Hindawi Publishing Corporation
Journal of Computer Networks and Communications
Volume 2012, Article ID 963728, 11 pages
doi:10.1155/2012/963728

Research Article

Context-Aware Adaptation of Component-Based Systems: An Active Repository Approach

Sindolfo Miranda Filho,[1] Julio Melo,[1] Luiz Eduardo Leite,[2] and Guido Lemos[3]

[1] *Centro de Tecnologia, Universidade Federal do Rio Grande do Norte, 59078-900 Natal, RN, Brazil*
[2] *Escola de Ciência e Tecnologia, Universidade Federal do Rio Grande do Norte, 59078-900 Natal, RN, Brazil*
[3] *Departamento de Informática, Universidade Federal da Paraíba, 58051-900 Paraíba, PB, Brazil*

Correspondence should be addressed to Sindolfo Miranda Filho, sindolfo@lavid.ufpb.br

Received 17 March 2012; Revised 17 June 2012; Accepted 18 June 2012

Academic Editor: Goreti Marreiros

Context-aware systems are able to monitor and automatically adapt their operation accordingly to the execution context in which they are introduced. Component-based software engineering (CBSE) focuses on the development and reuse of self-contained software assets in order to achieve better productivity and quality. In order to store and retrieve components, CBSE employs component repository systems to provide components to the system developers. This paper presents an active component repository that is able to receive the current configuration from the context-aware system and compute the components and the new architecture that better fit the given context. Since the repository has a wide knowledge of available components, it can better decide which configuration is more suitable to the running system. The repository applies Fuzzy logic algorithm to evaluate the adequacy level of the components and GRASP algorithm to mount the new system architecture. In order to verify the feasibility of our approach, we use a digital TV middleware case study to achieve experimental results.

1. Introduction

Pervasive computing is becoming increasingly popular, as introduced by Weiser [1], the term pervasive refers to the seamless integration of computer devices and software into the everyday life. Context-awareness adaptation is an important field of the pervasive computing area. Context-aware systems are able to monitor and automatically adapt their operation accordingly to the execution context in which they are introduced.

Component-based software engineering (CBSE) [2, 3] focuses on the development and reuse of self-contained software assets in order to achieve better productivity [4] and quality as software systems are composed by previously developed components used (and tested) in other projects. In this approach, the software system is composed by self-contained components that explicitly declare their provided functionalities (provided interfaces), required functionalities (required interfaces), and also their execution context requirements.

In addition to the advantages described previously, component-based software (CBS) may present other features like component update, functionality enhancement, and adaptability.

If a CBS needs to be updated, only the specific component implementing the updated feature needs to be updated. Functionality enhancement is also facilitated since new components with new functionalities can be added and dynamically loaded into the system. Finally, adaptability can also be accomplished by configuring and/or replacing a component by another that better fit the current execution environment.

One of the challenges faced by CBSE is the task of discovering suitable reusable assets that fulfill the requirements of the particular software system under development. In order to address such problem, several component repository systems were proposed [5–9]. Accordingly to [10], a component repository is a software system that provides functionalities to locate, select, and retrieve software components.

Ye [11] proposed a repository that provides active mechanisms of information delivery. Ye's active repository is capable of providing information to the users by monitoring their development activities without the need to receive explicit queries from them. These systems execute in background inside an integrated development environment (IDE) monitoring the user activity and suggesting possible software components to be used in the current development context.

Both the traditional and the active repositories provide functionalities only during the development phase of the software system. Once the system is deployed, the repositories are of no use. This way, in order to promote the runtime adaptation of context-aware component-based systems, this paper presents an active repository that actively provides new software components that are more adapted to the context of the adaptable system. Since the repository has a wide knowledge of the available components, it can better decide which configurations of components are more suitable to the running system. In the proposed approach, the context aware system informs to the repository its current configuration and its context information and the repository is able to compute the components and new architecture that better fit the given context. In this way, the component repository commonly adopted in component-based software systems is expanded to provide components, not only during the development stage but also context aware components during the operational stage of the system life cycle.

This paper is organized as follows. Section 2 presents the repository architecture and the mechanism used by our approach in order to evaluate the adequacy level of a component to the given context. Section 3 presents the heuristic algorithms used to compute a new architecture to the given system. These algorithms use the component adequacy level described in Section 2. Section 4 presents our digital TV middleware case study. Section 5 presents some experimental results achieved with our digital TV middleware implementation. The related works are discussed in Section 6. Finally, Section 7 presents our concluding remarks and future directions.

2. System Architecture

To allow the construction of adaptive component-based systems, that will be called now and forth client systems, we have created an architecture named REATIVO that will handle the client system context changes and generate new system architectures over the time through component reconfiguration. Figure 1 shows REATIVO's general architecture.

This architecture is not intended to be restricted to a specific component model. However, to use REATIVO's services, the client system needs to be developed using a component model that allows at least the simple reconfiguration commands (add, remove, replace, connect, or disconnect a component in the system).

In addition, the client system must implement some modules that allow REATIVO retrieve the system current configuration state and context representation.

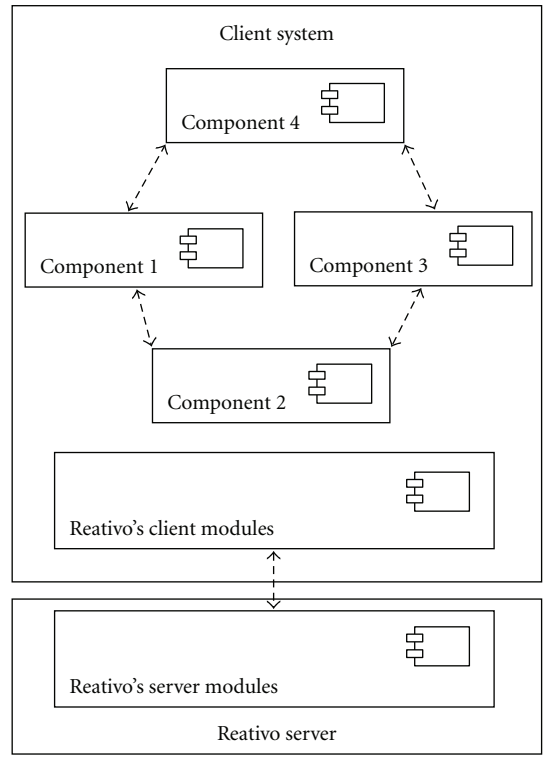

Figure 1: General architecture.

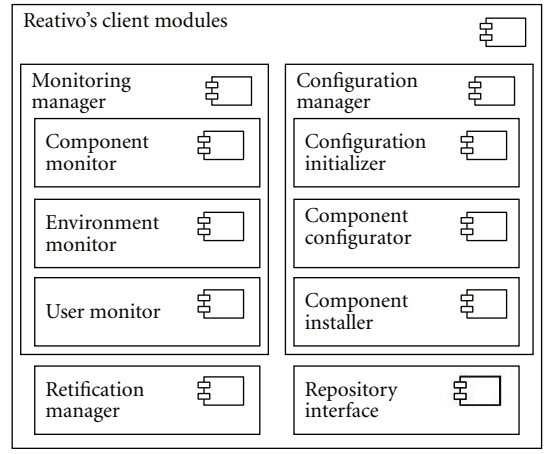

Figure 2: Client components.

REATIVO's modules can be divided into two disjoint groups, the Client components and the Server components. Figures 2 and 3 show these components, respectively.
As shown in Figure 2, the Client components are as follows.

(i) Monitoring manager: responsible to monitor the current execution context in terms of the state and component architecture (Component Monitor), the execution environment (Environment Monitor) and the users (User Monitor).

(ii) Reification Manager: responsible to combine information collected by the Monitoring Manager in a

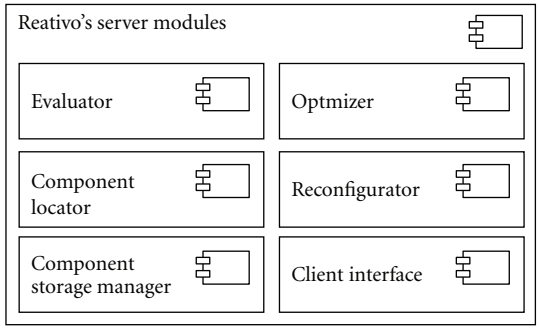

Figure 3: Server components.

meta-representation that will be used to generate a new optimized configuration to the system.

(iii) Configuration Manager: responsible for the instantiation of the initial system architecture (Configuration Initializer) and for instantiate new architectures sent by the REATIVO server, new architectures will be instantiated by adding, removing, or replacing components (Component Installer) or interconnecting existing ones (Component Configurator).

(iv) Repository Interface: provides communication with REATIVO server. This module is able to provide the client system an abstraction of the server location in order to enable the use of the distributed services.

The REATIVO server provides a set of services for receive client system context and provide reconfiguration commands if needed. The server components are listed below.

(i) Evaluator: evaluates quantitatively the adequacy level of a given component accordingly to a given client context.

(ii) Storage Manager: stores components into the repository.

(iii) Component locator: locates components in the distributed repository.

(iv) Optimizer: Implements heuristic algorithms to solve the best configuration problem (See Section 3. Generating optimized component architectures).

(v) Reconfigurator: reconfigures the client system by sending the reconfiguration commands: add, remove, replace, connect, or disconnect components.

(vi) Client Interface: Implements the communication between the clients and the REATIVO server.

2.1. Repository Behavior. When the client system starts up, the Configuration Initializer reads the initial component architecture and initializes it to put the system in full execution.

Once in execution, the system variables will be monitored by REATIVO client components. The variables monitored are chosen in accordance to an ontology developed focusing

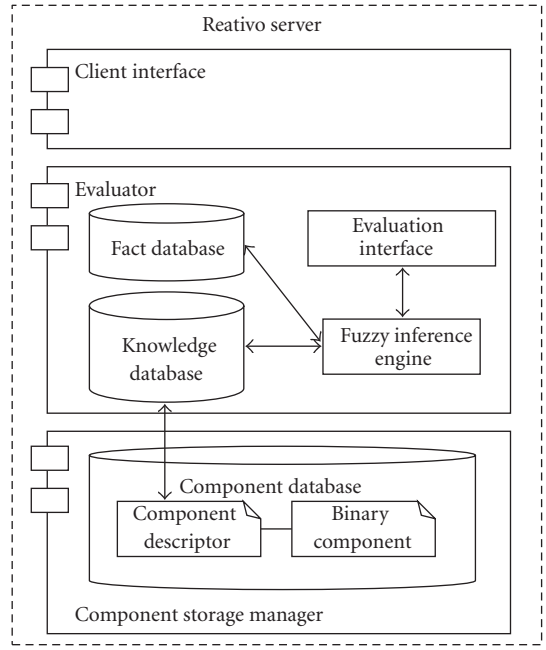

Figure 4: Component evaluation architecture.

the specific application domain. When a context modification is detected, the client system generates a meta-representation through the Reification Manager and uses the Repository Interface to pass it to REATIVO's server, which will activate the Optimizer in order to start the process to generate a new architecture.

The Optimizer will execute a heuristic algorithm to compute the best component architecture based on the received context. This heuristic algorithm will be described in Section 3. Generating optimized component architectures and relies on the adequacy level of each component.

When a better architecture is found, REATIVO's server sends the reconfiguration commands necessary to change the client system by using the Client Interface component.

2.2. Component Evaluation—Calculating the Component's Adequacy Level. To be able to evaluate a given component for a given execution context, REATIVO uses the Evaluator component (see Figure 4). The implementation of the Evaluator uses Fuzzy Logic [12] to perform a quantitative evaluation of the adequacy level of a component to a given execution context. In this context, the adequacy level consists of values ranging from 0% to 100% of adequacy.

When the Evaluator is queried for a component evaluation it requests to the Storage Manager a component description that contains a set of rules in the format: if "condition" then "component is adequate." By using this modeling one could thought that it is better to use first order logic to decide whenever a component is adequate or not, however some measurements in the execution context may not be like the rule: if "received signal has noise" then "component is adequate." In this case, the signal could have much, moderate or just a little amount of noise, so using first

order logic is not a good choice since it could not deal well with inaccurate values.

By using Fuzzy logic in the previous example, we could conclude that the component is "much adequate," "moderate adequate" or "little adequate" by using the generalized modus ponens [12] what explain our choice to the Fuzzy approach.

The Evaluator stores the component description in a Knowledge Database and the execution context sent by the client system in the Fact Base. This way, the Fact Base contains information determined by the Domain Ontology that constrains which are the variables and their value range, in addition to the following parameters to be used by the fuzzy inference engine.

(i) the pertinence function to the fuzzy set which the input variables may belong;

(ii) math operators that implement the fuzzy logic operations;

(iii) defuzzification function to be utilized;

(iv) Fuzzy rules to be inserted in the Knowledge Base;

(v) Fuzzy sets to which the output variables of the Inference Engine may belong.

In this context, the ontology is used to define which concepts are relevant to the domain (these concepts will be monitored by the client system) and the value range of each of these concepts. On the other hand, the component adequacy level is given by the Fuzzy inference engine, which receives as input the values of the domain ontology variables collected by the client system.

Each component in the repository provides its own set of parameters (listed previously) defined in the Component Descriptor and delivered with the component in the Storage Manager. This way the component developer has flexibility to define how the Evaluator module will behave when evaluating the components.

This concept may cause a problem where some components are evaluated differently for the same criteria. For instance a company could specify that its component has an evaluation 50 for determined context and another company could specify that its component, less adequate in fact, has evaluation 60 for the same context. In order to solve such problem, the component description must be specified according to a well defined standard; however, this problem is out of our scope in this work. To simplify the issue, our work assumes that the components are implemented by the same manufacturer or the components are defined by a consortium of manufacturers that uses the same evaluation criteria.

An important aspect that can be noted here is that REATIVO's repository approach is domain independent. If the repository needs to be used with a different client application, within a different application domain, a new domain ontology needs to be developed to the specific domain and the components registered in the repository needs to define their parameters to the Fuzzy inference engine compatible with the new ontology, in other words, needs to take into account the values of the variables defined in the new ontology. No changes need to be done in the repository server logic.

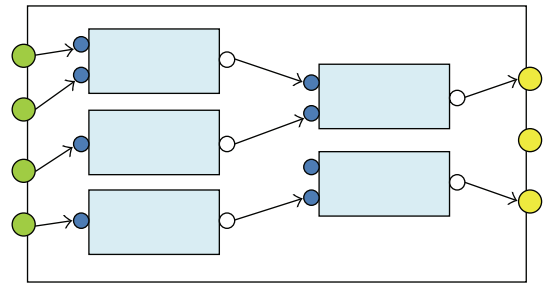

Figure 5: Internal diagram of a general component based system.

3. Generating Optimized Component Architectures

REATIVO stores a set of components that needs to be connected together to form a component-based system. Depending on the number of components stored, the number of possible combinations can be too high.

This way finding the most adequate component architecture to a given execution context will be a nontrivial job. This task is executed by the Optimizer component in our architecture; given an execution context sent by the client, the system will try to find a better suitable architecture by solving an instance of the component-oriented system optimization problem (COSOP).

3.1. Component-Oriented System Optimization Problem [13]. Any component-based system has at least one Provided Interface, that is used by external entities to access system's services and could has zero or more Required Interfaces that are used by the system to access external services.

In our case both types of interfaces are implemented by system's internal interconnected components. The interconnection is done by using the component's interfaces that could also be Required or Provided.

In Figure 5, the green and yellow circles represent system's provided and required interfaces. Dark blue and white circles represent components provided and required interfaces, respectively.

Each required interface represents a service needed by a component in order to work properly so it needs to be connected to a provided interface of another component to acquire the required service. This generates a compatibility property between the two interfaces that needs to be satisfied while connecting both.

This way a component-oriented system consists in interconnecting a set of components in such a way that the following rules are satisfied.

(i) All provided interfaces of the system are associated to a corresponding provided interface of an internal component.

(ii) All internal components' required interfaces are connected to a compatible component or associated to a corresponding system's required interface.

(iii) All provided interfaces could be connected to 0 or M_r required interfaces, where M_r is the maximum number of connections supported by the provided interface p.

Taking as an assumption that each component in REATIVO's repository can be evaluated by the Evaluator module, the cost of a component is defined as follows:

Cost evaluation function:

$$\text{cost}(c) = 100 - \text{evaluation}(c). \tag{1}$$

Therefore, using (1), the total cost of the system, represented by the sum of each system's component cost, must be minimized in order to achieve the best possible system.

3.2. Math Model. A mathematical model was developed in order to minimize the total system cost satisfying some constraints in order to maintain the properties of the component-based system. Based on the constraints to be satisfied and the minimization equation, we have the following optimization problem:

Equation to be minimized:

$$\text{minimize: } \sum_{i \in C} V_i \times X_i \tag{2}$$

within the following constraints.

Constraints of the minimization problem: Table 1 explains the complete set of variables considered in the optimization problem.

$$Y_{ikjl} - D_{kl} \leq 0 \quad i \in C,\ k \in A_i,\ j \in C,\ l \in P_j, \tag{3a}$$

$$\left(\sum_{j \in C} \sum_{l \in P_i} Y_{ikjl} \times D_{kl} \right) - X_i = 0 \quad i \in C,\ k \in A_i, \tag{3b}$$

$$\left(\sum_{j \in C} \sum_{l \in P_i} Y_{ikjl} \times D_{kl} \right) - M_{jl} \times X_j \leq 0 \quad j \in C,\ l \in P_j, \tag{3c}$$

$$X_i \in \{0, 1\}, \quad X_0 = 1 \quad i \in C, \tag{3d}$$

$$Y_{ikjl} \in \{0, 1\} \quad i \in C,\ k \in A_i,\ j \in C,\ l \in P_j. \tag{3e}$$

In this model, the objective function (2) has to be minimized in order to achieve the optimal component set to a given execution context.

The constraint (3a) means that only compatible interfaces could be connected, this way if a required interface k of a component i is connected to a provided interface l of a component j ($Y_{ikjl} = 1$) and the interfaces are not compatible ($D_{kl} = 0$), the constraint would be disrespected.

Table 1: Mathematic model.

Indexes	
i,j	Component indexes
k	Required interface index
L	Provided interface index
Parameters	
C	Set of components in the repository, C_0 is the system itself
A_i	Set of required interfaces of the component i
P_i	Set of the provided interfaces of the component i
M_{il}	Maximum connections supported by the interface l of the component i
V_i	Cost when using component i
D_{kl}	1 if the required interface k is compatible with the provided interface l
Decision variables	
X_i	1 if the component i was used in the current configuration; 0 otherwise
Y_{ikjl}	1 if the required interface k of the component i is connected in the provided interface l of component j; 0 otherwise

Restriction (3b) states that if a component was used in the solution, all its required interfaces must be connected to exactly one compatible provided interface. If one component i was used in the solution ($X_i = 1$), all its required interfaces k must be connected to exactly one compatible provided interface l of a component j, in this case, the equation results in 0.

However, (3c) indicates that the provided interfaces of a component j could only be used if it was used in the solution. In addition, the maximum number of connections in the provided interface l of component j has to be less than M_{jl}. If a provided interface l of a component j is used in the solution ($X_j = 1$), M_{jl} connections can be done between the component j and other components interfaces k that are compatible ($D_{kl} = 1$). This way, (3c) will result in 0 if the maximum number of connection is used or less than 0 if the number of connections established is less than M_{jl}.

3.3. Problem Complexity. The COSOP problem has a trivial solution if all components required interfaces are compatible with all other components provided interfaces, however, this is not the case for even the simplest system.

A better approach could be associate weights to a given interface connection. Compatible optimal interfaces have cost of 0, while incompatible interfaces have a high cost, other less suitable interfaces have intermediate costs, so the problem would be reduced to "interconnect a set of nodes (provided and required interfaces) through intermediate nodes (software components), minimizing the cost of interconnection." This problem was already solved and presented as the Stainer Tree Star [14, 15].

As the Stainer Tree Star problem is proved to be NP-Complete and our problem could be reduced to it,

our problem can also be considered NP-Complete by the reduction proof.

3.4. Applying the GRASP Heuristic. As an NP-Complete problem, finding an optimal solution to a COSOP instance is not a trivial task. We decided to use the Greedy Randomized Adaptive Search Procedure (GRASP) [16] heuristic to find approximate solutions making the Optimizer component faster and reliable in terms of a real system.

The GRASP approach consists in an iterative algorithm where each iteration is composed of two stages: the build stage and a local search stage. The algorithm's final answer is the best solution among those found along successive iterations.

In the COSOP, the GRASP heuristic will consider C as the set of all components stored in the repository and M as the components that were assembled in the current solution. Two more sets are defined, A and P, where A is the set of the system and components' required interfaces in M that were not connected to provided interfaces and P the set of provided interfaces of the system and components' present in M.

In the build stage, it will be constructed a graph where the nodes represent the components and the edges represent the interconnections between their interfaces.

The evaluation cost and constraints to each solution are the same defined in (2) and (3a)–(3e), this way the algorithm will run until it finds a solution where all required interfaces of the components in M are connected to provided interfaces. In other words, the A set is empty.

3.5. Grasp Build Stage. This stage is initiated with an empty solution S in the graph. In each iteration, a required interface "a" from the set A and all components from C that has a provided interface compatible with "a" are chosen. The components are put in a Candidate List (CL) and then a Restricted Candidate List (RCL) is generated randomly using some of the best candidates in CL. In RLC will be components that satisfy (4) where $g(c)$ is the cost function given in (1).

Candidate evaluation:

$$g(c) \leq g_{\min} + \alpha(g_{\max} - g_{\min}). \quad (4)$$

After building the RCL, a component is chosen randomly from the set and the interface "a" that had been chosen in the beginning is connected to it. The Build Stage pseudocode is showed in Algorithm 1.

3.6. Local Search. In the local search stage, the neighborhood solutions of that one found in the build stage are generated using two perturbation operations: the SWAP operation, that consists in changing one component in the solution, and the DROP operation, that consists in removing components from the solution.

In both cases, using the SWAP or DROP operations implies in rebuilding a reliable solution by using the remaining components, however, those operations could improve the actual solution that is replaced by the better one in that case.

4. FlexTV Middleware Case Study

In this section, we analyse our case study, used in order to validate the REATIVO's approach. Current Digital TV (DTV) systems adopted around the world define middleware services in order to enable the TV receiver to execute TV application sent by the broadcasters in addition to the simple audio and video exhibition.

A DTV middleware basically standardize the application lifecycle and the APIs provided to the interactive applications running on top of it. In general, these middleware must deal with multiple context variations like different hardware platforms, support for broadcast applications, different signal quality and QoS requirements, and so forth. This way, a DTV middleware forms a relevant case study for the work proposed in this paper.

In this case study, we use a reference implementation of the Brazilian Digital TV procedural middleware called FlexTV. Our implementation follows a component-based architecture defining a set of loosed coupled components. Figure 6 presents a high level view of the FlexTV architecture showing the components and the execution environment.

The reception of low level streams is handled by the dark blue components. These components process the streams of audio, video and data, sending the video and audio to the *Media Processor* and the data streams to the *Data Processor*.

The yellow components are responsible to present media and user interface elements to the users and to capture the user interaction. For example, a key pressed in the user's remote control.

The applications that will execute over the middleware can use the *Application Communication* component to communicate with each other and the *Interaction Channel* component provides network access. The *Profile Manager* and *Persistence* are responsible for data storage. They store user preferences and other diverse data, respectively. The *Conditional Access* component handles access to restricted data and application security, authenticating applications that require privileged access to some system resources/functionalities. Finally, the *Application Manager* handles the load, configuration and execution of applications designed to run and use the middleware services.

The *Middleware Management* layer consists on the implementation of the REATIVO's client components described in Section 2.

As described in Section 2, the context variables are collected based on domain ontology. We developed a Digital TV ontology using the OWL language and the Protégé [17] tool. This Ontology contains 62 classes, 400 properties, and 33 relationships. The ontology classes are separated into three groups: classes related to user concepts, those related to the platform, and those related to the environment.

The user concepts are used to characterize the current users of the DTV system. The platform concepts are used to identify the current state of the DTV systems in terms of CPU

```
//A - list of system's access interfaces that are not connected
//P - List of system's provider interfaces
//C - List with the components that are in the repository
S = {} //solution graph empty in the beginning
while (A is not empty) do
{
        a = first(A)   //removes the first element from A
        if (exists (c) in P where c has provider interfaces
                that is compatible with a){
        {
                connect a to its compatible interface in c
                continue
        }
        LC = {c in C where c has provider interfaces compatible with a)
        g_min = min { g(c) of LC }
        g_max = max{ g(c) of LC }
        LCR = { all c in LC where g(c) <= g_min + alpha * (g_max - g_min) }
        select a random member v in LCR
        S = S + {v}
        add the provider interfaces from v to P
        add the access interfaces from v to A
}
return S
```

Algorithm 1: The build stage.

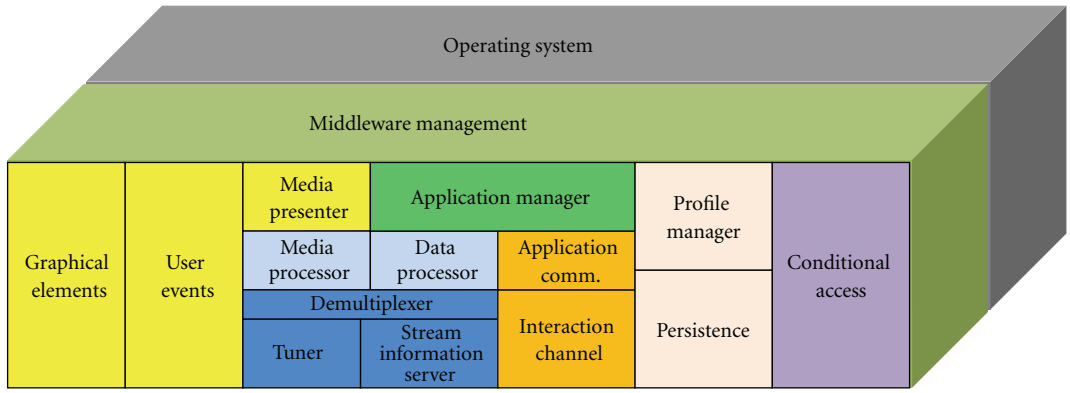

Figure 6: High level system architecture.

usage, memory usage, exhibition resolution, return channel type, and so forth. Finally, the environment variables can be used to define variables like the communication protocols currently being used, the network QoS, and so forth.

The concepts collected in this ontology were identified with 30 DTV researchers that worked in the Brazilian DTV system.

Based on this ontology, we can have a set of reconfiguration decisions. A very simple example can be shown in the case of the MediaProcessing component. Trojahn et al. [18] presents an evaluation of two Media Processing implementations. We can see that one of them requires less processing capabilities while the other one requires less memory. Based on this information and the information collected by the DTV client system, we can decide which one is more suitable for the moment. Of course, this example considers only two variables (processor and memory usage), but in the real case, several other variables may be considered as input to the Fuzzy logic engine.

5. Experimental Results

We performed a series of experiments with the middleware case study in order to get a clearer view of the feasibility of our approach.

As stated in the previous section, the client system components were developed in the Middleware Management layer of Figure 6. The tests were executed in a set-top box receiving the Digital TV signal through a 6 MHz broadcast channel and return channel based on PSTN modem. In order to execute the proof of concept, we generated a component descriptor with Fuzzy rules for two environment variables:

Table 2: Experimental results for GRASP based algorithms.

Instance		Optimal		GRASP			GRASPf		
ID	NC	z^*	T(s)	Z_{avg}	T(s)	Δ_{avg}%	Z_{avg}	T(s)	Δ_{avg}%
1	20	1184	1	1184	0.061	0.000	1184	0.409	0
2	30	592	2	600	0.031	1.351	592	0.376	0
3	50	870	30	927	0.06	6.552	870	0.44	0
4	60	609	109	650	0.067	6.732	613	0.449	0.657
5	80	—	—	1185	0.115	—	1106	0.729	—

device.resolution and *video.resolution*, referring, respectively, to the display device resolution and the received video resolution. After the fuzzy evaluation, the variables can be classified into three sets: *LDTV* (low definition), *SDTV* (standard definition), and *HDTV* (High definition). The output result can assume a value from 0 to 100 (zero means no adequacy, 100 means the best adequacy). We observed that the Fuzzy engine returned maximum adequacy level when the video resolution and the device resolution were equal. Minimum adequacy was received when we had big differences between video and display resolution.

In order to further validate our Fuzzy logic engine, 5 developers were invited to express their components adequacy in terms of fuzzy logic in order to feed the engine. Each generated component descriptor contained an average of 10 context variables. Accordingly to the developers, the values returned by the engine were consistent with their expected result.

Once the adequacy level is calculated, we can start the optimization stage. In this stage, the new architecture will be computed. As described in Section 4, the active repository uses a GRASP metaheuristic to compute the architecture and was developed using C++ language.

In order to test the GRASP algorithm, we generated several instances of the COSOP stated in Section 3.1 considering the number of components as NC = {20, 30, 50, 60, 80, 100, 500, 1000, 2000, 5000}. Each component in each instance was generated with a random number of required and provided interfaces. In these tests, the adequacy level was also randomly generated from the interval [50, 100].

We tested the instances with three algorithms: the GRASP, GRASP*f* (a variation of GRASP that chooses only the best solutions before entering the local search step), and Branch and Bound algorithms. The branch and bound algorithm is used in order to find the optimal solution, this way we are able to compare the GRASP-based algorithms with the optimal solution.

The results can be found in Table 2.

The z^* indicates the optimal solution found by the Branch and Bound method. $T(s)$ indicates the time, in seconds, spent by the approach to reach the solution. The Z_{avg} column indicates the average of the solutions found after five executions of the metaheuristics. Finally, the following equation is used to evaluate the average of the delta value that indicates the difference between the Z_{avg} and the optimal solution.

Average of the delta value:

$$\Delta_{AVG} = \left[\frac{(Z_{AVG} - z^*)}{z^*}\right] \times 100. \quad (5)$$

According to Table 2, the GRASP*f* algorithm found the best solution when NC equals to 20, 30 and 50. In the instance number 4 (NC = 60) the gap was less than 1%. On the other hand, the GRASP algorithm results had gaps of more than 1%.

With respect to computation time, we can see that the time to execute a simple instance with 60 components were more than 1 minute in the case of the optimal solution. Furthermore, the optimal solution could not compute the instance 5 (NC = 80) because there was no system resources to instantiate the problem. The branch and bound results were obtained using the LINGO Optimization Modeling software [19].

In order to evaluate our approach with the case study, we employed our active repository to receive context information from a set of FlexTV middleware instances. In this case, the context information was sent from the middleware to the repository as an OWL instance file. This OWL representation was generated using the Jena framework [20]. This framework provides, among other tools, an API to read and modify OWL models. Through this API, we were able to verify which concepts are present in the ontology and instantiate these concepts associating values to their properties. Once the context is delivered to the active repository, the components start to be evaluated accordingly to it. The Fuzzy engine was implemented as a Fuzzy inference system using Matlab [21] mathematical programming tool. The Optimizer is responsible to calculate the new component architecture using algorithms discussed in Section 3. Generating optimized component architectures generates adapted architectures and reconfiguration commands and sends to the client system by using a XML file.

In the user point of view, the time spent by the system to perform the necessary data collection, to send the context information to the server and to receive and load components into memory are irrelevant since these operations occur in background, so no impact to the user experience is noted. Only the time spent during the system reconfiguration is relevant, since during this period the system will be unavailable to the user. In the performed tests, we verified that the maximum time that the system became unavailable to the user was 2,190 ms, in this specific case, most of the time was spent changing the components responsible to handle the video stream (1,800 ms). The restart process of these components is slow and similar to the time required to perform a channel change in the digital TV receiver.

Considering that no reconfigurations caused failures in the system and that the worst unavailable time was similar to the channel change time (which the users are already used to), the test results are considered to be satisfactory. Furthermore, if we decide to change some component that is not related to the video stream chain, the video will not be disrupted and the reconfiguration may be totally seamless to the user.

6. Related Works

Component based software is an area that have grown in the pass years, the concept of components and their capacity of improve the software development cycle have created a vast number of architectures to support component-based systems.

6.1. Code Conjurer. Code Conjurer [22] works as a component finder that analyses the developer code in project or writing stages. To do the component search it uses the component repository Merobase [23]. Due to the high number of components found every time in Merobase's search, many components are high reclined to fail in the functionality aspects. To overcome this issue the Code Conjurer implements a test-driven search that will test the components found by using user-defined test cases to the system and then suggest only that ones that passed all test cases. This way the developer will have a better quality of the suggested components. Furthermore, the developer can also employ UML diagrams and descriptions to suggest eligible components.

There are several differences between our work and that presented in Code Conjurer. First the information used by it is focused in the programmer's context, our system uses contextual information that comes from the execution context. Second, Code Conjurer indexes source code that the programmer can adapt to his system in development time, the components stored in our system are already compiled within a well-defined component model.

There are many other software that use components as the Code Conjurer tool [11, 24] and others that follow a similar category but focus on helping the developers navigating the vast number of APIs provided by their currently used languages [25, 26], however, as mentioned before they focus on accelerating code development not focusing on running systems.

6.2. Fractal. Fractal [27–29] is a hierarchical and reflective component model that provides functionalities such as inspection and change of component's internal behavior. In order to change the components internal behavior, Fractal defines a flexible component system. The main concept is that Fractal components work as a membrane for existent components that allows introspection, composition, sharing, and reconfiguration capabilities.

A Fractal component is composed of a membrane and its contents. The content is a finite set of components. The membrane is an encapsulation layer that can have external interfaces, which can be accessed by external components; internal interfaces, which can be accessed only by internal component; controllers, which are used to, among other functions, inspect the component's internal contents, control internal components behavior and intercept actions.

To test the proposed model the authors have implemented Julia, a java version of the component model that implements many of the Fractal's capabilities. The results are presented as an evaluation of Julia's implementation in terms of overhead, memory allocation, and framework usage. The main difference between Fractal's architecture and the architecture considered in our approach is that Fractal the first one focus on software components and forms of handling them in different ways. A possible future work is to evaluate Fractal as the component model used by the middleware implementation. This way, we can identify possible positive and negative aspects and extension points to Fractal.

6.3. OpenCOM. OpenCOM [30] is a component-based technology that is programming language independent. OpenCOM was designed to support the development of general systems while allowing run time configuration.

In the OpenCOM model, components are loaded in a macro component called Capsule through a set of APIs called Component Runtime Kernel (CRTK) containing all configuration operations (such as load, connect, disconnect, among others) associated to a rollback mechanism that is used in case of failure. Components are functional units that are encapsulated and have Receptacles and Interfaces that are, respectively, required and provided.

OpenCOM support additional services that are implemented as components thus nothing changes in the component model, like the reflexive meta-models that improves the dynamic reconfiguration of systems. Among the reflexive meta-models those that are more related with our work are what follows.

(i) Architectural meta-model: exposes the architectural composition of the components in a Capsule as a connected graph.

(ii) Interface meta-model: Allows the discovering of information about the component's interface types and the using of dynamic interface discovery.

(iii) Interception meta-model: Allows the creation of interceptors between interfaces and receptacles.

(iv) Resource meta-model: Offers the access to the low level platform resources such as threads, procedures, memory, and others.

As in the case of Fractal, we also identify as future work, the possibility to evaluate OpenCOM in REATIVO's context.

6.4. Design and Implementation of a Safe, Reflective Middleware Framework. This work presents a model for composition of safe middleware services where applications could have variant nonfunctional requirements over the time [31].

To make this possible, the authors identify core services that are the basis to other services in order to make possible the variation of the requirements. According to the authors, this model handles the complexity of reasoning over the components in distributed system in order to provide a better component if such is available.

The new composed services have to be weakly coupled with the application requirements so these components could be changed over the time without any major reconfiguration issue. In our work, we go further by providing a complete infrastructure for context evaluation and architecture reconfiguration.

7. Conclusion and Future Directions

This paper presented an approach to adapt context-aware component-based software systems. A new active repository approach is employed in which components are made available not only during software development phase, but also during the operational phase of the software in a context aware form. In this context, the repository is able to receive the client system context and generate a new architecture that better fit this context.

We defined the concept of execution context as an instance of a domain dependent ontology. This ontology defines a set of variables that are relevant to the system in terms of functionality.

In order to decide which components are more suitable to a given context, we employed a fuzzy logic approach to calculate the adequacy level of the components. To be compatible with REATIVO, components registered in the repository must provide a set of definitions as parameters to the fuzzy inference engine.

REATIVO is able to evaluate the entire system adequacy by assembling the components in a system composed by the "most adequate" components. The Component-Oriented Software Optimization Problem is proven to be NP-Complete, thus it would need an optimization approach. In this context, we employed GRASP metaheuristic to compute the optimized architecture.

In order to achieve a more concrete result, we applied our solution to the FlexTV Middleware case study and obtained some convincing results regarding the feasibility of the approach, since both the client side and server side were successfully implemented and the time spent in reconfiguration procedures were satisfactory.

In Section 2.2, we stated that REATIVO's repository is domain independent. As a future work, we need to employ REATIVO in new domains other than digital TV in order to provide empirical proof of this statement, in addition to the reasons presented in that section. Furthermore, the application of different algorithms to substitute the fuzzy logic and GRASP is also desired. In this way, a better evaluation can be performed related to the efficiency of the chosen algorithms.

References

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[2] C. Szyperski, D. Gruntz, and S. Murer, *Component Software—Beyond Object-Oriented Programming*, ACM Press, 2nd edition, 2002.

[3] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.

[4] C. W. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.

[5] Component Source, 2012, http://www.componentsource.com.

[6] S. Henninger, "Using iterative refinement to find reusable software," *IEEE Software*, vol. 11, no. 5, pp. 48–59, 1994.

[7] K. Inoue, R. Yokomori, H. Fujiwara, T. Tamamoto, M. Matsushita, and S. Kusumoto, "Component rank: relative significance rank for software component search," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pp. 14–24, Portland, Ore, USA, May 2003.

[8] C. J. M. Geisterfer and S. Ghosh, "Software component specification: a study in perspective of component selection and reuse," in *Proceedings of the 5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, p. 9, February 2006.

[9] B. Srivastava, K. Ponnalagu, N. C. Narendra, and K. Kannan, "Enhancing asset search and retrieval in a services repository using consumption contexts," in *Proceedings of IEEE International Conference on Services Computing (SCC '07)*, pp. 316–323, July 2007.

[10] D. Lucrécio, A. Prado, and E. Santana, "A Survey on software components search and retrieval," in *Proceedings of the 30th Euromicro Conference (Eromicro '04)*, pp. 152–159, 2004.

[11] Y. Ye, *Supporting component-based software development with active component repository systems [Ph.D. thesis]*, Department of Computer Science, University of Colorado, 2001.

[12] G. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic*, Prentice Hall, Englewood Cliffs, NJ, USA, 1995.

[13] L. Leite, G. Farias, G. Souza, and S. Meira, "Uma Metaheurística GRASP para Otimização de Sistemas Orientados a Componentes de Software," in *Anais do Simpósio da Sociedade Basileira de Pesquisa Operacional*, Ceará, Brazil, 2007.

[14] P. Winter and M. Zachariasen, "Euclidean Steiner minimum trees: an improved exact algorithm," *Networks*, vol. 30, no. 3, pp. 149–166, 1997.

[15] Y. Lee, S. Y. Chiu, and J. Ryan, "A branch and cut algorithm for a Steiner tree-star problem," *INFORMS Journal on Computing*, vol. 8, no. 3, pp. 194–201, 1996.

[16] M. Resende, "Greedy randomized adaptive search procedures (GRASP)," in *Encyclopedia of Optimization*, vol. 2, pp. 373–382, Kluwer Academic Publisher, 2001.

[17] Stanford Center for Biomedical Informatics Research, "The Protégé Ontology Editor and Knowledge Acquisition System," 2012, http://protege.stanford.edu/.

[18] T. H. Trojahn, J. L. Goncalves, J. C. B. Mattos et al., "Tests and performance analysis of media processing implementations for the middleware of Brazilian Digital TV system using different scenarios," in *Proceedings of the 5fth International Conference on Multimedia and Ubiquitous Engineering (MUE '11)*, pp. 95–100, June 2011.

[19] Lindo Systems, "Lingo—Optimization Modeling Software for Linear, non-linear and Integer Programming," 2012, http://www.lindo.com/index.php?option=com_content&view=article&id=2&Itemid=10.

[20] The Apache Software Foundation, "Apache Jena," 2012, http://incubator.apache.org/jena/.

[21] MathWorks, "Matlab—The Language of Technical Computing," 2011, http://www.mathworks.com/products/matlab/.

[22] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: pulling reusable software out of the thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.

[23] "Merobase—Software Component Finder," 2012, http://www.merobase.com.

[24] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick, "Rascal: a recommender agent for agile reuse," *Artificial Intelligence Review*, vol. 24, no. 3-4, pp. 253–276, 2005.

[25] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Proceedings of the ACM SIGPLAN Conference on Programming Language*

Design and Implementation (PLDI '05), pp. 48–61, ACM Press, Chicago, Ill, USA, June 2005.

[26] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pp. 204–213, ACM Press, New York, NY, USA, November 2007.

[27] G. Blair, T. Coupaye, and J.-B. Stefani, "Component-based architecture: the Fractal initiative," *Annals of Telecommunications*, vol. 64, no. 1-2, pp. 1–4, 2009.

[28] P. Merle and J.-B. Stefani, "A formal specification of the fractal component model in alloy," INRIA Research Report 6721, 2008.

[29] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The FRACTAL component model and its support in Java," *Software—Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[30] G. Coulson, G. Blair, P. Grace et al., "A generic component model for building systemssoftware," *ACM Transactions on Computer Systems*, vol. 26, no. 1, article 1, 2008.

[31] N. Venkatasubramanian, S. Gutierrez-nolasco, S. Mohapatra et al., "Design and Implementation of a Safe, Reflective Middleware Framework," CiteSeerX—Scientific Literature Digital Library and Search Engine, 2008.