2009, pp. 142–151 (2009)
3. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer (IEEE Computer Society) 37(10), 46–54 (2004)
4. Grundy, J., Hosking, J.: Developing adaptable user interfaces for component-based systems. Interacting with Computers 14(3), 175–194 (2002)
5. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.: A Classification Framework for Software Component Models. IEEE Transactions on Software Engineering 37(5), 593–615 (2011)
6. Iribarne, L., Troya, J.M., Vallecillo, A.: A Trading Service for COTS Components. The Computer Journal 47(3), 342–357 (2004)
7. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A Definitive Guide. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 58–90. Springer, Heidelberg (2012)
8. Rodríguez-Gracia, D., Criado, J., Iribarne, L., Padilla, N., Vicente-Chicote, C.: Runtime Adaptation of Architectural Models: An Approach for Adapting User Interfaces. In: Abelló, A., Bellatreche, L., Benatallah, B. (eds.) MEDI 2012. LNCS, vol. 7602, pp. 16–30. Springer, Heidelberg (2012)
9. Mohamed, A., Ruhe, G., Eberlein, A.: COTS selection: past, present, and future. In: Engineering of Computer-Based Systems (ECBS 2007), pp. 103–114 (2007)
10. Iribarne, L., Troya, J.M., Vallecillo, A.: Selecting Software Components with Multiple Interfaces. In: IEEE 28th Euromicro Conf., pp. 26–32 (2002)
11. Kontio, J., Caldiera, G., Basili, V.R.: Defining factors, goals and criteria for reusable component evaluation. In: CASCON 1996, p. 21 (1996)
12. Saaty, T.L.: How to make a decision: the analytic hierarchy process. European Journal of Operational Research 48(1), 9–26 (1990)
13. Morera, D.: COTS evaluation using desmet methodology & analytic hierarchy process (AHP). In: Oivo, M., Komi-Sirviö, S. (eds.) PROFES 2002. LNCS, vol. 2559, pp. 485–493. Springer, Heidelberg (2002)
14. Finnie, G.R., Wittig, G.E., Petkov, D.I.: Prioritizing software development productivity factors using the analytic hierarchy process. Journal of Systems and Software 22(2), 129–139 (1993)
15. Min, H.: Selection of software: the analytic hierarchy process. International Journal of Physical Distribution & Logistics Management 22(1), 42–52 (1992)
16. Carney, D.J., Wallnau, K.C.: A basis for evaluation of commercial software. Information and Software Technology 40(14), 851–860 (1998)
17. Shyur, H.J.: COTS evaluation using modified TOPSIS and ANP. Applied Mathematics and Computation 177(1), 251–259 (2006)
18. Grau, G., Carvallo, J.P., Franch, X., Quer, C.: DesCOTS: A Software System for Selecting COTS Components. In: IEEE 30th Euromicro Conf., pp. 118–126 (2004)
19. Cortellessa, V., Crnkovic, I., Marinelli, F., Potena, P.: Experimenting the Automated Selection of COTS Components Based on Cost and System Requirements. J. UCS 14(8), 1228–1255 (2008)
20. Mohamed, A., Ruhe, G., Eberlein, A.: Mismatch handling for COTS selection: a case study. J. Softw. Maint. Evol.-Res. Pract. 23(3), 145–178 (2011)