



UNIVERSITY  
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14  
<http://www.dit.unitn.it>

COLLABORATIVE CASE RETRIEVAL  
WITHIN THE IMPLICIT CULTURE  
FRAMEWORK.

Aliaksandr Birukou, Enrico Blanzieri, Paolo  
Giorgini, and Michael Weiss

February 2007

Technical Report # DIT-07-015



# Collaborative Case Retrieval within the Implicit Culture Framework

Aliaksandr Birukou<sup>1</sup>, Enrico Blanzieri<sup>1</sup>, Paolo Giorgini<sup>1</sup>, and Michael Weiss<sup>2</sup>

<sup>1</sup> Department of Information and Communication Technology  
University of Trento - Italy

{aliaksandr.birukou, enrico.blanzieri, paolo.giorgini}@dit.unitn.it

<sup>2</sup> School of Computer Science  
Carleton University, Ottawa - Canada  
weiss@scs.carleton.ca

**Abstract.** The retrieval of cases plays important role in CBR systems. Current solutions to the problem of case retrieval adopt various similarity metrics such as utility-based, adaptation-guided similarity, and collaborative retrieval. In this paper, we present a collaborative architecture for case retrieval, where users share their experience in using cases and interact with one another by means of software agents. Past experiences are used to improve case retrieval. We describe a system for choosing software patterns as an implementation of the proposed architecture. Experimental results show that the system performs better than conventional retrieval engines.

## 1 Introduction

An important functionality of CBR systems is the retrieval of cases. Although usually retrieval is based on similarity measures, other notions such as *adaptability*, *diversity*, and *utility* have been used to retrieve cases [5, 21, 24, 25]. In adaptation-guided similarity [21], the information about *adaptability*, i.e. the knowledge about whether a case can be easily modified to fit a target problem, drives the retrieval. In other domains, the *diversity* of retrieved cases is an issue [12, 19, 22]. In *utility*-based similarity [5, 24, 25], the basic idea is that the best case is not the most similar (w.r.t some similarity definition) but the most useful (w.r.t some utility definition). In general, the utility function is unknown and emerges from the behavior of the users, so learning appears to be a viable choice. Recently, Stahl [25] proposed an approach aimed at learning directly the utility function of the user and then using it for the retrieval.

On the other hand, user collaboration plays an important role in the retrieval of cases and helps to improve the retrieval. In this direction Aguzzoli et al. [1] and Hayes et al. [18] proposed methods that exploit Collaborative Filtering [15]. They use one of the basic notion of Collaborative Filtering (CF): the representation of items is based on the judgments that the users gave. In the recommendation system literature, it is common to distinguish between content-based recommendation and collaborative-based recommendation. The former are based on a

representation of the item content, which is usually built (semi-)automatically. The latter are based on ratings that users gave to items. The strength of CF is that representing items by means of ratings does not require the representation of the content of the items and, conversely, its weakness is that it is impossible to produce results without ratings. The possibility of integrating CBR and CF has already been exploited in the past in order to overcome the reciprocal limitations, see e.g. [3]. Freyne and Smyth [14] considered collaborative retrieval across multiple case bases, which allows for the reflection of the expertise and preferences of complementary search communities.

The limitations of the solutions proposed so far are as follows: (1) collaboration occurs only during the retrieval step (mainly because the use of CF is limited to this step), while the other steps of the process, such as adaptation and reuse of cases are performed without collaboration; (2) it is not possible to consider general actions on cases, e.g. an adoption or a rejection of a case, for just a particular type of them (rating) is supported; (3) it is not possible to change the goal that drives the retrieval. For instance, a system designed to do a utility-based retrieval cannot perform adaptation-based retrieval for another group of users; (4) there are no approaches that include both collaboration and utility.

In this paper, we present a multi-agent architecture based on the Implicit Culture framework [9], for collaborative retrieval of cases. The Implicit Culture deals with the general problem of transferring implicit knowledge between/within agent communities, so as to allow an agent to exploit others' experience. The notion of Implicit Culture has been proposed as a generalization of CF [11], addressing the above shortcomings w.r.t general actions, system objective, and supporting interactions of artificial agents with the system. We present the application of the multi-agent architecture to a system that facilitates the process of selecting software patterns within a community of developers. Patterns are the cases and the Implicit Culture framework helps developers in the process of choosing a pattern suitable for a specified problem. A query about the problem includes a keyword-based problem description and a property-based project description. Thus, the system supports more complex queries than conventional keyword-based ones, allowing one to specify the context, namely the information about the project where the problem occurs. The proposed system outperforms a conventional retrieval engine in terms of precision and recall of cases. A related approach is used in the ReBuilder framework [16], where cases represent situations (problems) in which a pattern was applied in the past to a software design. ReBuilder supports the retrieval and adaptation of patterns. Cases are described in terms of class diagrams. Cases are retrieved based on a combination of structural similarity between the current design and a pattern, as well as the semantic distance between class names and role names in the pattern. Our approach is complementary as patterns are selected on the base of previous actions of other users.

The paper has the following structure. In Sect. 2, we describe the architecture. Section 3 describes an example of the proposed architecture, the system for

choosing software patterns. The experimental evaluation of the system is in Sect. 4, and, finally, Sect. 5 concludes the paper and discusses future work.

## 2 Implicit Culture Framework for the Retrieval of Cases

This section describes the multi-agent architecture based on the Implicit Culture framework and shows how it handles the retrieval of cases. We argue that given a case base, it is possible to build metadata about cases in terms of the actions that users performed on the case base in the past and to use this representation for the retrieval of cases. Therefore, the representation of a case is not based on its content but on the actions users performed on it, in analogy to the approaches that exploited CF [1, 18].

Actions are strongly related to the experiences users have in using the case base. For instance, concerning the use of patterns, the actions of a new user, who is unfamiliar with the case base are more explorative, while users familiar with the case base perform more exploitative actions [23]. It is often the case that the behavior, i.e. actions, of more experienced users are close to optimal, and it is because they have acquired the necessary knowledge to use the case base effectively. This knowledge, which we introduce as a “community culture”, very often results in being implicit, i.e. it is not represented by means of documents and/or information bases.

Implicit Culture is based on the assumption that it is possible to elicit the community culture by observing the actions of people in the environment and to encourage the newcomer(s) to behave similarly to more experienced people. Implicit Culture assumes that *agents* perform *actions* on *objects* (queries or cases, in our specific application) in the *environment* (see [10] for more details). The actions are considered in the context of *situations*, and therefore we say that agents perform *situated actions* [26]. The “culture” contains information about actions and their relation to situations, namely which actions are usually taken by the observed group and in which situations. This information is then used to provide newcomers with information about others’ behavior in similar situations. When newcomers start to behave similarly to the community culture (i.e. when they use cases in a proper way) we have the knowledge transfer. This knowledge transfer is realized by the System for Implicit Culture Support (SICS) and the relation characterized by this transfer is called *Implicit Culture*: “Implicit Culture is a relation between a set and a group of agents such that the elements of the set behave according to the culture of the group” [10].

The general architecture of an Implicit-Culture-based multi-agent system for case retrieval is shown in Fig. 1. The agents play the role of personal user agents and form a virtual community that represents a group of users. The use of agents provides a flexible and implicit way of sharing information about actions: they use the SICS module to answer user queries about cases, provide retrieved cases, and store all the actions in the database of observations using the SICS. Agents can also interact with one another to share expertise and knowledge of their users in using the case base.

The architecture of a SICS consists of the following three components:

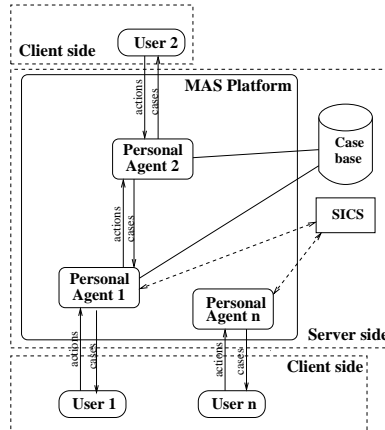
- an *observer*, which stores information about actions performed by the user in a database of observations;
- an *inductive module*, which analyzes the stored observations and applies learning techniques (namely, data mining or machine learning) to develop a theory about actions performed in different situations;
- a *composer*, which exploits the information collected by the observer and derived by the inductive module to suggest actions in a given situation.

The observer stores information about the case queries (needs behind the case retrieval of a user), which cases have been proposed as a solution, and which case has been chosen in return. The inductive module discovers problem-solution pairs by analyzing the history of the interaction of users with the system. A set of query-case pairs (which cases are selected for what queries) can be included in a *theory*. The theory contains rules of essentially the following form:

**if** *action\_one*(objects) **and** *action\_two*(objects) **then** *action\_three*(objects).

This means that the *action\_three* (and not, e.g. an *action\_four*) action must follow the *action\_one* and *action\_two* actions. The theory is used to specify the general goal of the system in the retrieval of cases, and it can be changed or modified at runtime. For instance, it can be extended with rules learnt directly from actions of the users on the case base. The goal of the composer is twofold. Firstly, it compares the query with the problem part of the theory mined by the inductive module in order to suggest the corresponding case. Secondly, the composer tries to match the query with the case by analyzing the history of observations and calculating the similarity between actions, taking into account the similarity between queries (the current one and those submitted in the past) and between cases, which can be defined in a particular way. Therefore, it can easily handle the situation when the similarity of cases is already defined and the architecture is used to add collaboration into the retrieval. The main role of the SICS in our architecture is to improve the retrieval of cases, transferring implicitly the knowledge the users have about cases.

The following actions are observed by the SICS in our architecture: a user can *request* a case from the case base, specifying a *query*; she can then either *apply* or *reject* one of the retrieved *cases*. The latter actions are observed in the context of the previously submitted query. In all cases agents of the actions are personal agents that represent their users. The general algorithm describing the retrieval process is shown in Fig. 2. The algorithm is executed by the personal agent of the user and depends on the action performed by the user. For instance, in case of the request action, the agent can retrieve cases from the case base. This step is optional and corresponds to the retrieval of cases with tools provided by the case base itself. The next step is to contact agents propagating the user query. This step is also optional and should be taken in case agents have some private collections of cases apart from the case base. After these steps, the agent contacts the SICS and, finally, shows all the available results to the user. Two



**Fig. 1.** The general architecture of Implicit-Culture-based case retrieval system

```

for all action a of the user do
  if (a.type == 'request') then
    query q = a.query
    observe-action(request,q)
    retrieve-cases-from-case-base(q)
    contact-agents(q)
    contact-SICS(q)
    show-user(results)
  else if (a.type == 'apply') then
    observe-action(apply,a.query,a.case)
    contact-agents(apply,a.query,a.case)
  else if (a.type == 'reject') then
    observe-action(reject,a.query,a.case)
    contact-agents(reject,a.query,a.case)
  end if
end for

```

**Fig. 2.** The algorithm of the retrieval of cases

kind of feedback actions, 'apply' and 'reject' are observed and propagated to the other agents, in case agents were asked when processing the query. We refer the reader to our previous paper [10] for the details on the algorithms adopted within the SICS.

The architecture has several advantages. Firstly, general actions on using cases is supported, so we are not limited to just ratings as in CF. For instance, high ratings can be mapped to *apply* actions in our architecture, while low ratings can be mapped to *reject* actions, and the possibility of considering an action *rate* still remains. Also, the relation between different actions are supported by means of cultural theory [10]. Secondly, the goal of the system can be defined according to the needs of the current community. In this case, the utility of retrieved cases can be measured as deviation from the goal, and a case is considered useful if it helps to achieve the goal. Moreover, the goal can be learnt implicitly from the actions of the community. Thirdly, our architecture could be used in order to add collaboration in systems where the similarity of cases is already defined. In this way, it can integrate utility and collaboration if similarity is defined in terms of utility. Also, agents in our architecture can cooperate with one another autonomously, namely without the direct control of the users, and thereby decrease the amount of input required from users. Finally, an agent can interact and cooperate with other agents on behalf of its user in order to manage the user's social relationships, even when the user is not connected to the system (see e.g. [7, 8] as examples of such systems).

### 3 The System for Choosing Software Patterns

This section presents the system that helps to choose software patterns and describes the retrieval process within the system. The system is intended for the use within an IT-company, or just within a project group, and it should adapt the suggestions on the use of software patterns to the specificity of the software development process adopted within the company or project group, converging to the “community culture”.

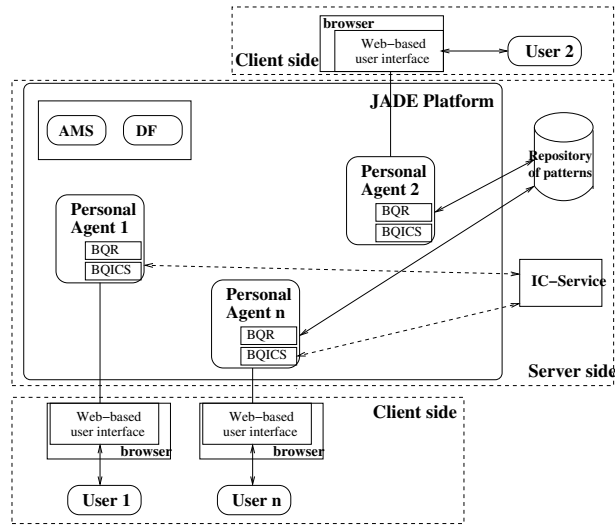
Patterns enable an efficient transfer of design experience by documenting common solutions to recurring design problems in a specific context [2]. Each pattern describes the problem it is dealing with, situations when the pattern can be applied, known uses, etc. Our motivation for adopting an implicit culture approach in the system for choosing software patterns stems from: (1) the continuous increase in the number of documented patterns, for instance, the *Pattern Almanac* [20] lists more than 1200 patterns; (2) the difficulty less experienced developers face in using patterns. The following quote from [23] is indicative of the difficulty inherent in using patterns:

Only experienced software engineers who have a deep knowledge of patterns can use them effectively. These developers can recognize generic situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

The difference between these two types of developers is that an experienced developer uses implicit knowledge (in particular, her own experience) about the problem (see [13] for a more general discussion on this point). When we look at the community of a developer’s peers, knowledge is called *implicit* when it is embodied in the capabilities and abilities of the community members (developers). It is *explicit* when it is possible to describe and share it through documents or knowledge bases. To select appropriate patterns, inexperienced developers should acquire the implicit knowledge that more experienced developers have.

For example, let us consider a repository of security patterns and a programmer that needs to improve access control in a system that offers multiple services. Let us suppose that for an experienced developer knowledgeable in security it is apparent to use the Single Access Point pattern. If the system is able to use previous history to suggest that the novice uses the Single Access Point pattern and she actually uses it, then we say that she behaves in accordance with community culture and the implicit culture relation is established. We will use this example as a running example throughout the paper.

The architecture of the system is a refined Implicit-Culture-based collaborative retrieval architecture and it is given in Fig. 3. The system consists of a web-based user interface at the client side and a multi-agent platform at the server side. As a case base, we use the repository of security patterns (adopted from [patternshare.org](http://patternshare.org) [17]), however, it can be further extended with adding



**Fig. 3.** The architecture of the system. *Personal agents* process queries from *users* and access the repository of patterns to retrieve potentially relevant patterns; the *IC-Service* is exploited by the agents in order to create recommendations from the history of past interactions; the *Agent Management System (AMS)* exerts supervisory control over the platform: it provides agent registration, search, etc.; the *Directory Facilitator (DF)* provides agents with other personal agents' IDs; *BQR* stands for BehaviourQueryRepository used to access the repository, and *BQICS* stands for BehaviourQueryICService respectively.

other patterns. A user accesses the system by submitting a query via the web-based interface in her browser. In this system a query includes a description of the problem and a description of the project, in which the problem is encountered. The problem is described by a set of keywords, optionally restricted to specific elements of the pattern description (e.g. problem, context, etc.). The project description can be represented as a set of properties (e.g. project size, required level of data protection, etc.). In our running example, the user could submit a query with the following problem description: “access control in a system that offers multiple services” related to the project that has the following set of properties: {Name: OnlineBanking, SecurityLevel: High, ProjectSize: Medium}. The other considered projects have the following properties: {Name: e-BookShop, SecurityLevel: Medium, ProjectSize: Medium}, {Name: eLectons, SecurityLevel: High, ProjectSize: Big}.

Each user is assisted by a personal agent. The goal of a personal agent is to help the user choose a pattern suitable for the submitted query. In order to fulfill this goal, the agent can access the SICS via the *IC-Service* [6], or to access the case-base directly via the API provided by Lucene, a fully-featured text search engine library (<http://lucene.apache.org/>). The personal agents in the system are software agents running on the multi-agent platform at the server side.

In our example, the user's personal agent should suggest using the Single Access Point pattern. If the agent does so because someone else has already

used this pattern for similar problems, it distributes the knowledge about the use of patterns within the community.

In our application, the SICS analyzes the following actions:

action	objects
request	problem_description, project_description
apply	pattern, problem_description, project_description
reject	pattern, problem_description, project_description

Since all the actions are performed by developers, we omit agents from the table. We explain the information contained in the table in detail. A developer *requests* the system to find patterns that are suitable for her task. The developer *applies* the pattern when she implements it in the code, and can specify the inapplicability of a pattern to the task as a *reject* action. To observe these action, for now, we request explicit feedback from the developer. This is a reasonable assumption, since the amount of the input required from the user is very low. It does not contradict the notion of Implicit Culture, since it is the propagation of the culture within community which is implicit, and not the observations on actions. In the running example, examples of actions are: `request(query)`, `apply(SingleAccessPoint, query)`, `reject(Authenticator, query)`, where `query` contains `problem_description` and `project_description`.

The search scenario is given in Fig. 4. A user submits a query via the user interface, from where the query is forwarded to the user’s personal agent. In the first step of the search process, the personal agent accesses the pattern repository and retrieves a set of patterns relevant to the query. In the second step, the personal agent submits a query to the SICS and receives a list of recommended patterns. Thus, the result consists of patterns retrieved from the repository and patterns recommended by the SICS. The feedback from the user is collected via the *apply* and *reject* actions, which mark a pattern as suitable or unsuitable for the problem, respectively.

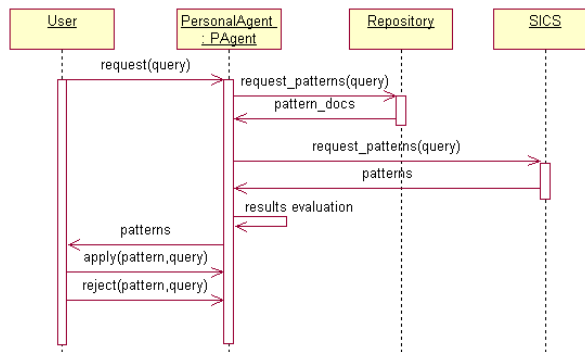


Fig. 4. Sequence diagram of the search process.

The SICS inside the *IC-Service* processes the query within two steps. In the first step, the SICS matches the action contained in the query, i.e. the *request* action, with the theory and determines the action that must follow, i.e. the *apply* action. In the second step, the SICS finds situations where the *apply* action has been previously performed, thus determining the patterns used for similar problems in the past. In this step, the similarity between the current query and the previously submitted queries is calculated. As a result, the SICS returns a set of patterns that have been used for similar problems in the past. A pattern is recorded as “applied” or “rejected” if a user indicates so explicitly.

Let us illustrate how the search process takes place in our example. The user submits the *request* action with the following query: {ProblemDescription: “access control in a system that offers multiple services”; Project: {Name: OnlineBanking, SecurityLevel: High, ProjectSize: Medium}}. In the first step the agent retrieves patterns from the repository: SingleAccessPoint and RoleBasedAccessControl. In the second step, the agent queries the *IC-Service*. The SICS matches the *request* action with the theory. The theory contains rules of essentially the following form:

if *request*(query) then *apply*(pattern-X,query)

This means that the *apply* (and not, e.g. a *reject*) action must follow the *request* action. So, the SICS matches the *request* action with that part of the theory that represents a problem, and searches for situations where the *apply* action has been performed. It finds the following situations (situation\_id, the action, problem description, project, pattern):

1	apply	access control in a system that offers multiple services	pp	SingleAccessPoint
2	apply	only authorized clients should access the system	pp	PolicyEnforcementPoint

where pp={Name: e-BookShop, SecurityLevel: Medium, ProjectSize: Medium}. As a result, the SICS returns the SingleAccessPoint pattern, chosen in the most similar situation w.r.t. the submitted query<sup>3</sup>. After the evaluation of the results, the following list of patterns is displayed in the user interface: {SingleAccessPoint, PolicyEnforcementPoint, RoleBasedAccessControl}. Having analyzed the proposed patterns, the user *applies* the SingleAccessPoint pattern and indicates this in the user interface. She also marks the RoleBasedAccessControl pattern as unsuitable, thus performing the *reject* action.

The system is implemented using JADE 3.4.1 (Java Agent DEvelopment framework) and uses the *IC-Service* [6] for the retrieval of patterns. Although

<sup>3</sup> Without going in detail of the general algorithm of similarity calculation, let us say that the similarity between two actions in this case is calculated based on the similarity of names of actions and objects. In this case we have two objects: problemDescription and projectDescription, and the similarity between problem descriptions is calculated as the fraction of common terms, while the similarity between project descriptions is calculated as the fraction of equal properties (ProjectName, ProjectSize, SecurityLevel).

called a “service”, the *IC-Service* can be used in a number of ways, in particular as a Java library (the way we use it in the system).

## 4 Experimental Results

The goal of the experiment is to compare the performance of the system with and without the SICS.

In the experiment we implemented in each agent a class that simulates the querying behavior of a real user. The main functions of this class are: (1) provide pseudo-user input in order to enable the personal agent’s recommendations, and (2) generate pseudo-user response to the recommendations. The input is provided and the responses are generated according to a user profile. The user profile contains a sequence of sets of keywords and a set of pairs pattern-project name. The intuition behind the user profile is as follows: the user has a single problem to solve using patterns, the problem occurs in different projects, can be described in a number of ways (each set of keywords in the sequence describes the problem), and can be solved with the use of one of the patterns contained in the user profile.

We use the following measures [4] in order to evaluate the quality of suggestions:

- Given a query composed of a set of keywords and a project description, a pattern is **relevant** to the query if the keywords and the corresponding pattern-project name pair are in the profile.
- **Precision** is the ratio of the number of suggested relevant patterns to the total number of suggested patterns, relevant and irrelevant.
- **Recall** is the ratio of the number of proposed relevant patterns to the total number of relevant patterns.
- **F-measure** is a trade-off between precision and recall. It is calculated as follows:

$$\text{F-measure} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}.$$

In our experiment, we have not used the inductive module of the SICS to update the theory and the recommendations are generated entirely by the composer module. Also, models of the users have not produced *reject* actions, just *request* and *apply*.

To build a small community of five developers, we took away five patterns from the repository of Security Patterns<sup>4</sup> and assigned them to each of the developers as shown in Table 1. These patterns have been used in order to create sequences of sets of keywords, corresponding to the descriptions of the problems queried by the users. The sequences are created as follows: given a document, we construct a distribution of the terms in the document and then each element of the  $k$ -element sequence is a sample from this distribution, represented as an  $n$ -dimensional tuple. Here  $k \geq 1$  is the number of searches performed in a

<sup>4</sup> The repository of Security Patterns contains 59 patterns.

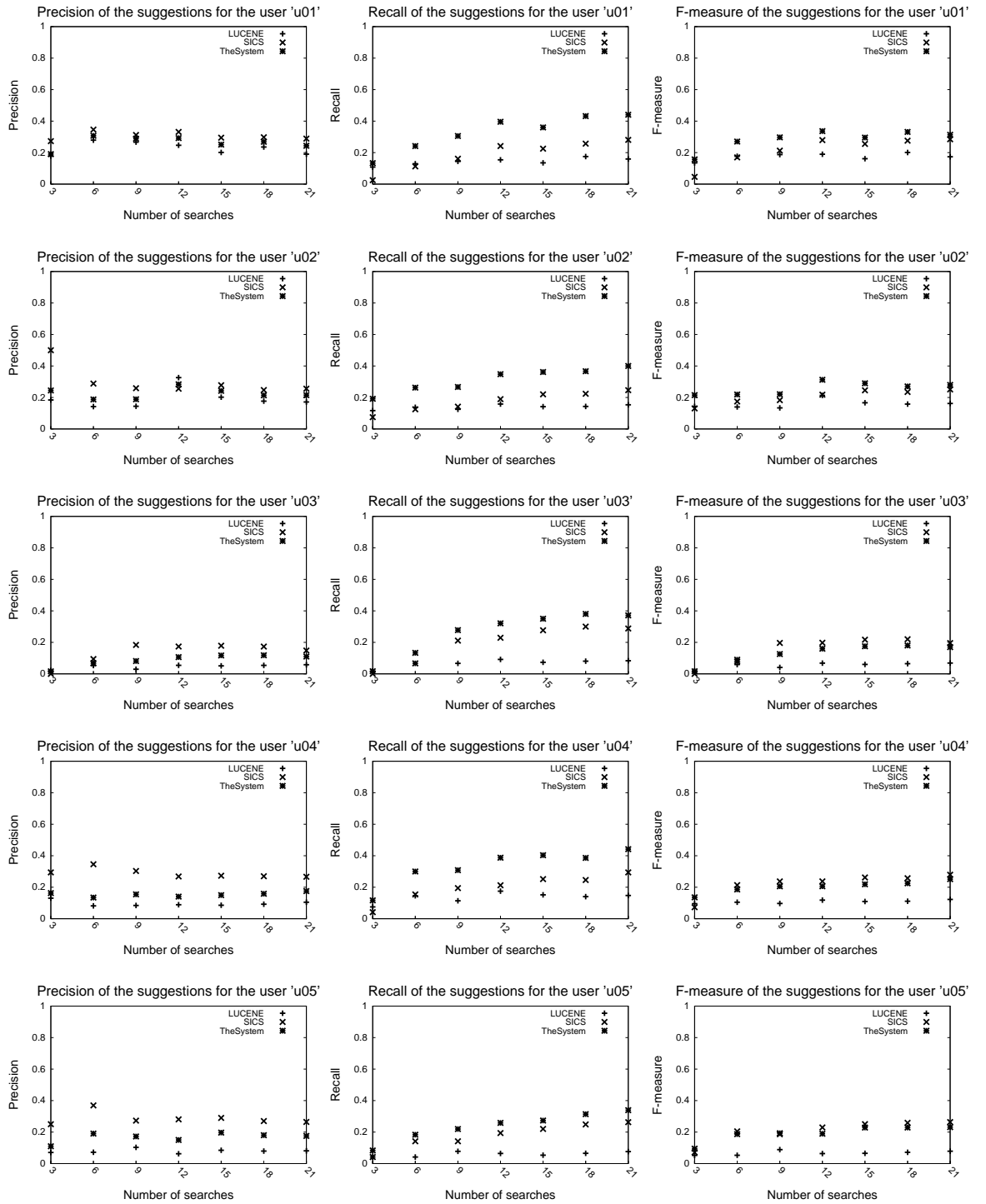


Fig. 5. The precision, recall, and F-measure of suggestions in the experiment

**Table 1.** User profiles

ID	profile pattern	project	relevant pattern
u01	ControlledProcessCreator	OnlineBanking OnlineBanking e-BookShop eLectons	ControlledObjectCreator Execution Domain ControlledObjectCreator Execution Domain
u02	StatefulFirewall	OnlineBanking OnlineBanking e-BookShop eLectons	PacketFilterFirewall ProxyBasedFirewall PacketFilterFirewall ProxyBasedFirewall
u03	VirtualAddressSpaceAccessControl	OnlineBanking eLectons	ExecutionDomain ExecutionDomain
u04	Authorization	OnlineBanking OnlineBanking e-BookShop eLectons	ReferenceMonitor RoleBasedAccessControl ReferenceMonitor RoleBasedAccessControl
u05	MultilevelSecurity	OnlineBanking OnlineBanking e-BookShop eLectons	ReferenceMonitor RoleBasedAccessControl ReferenceMonitor RoleBasedAccessControl

simulation, and each query in the sequence consists of  $n \geq 1$  keywords. The problems occur in the context of the projects listed in the running example in the previous section: OnlineBanking, e-BookShop, eLectons. As it is possible to see from the table, the project affects the choice of pattern.

To determine the patterns that are marked as “solution to the problem” and are placed in the user profile, the following approach is adopted. We represent each document in the repository as ‘*a bag of words*’ [4]. Then we calculate the similarity between the document used to create the profile and the rest of the repository. The cosine similarity metric [4] is used. The document(s) with the highest similarity (excluding the documents used to create profiles) are added to the profile as “solutions to the problem”. We selected two documents for users u01, u02, u04, u05, one document for user u03 and created five user profiles using patterns that are given in Table 1. Please note that there is a partial overlap in the profiles (e.g. u01 and u03), so the transfer of knowledge takes place. The profile of user u03 contains only one pattern.

The similarity between problem descriptions is calculated as the fraction of common keywords in the two descriptions. The similarity between projects is calculated as the fraction of equal properties. In the experiment we set the similarity threshold in such a way that actions are similar if their names are the same, and they have one keyword and one project property in common (or two keywords, or two project properties). In other words, the *apply* actions are similar if problems have a non-trivial overlap. Please, note, that focusing on the community as a whole, we do not differentiate between different developers when calculating similarity between actions.

In the experiment we set  $n = 3$ , so user queries consisted of three keywords and the project description in the query was chosen randomly with equal probabilities among the three above-mentioned projects. We ran simulations with different number of searches, namely  $k=3,6,9,12,15,18$ , and 21, measuring the

precision, recall, and F-measure of the recommendations after completing each  $k$ -query sequence. At the end of each  $k$ -query sequence, the database of observations is deleted in order to have the *IC-Service* producing recommendations from scratch. We repeated simulations 10 times and averaged the precision, recall, and F-measure to control the effect of the order and keywords of queries.

The results contain the precision, recall and F-measure of the patterns retrieved from the Lucene pattern repository, recommended by the SICS module, and by the system (both repository results and recommendations). Figure 5 shows the precision, recall, and F-measure of the recommendations produced by the five personal agents for five developers. The curves marked as “LUCENE” correspond to the performance of the system without the SICS module.

The results show that the recommendations of the system maintain a certain level of quality even for a small number of searches. The precision and recall of the SICS’s recommendations is almost always higher than the precision and recall of patterns obtained from the Lucene repository. This is notwithstanding the fact that the number of patterns retrieved from the Lucene repository is limited only by the number of documents relevant to the query, while the number of recommendations from the SICS is limited by the number of previous uses of the pattern and is usually smaller. The F-measure of suggestions produced by the system as a whole, in the most cases is higher than the F-measure of the suggestions produced by the Lucene or the SICS alone. This suggests that (1) the system with the SICS module outperforms the system without this module, (2) the approach of complementing results from the pattern repository with recommendations of the SICS proves to be useful.

## 5 Conclusion and Future Work

We have presented a multi-agent Implicit-Culture-based architecture for collaborative case retrieval. The architecture enables effective sharing of knowledge on the use of cases in a community of users. The implementation of the proposed architecture has been considered as a case study and experiments have indicated improvements over conventional case retrieval.

The multi-agent architecture permits the basic operations of the SICS to be performed without direct involvement of users. Indeed, agents contribute to the propagation of the information about user actions to other agents.

Agents access a single SICS module, which learns how the community as a whole uses the case base. However, in some domains (see, e.g., the recommendation system for web search, *Implicit* [8], and the system for facilitating the search of scientific publications [7]), it might be useful to incorporate the SICS module into each single agent, e.g. in order to have more autonomy, or because communication overload can be very heavy.

In this paper we have shown that the Implicit Culture framework can be applied for collaborative indexing and retrieval of cases. However, in principle, the Implicit Culture framework could also deal with adaptation and repair of cases by observing relevant actions. In particular, the repair can be considered

as an adjustment of the system to changes in usage. The adaptation could be performed with a specific use of the inductive module within a SICS: a general theory, representing the process of pattern selection could be learned, and new patterns could be selected based on the already chosen ones. In future work, we will elaborate on how such sequences of patterns can be learned using our framework.

## 6 Acknowledgements

This work is funded by research projects EU SERENITY "System Engineering for Security and Dependability", and by Fondo Progetti PAT, MOSTRO "Modeling Security and Trust Relationships within Organizations" and QUIEW (Quality-based indexing of the Web), art. 9, Legge Provinciale 3/2000, DGP n. 1587 dd. 09/07/04.

## References

1. S. Aguzzoli, P. Avesani, and P. Massa. Collaborative case-based recommender systems. In *ECCBR '02: Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, pages 460–474. Springer-Verlag, 2002.
2. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A pattern language*. Oxford University Press, 1977.
3. M. Balabanović and Y. Shoham. Fab: content-based, collaborative recommendation. *Communications of the ACM*, 40(3):66–72, 1997.
4. P. Baldi, P. Frasconi, and P. Smyth. *Modeling the Internet and the Web: Probabilistic Methods and Algorithms*. Wiley, 2003.
5. R. Bergmann, M. Richter, S. Schmitt, A. Stahl, and I. Vollrath. Utility-oriented matchin: a new research direction for case-based reasoning. In *Proceedings of the German Workshop on Case-Based Reasoning*, 2001.
6. A. Birukou, E. Blanzieri, V. D'Andrea, P. Giorgini, N. Kokash, and A. Modena. IC-Service: A service-oriented approach to the development of recommendation systems. In *Proceedings of ACM Symposium on Applied Computing. Special Track on Web Technologies*, 2007.
7. A. Birukou, E. Blanzieri, and P. Giorgini. A multi-agent system that facilitates scientific publications search. In *AAMAS '06: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems.*, pages 265–272. ACM Press, 2006.
8. A. Birukov, E. Blanzieri, and P. Giorgini. Implicit: An agent-based recommendation system for web search. In *AAMAS '05: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 618–624. ACM Press, 2005.
9. E. Blanzieri and P. Giorgini. From collaborative filtering to implicit culture: a general agent-based framework. In *Proceedings of the Workshop on Agents and Recommender Systems*, Barcelona, 2000.
10. E. Blanzieri, P. Giorgini, P. Massa, and S. Recla. Implicit culture for multi-agent interaction support. In *CoopIS '01: Proceedings of the 9th International Conference on Cooperative Information Systems*, pages 27–39. Springer-Verlag, 2001.

11. E. Blanzieri, P. Giorgini, S. Recla, and P. Massa. Information access in implicit culture framework. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 565–567. ACM Press, 2001.
12. P. Cunningham and G. Zenobi. Case representation issues for case-based reasoning from ensemble research. In *ICCBR '01: Proceedings of the 4th International Conference on Case-Based Reasoning*, pages 146–161. Springer-Verlag, 2001.
13. H. L. Dreyfus and S. E. Dreyfus. *Mind over machine: the power of human intuition and expertise in the era of the computer*. The Free Press, 2000.
14. J. Freyne and B. Smyth. Further experiments in case-based collaborative web search. In *Proceedings of ECCBR 2006*, pages 256–270, 2006.
15. D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.
16. P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento. Using cbr for automation of software design patterns. In *ECCBR '02: Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, pages 534–548. Springer-Verlag, 2002.
17. M. Hafiz and R. E. Johnson. Security patterns and their classification schemes. Technical report, 2006.
18. C. Hayes, P. Cunningham, and B. Smyth. A case-based reasoning view of automated collaborative filtering. In *ICCBR '01: Proceedings of the 4th International Conference on Case-Based Reasoning*, pages 234–248. Springer-Verlag, 2001.
19. D. McSherry. Diversity-conscious retrieval. In *ECCBR '02: Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, pages 219–235. Springer-Verlag, 2002.
20. L. Rising. *The Pattern Almanac*. Addison-Wesley Longman Publishing Co., Inc., 2000.
21. B. Smyth and M. T. Keane. Adaptation-guided retrieval: questioning the similarity assumption in reasoning. *Artif. Intell.*, 102(2):249–293, 1998.
22. B. Smyth and P. McClave. Similarity vs. diversity. In *ICCBR '01: Proceedings of the 4th International Conference on Case-Based Reasoning*, pages 347–361. Springer-Verlag, 2001.
23. I. Sommerville. *Software engineering (7th ed.)*. Addison-Wesley, 2004.
24. A. Stahl. Defining similarity measures: Top-down vs. bottom-up. In *ECCBR '02: Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, pages 406–420. Springer-Verlag, 2002.
25. A. Stahl and T. Gabel. Using evolution programs to learn local similarity measures. In *Proceedings of ICCBR-03*, 2003.
26. L. A. Suchman. *Plans and Situated Action*. Cambridge University Press, 1987.