The ICPCS Project
Current Status and Future Directions

Robert P. Biuk-Aghai
Faculty of Science and Technology
University of Macau
fstrpba@umac.mo
http://iccs.sftw.umac.mo/

Abstract

This paper presents some current results and future research directions of the ICPCS project—a research project underway at the University of Macau. The ICPCS project is concerned with studying principles and developing prototypes of systems that can support the distributed development of software by geographically dispersed software engineers. The initial focus of the project has been on specifying and developing a networked software configuration management repository for sharing software artifacts among a group of developers. The development of an initial repository has been completed and its main features are presented. Currently further work is going on to embed this repository, together with other software development tools, in a component-based, dynamically reconfigurable software engineering meta-environment. The basic ideas are explained and future directions are outlined.

Keywords: Collaborative software engineering, computer aided software engineering, software configuration management, component architecture, Java, JavaBeans.

1 Introduction

Over the past few years certain buzzwords have appeared in the literature: virtual enterprise, virtual classroom, distributed collaboration and others. They all reflect a growing interest in new ways of organizing human activity, enabled by the advances in computer communication technology. There are reports of IT being applied to distance learning [27], distributed meeting and presentation support [18], and to international medical collaboration [25]. Within the computer and software areas themselves, but also in the wider engineering field, interest in these trends has been active too, and various virtual teams work on projects in this fashion: Cutkosky et al. [9] report on collaborative engineering design over the Internet, while Fielding and Kašer [15] relate the experience of a large-scale software project collaboration.

Such mode of operation brings obvious advantages to an organization, principal among which are: cost reduction, by avoiding the relocation of project team members for the duration of the project, and thus the related travel and accommodation costs; and resource sharing, as highly sought-after experts can be shared on multiple concurrent projects. Together, these advantages may favourably affect the competitiveness of a company, and allow a more effective utilization of the human resources of an organization. One step further, several companies may forge temporary virtual organizations to collaborate on a product development or other project, which none of them alone would be able to achieve otherwise [20]. This enables organizations to step into markets which previously were closed to them, and brings about a greater degree of flexibility and nimbleness as virtual organizations regroup to suit new conditions. We believe that in the future virtual organizations and distributed group work will become increasingly common.

In software engineering, working with a distributed development team—one where team members are located geographically dispersed, yet cooperate on a shared piece of work—has become the focus of much research in its own right. As an example, at the past International Conference on Software Engineering, one panel, one state-of-the-art report and one workshop were entirely dedicated to this topic. On the one hand there is fundamental research into collaborative software engineering, addressing the combined aspects of Computer-Supported Cooperative Work (CSCW) and Software Engineering [26, 32, 16, 24]. On the other hand, however, there is the realization that collaboration in software engineering is in need of means that can enable such work patterns, namely a suitable support infrastructure, allowing for the effect-

1Held in Boston, Massachusetts, USA, 17-23 May 1997.
tive and efficient exchange of both messages and objects between members of a distributed work group. Research groups around the world are addressing various aspects related to this. Several groups focus on the aspect of making shared objects available \[12, 34, 5, 22, 17, 28\]. These may be referred to as approaches to object management, having in common that they establish some form of global repository and create an access mechanism to create, read, write, lock and unlock objects. There are also more specific projects directed at, for example, code inspection using the WWW \[30\], collaborative debugging using the WWW \[11\], distributed software configuration and deployment \[21\], secure distribution of code \[36\], and others. Some effort is also underway to combine these various solutions to create integrated software engineering environments, many of which are based on WWW technology or extensions thereof, such as the OzWeb system \[23\].

It is against this background of research activity that we have initiated the ICCS project at the University of Macau, to discover principles that are of importance in infrastructures supporting collaborative software engineering, and to develop relevant prototype systems. The following section gives an overview of the ICCS project and the work done so far. Section 3 introduces the repository that we have developed and section 4 presents our ideas on plugging software tool components. In section 5 we outline directions for further work, while some conclusions are presented in section 6.

2 ICCS Project Overview

In 1996, the ICCS project set out to define requirements for a distributed software engineering environment, and to implement relevant prototype systems. Our vision of distributed software engineering may be illustrated by following scenario:

Bill is a software engineer somewhere in the Far East. Currently he works on a project involving about 10 core team members in 6 countries around the world, none of whom he has ever met in person. He usually works on assignments like these for a period of 6 months to one year, being valued for his extensive knowledge and experience in developing specialized communication software.

At the beginning of every project, Bill sets up his development environment to give him the tools and support that are required for the new project. Since every project is different and since his personal preferences also are subject to change, he finds himself spending about one or more hours at the start of the project configuring his environment: choosing the software repository that will hold the whole project's artifacts (which almost always is in a different physical location than his own), the design tools required (on different projects different design methods and documentation conventions are dictated by the project leader), the compilers and other support tools for the required programming language, his favourite text editor, etc.

Configuring his software engineering environment is achieved by using a visual CASE tool builder. There are many vendors that produce CASE components, and typically his project client will send him all specifically prescribed tools as platform-independent mobile code through the network to his workstation. He then just plugs them together by manipulating visual representations of the tools on the screen, adding in his own tools and making sure that every functionality required by some tool is provided by others. After the final configuration of some options, he is set for getting down to work on his new project . . .

The rest of the story is left for the imagination of the reader. The main points which the story highlights are:

1. The software engineering environment is assembled from pre-fabricated components using a visual builder tool.
2. Tool components are implemented in mobile, platform-independent code.
3. A remote repository is used for holding software artifacts.

These points correspond to three overriding requirements that we set ourselves early on in the project. Our initial focus then was directed at defining detailed requirements of the software repository, while more recently we started investigating the component architecture to enable the plugging together of components. We view the repository as an interchangeable component of the software engineering environment just as the other tool components, and it too is implemented in a mobile language.

We decided to use Java as the mobile language, primarily because of its mobility property, but also for practical reasons as its immense popularity promised us a large number of deployment platforms possessing a Java Virtual Machine. Besides, it facilitates our own development task on a mix of
Unix and Windows 95 platforms. All ICCS prototypes developed so far are implemented in 100% pure Java with no native code. The following sections describe the current and planned ICCS components in greater detail.

3 ICCS Repository

As mentioned above, the ICCS project started by developing a networked repository. The aim was to provide a location on the network where distributed teams can share common project artifacts with each other effectively and efficiently. Such artifacts can be the products from any stage of the development process, such as analysis specifications, design diagrams, program source and object files, etc.

3.1 System Architecture

The ICCS system architecture is based on a client/server model. In order to enable a heterogeneous mix of backend servers hosting the contents of the repository we decided to adopt a three-tier client/server architecture, as shown in figure 1. A repository client component may request an operation on a repository object from a repository server, which processes the request and in turn issues a request to a database backend, where objects are actually stored. The responses are returned in the opposite direction. The repository server thus serves as the interface between the Java and non-Java parts of the system—other components are not aware of the non-Java database backend. This approach allows us to integrate existing non-Java repositories and legacy databases without affecting the code mobility of the rest of the system. It also makes it possible for a repository client to access different repositories with a heterogeneous mix of backend databases, as illustrated in figure 2. This does put some requirements on the type of backend database that may be used, which are explained in section 3.3 below.

In addition, we use a layered model of services provided by the repository, as shown in figure 3. The repository, consisting of repository client and server parts, provides certain services to other tool components. On the lowest level, communication to the database server is handled. The next-higher layer provides basic services for manipulating the repository. Both of these two layers reside in the repository server component, and their implementation is dependent on the database server used. The next layer provides higher-level services to access the repository, while the top layer provides services corresponding to specific configuration management policies (explained below). Both of these upper layers reside in the repository client and their implementation is independent of the chosen database server. Between each other, the repository client and server also exchange requests and responses, as the higher level services utilize the lower level services. The service model is explained in more detail elsewhere [4].

3.2 CM Features

For a distributed development to be workable, rigorous control of the artifacts is a key requirement, i.e. to know who did what to which artifact when and why. This is the primary objective of a software configuration management (CM) system, encompassing at least version management, but often also release management, and may even include change management, problem management, and process management. For the purposes of our project we decided to create a software configuration management repository supporting the core functionalities of version management and release management. We have specified following main features for our repository (in our description we adopt the term-
Version management: The control of revisions and variants of individual objects that are placed under version control. Revisions are different versions of an object evolving over time. A change history is maintained as a series of directed deltas, with the latest version existing in full. Variants are different versions of an object existing at the same time, e.g. versions for different operating systems, windowing systems, etc.

Release management: The control of composite objects that form a release and are made up of individual objects of various revisions and (typically) the same variant; e.g. a software system consisting of numerous individual files.

Separate meta-data: Data about objects is kept separate in the form of attributes that are attached to the object. This facilitates indexing and searching of the object space and allows arbitrary attributes to be added to support new required functionality.

Total versioning: Every object in the repository, corresponding to both the files and directories in a file system, is versionable.

N-dimensional variants: Objects may have different variants in different dimensions, e.g. \( x \) variants for different operating systems, and \( y \) variants for different database systems, yielding \( x \cdot y \) possible combinations. Variant dimensions are identified in the repository by attached variant attributes.

Policy-neutral: The repository is structured in such a way as not to favour any specific CM policy. CM policies prescribe the way that objects are to be manipulated, e.g. the checkout/check-in policy is popular in many existing CM systems [14]. By delegating policy issues to a higher service level, the same repository may simultaneously suit different CM policies.

To illustrate the features of the repository with a practical example, consider the case of a simple source tree as shown in figure 4. Here, ellipses correspond to directories, rectangles to files, and lines to composition relationships, e.g. we can see that directory `src` contains file `n.java` and sub-directories `a` and `b`, which in turn contain further files.

Placing this source tree under version control, it may now evolve in different revisions and variants. After some time, this source tree may have evolved into a form as that in figure 5 (in order to keep the example simple, only the revision evolution is shown and no variants). In this figure, object names are now suffixed with a revision number in the form “\( n \)”, where later revisions carry a higher revision number. For example, we can see that the file `n.java` has evolved into two revisions, and `p.java` into three revisions. Furthermore, entire directories have evolved: revision 2 of sub-directory `b` now contains one more file than before (`x.java`), while revision two of the parent directory `src` no longer contains the file `n.java`.

3.3 Repository Implementation

As mentioned earlier, the ICCS repository consists of three parts: a client component, a server, and a backend database system. The first two of these are implemented fully in Java, while the third one can be implemented in any language and can be of any degree of sophistication: it could be as simple as an application that stores flat files in the file system, or as sophisticated as a database management system. In order to maintain our code mobile, however, we require that non-Java backends allow access (i.e. both requests and responses) in any one of three ways: either through a socket interface, through a JDBC interface, or through a CORBA IDL interface.

3.3.1 HyperWave backend

For our prototype repository we adopted the HyperWave\(^2\) system as our backend database. HyperWave is an advanced Web server at whose core is an object-oriented database system. It has many

\(^2\)Previously called Hyper-G; its main features are introduced in [3]
features that are not found in similar systems and that make it particularly suitable for our purposes, principal among which are:

**Client/server protocol:** Socket communication with the HyperWave server is supported through a dedicated protocol, enabling remote clients to perform operations on HyperWave objects. Connections are persistent, which means that after a connection is established, subsequent operations can be performed very fast.

**Object database:** Objects under the control of HyperWave are stored in an object-oriented database, which is accessible for read and write access by using the client/server protocol.

**Object types:** A number of predefined HyperWave object types exist. Most important for our purposes are the collection object, which serves to group a number of other objects together (similar to a directory), and the generic object, which can contain any type of text or binary data.

**Access control:** Users and user groups can be created and access rights assigned to users and groups in a similar way as with Unix files. This can provide a high degree of security of the data.

**Per-object attributes:** Each HyperWave object possesses a set of attributes describing it, including owner id, date created and last modified, title, keywords, and others. Attributes are indexed by the system, making fast object queries by attribute value possible.

**Linking and replication:** The HyperWave database may contain links to objects located elsewhere in the system, or even on a remote system. These are treated just like local objects, giving location transparency. Furthermore, replicas of remote objects may be created on the local system; whenever HyperWave receives a request for a remote object, it will serve the local replica instead, resulting in quicker access times.

**Server pools:** Groups of HyperWave servers can be configured to operate as a logical union with shared user and group databases, facilitating scalability of the system by adding additional servers.

Initially, after the ICCS repository server starts up it makes a connection to the configured HyperWave server, which it maintains until shut down. For performance reasons, we run the repository server on the same machine as the HyperWave server, because then only one leg of the client-server-backend connection requires a remote connection, with further communication being performed within the server machine.

### 3.3.2 Repository objects

After startup the repository server is ready to accept requests from repository clients. Clients first need to authenticate themselves with userid and password before being allowed access to the server. Regardless of the actual representation of the object in the database server, within the ICCS repository only one representation is used: the repository object (RO). Database server-specific services within the repository server perform the necessary mapping from stored format to RO format.

ICCS repository objects are of two principal types: files and folders, corresponding to files and directories in a file system, or more generally, atomic and composite objects. Each of these are implemented as a HyperWave collection of revisions of the object in question. For example, a repository file object is stored as a collection of revisions of files (of the generic HyperWave object type), while a repository folder object is stored as a collection of revisions of collections.

A repository object consists of two parts: a set of meta-data and a body. The meta-data contains attributes describing the object, such as its name in the client, its internal name in the server, owner id, revision number of an individual revision of the object, revision number of the latest revision of the whole object, lock status etc. The body consists of the actual content of the object, which in the case of a file is its data and in the case of a folder is a list of the objects contained.

### 3.3.3 Client/server communication

Between repository client and repository server, the mode of communication and object transmission is through Java Remote Method Invocations (RMI). After the repository server starts up, it creates an object that handles requests for repository objects. Clients then remotely invoke methods in this object to perform operations on repository objects, such as openFolder, openFile, copyFolder, etc. Since RMI handles the marshaling, low-level transmission and unmarshaling of objects automatically, the whole process is largely transparent to the client, who can treat remote objects and local objects almost the same.

In addition, our implementation of the repository server is multi-threaded, maintaining three main threads: one for handling exchanges of messages with the HyperWave server, and two threads to handle the queueing, sending and receiving of files to and from HyperWave. Each thread uses its
own socket connection, which allows parallel data transmission over three sockets and thus improved performance.

4 Pluggable Software Engineering Environments

As indicated earlier, we consider it important that software engineering environments (SEE) facilitate dynamic reconfiguration in order to provide the best match of support tools to the task or project at hand. Today’s software engineering tools and CASE environments are often monolithic, inflexible, and bloated thus inefficient “mega-tools”. Their aim, whether stated or implicit, often is to provide full support for all software engineering activities involved in a software project. This has proven to be unrealistic, and frequently it is necessary to acquire tools from different vendors and manually integrate them into one environment. However, this can be a non-trivial task, as tools often do not cater for interoperation with other third-party tools, and so extensive development effort may be required to achieve the desired integration. Recent research has been directed at producing meta-CASE environments, i.e. reconfigurable CASE toolsets that permit the tailoring and customization of a CASE environment to suit a specific software development methodology and the requirements of a specific project (see e.g. [19]). At the same time, component architectures have received much attention lately, as they promise to enable code-free program development [35] and software plug-and-play, i.e. enabling systems to be assembled from pre-fabricated components, much like it is done in the electronics industry today [33, 6, 7, 31].

The current direction of ICCS is to combine the areas of component architectures, meta-CASE environments, and mobile code to produce pluggable and mobile software engineering environments.

4.1 Plugging—The Goal

Our goal is to define and develop a framework for interoperation that makes it possible to assemble various software tools with a minimum amount of effort—ideally through visual manipulation of graphical objects on a computer screen, without any conventional programming. This would be accomplished through a builder environment. In this context, we are specifically concerned with the case of coarse-grained components, i.e. components that are software applications in their own right and that may be used on their own. We firstly present some key concepts that we expect a suitable component architecture to support:

Components: Components are units of software that are designed to provide a specific functionality. Two primary types of components may be distinguished: self-contained, independent components that can function on their own; and dependent components that require services from other components (see below).

Connectors: Components can be connected with each other through “connectors”—public interfaces that handle all access to a component.

Services: Components implement certain functionality which they may provide to other components as services. Components can be implemented to use services in three different ways: 1) Components may require specific services from other components. If the service is not available, the component cannot function. 2) Components may require one of a set of services, i.e. at least one member of the set must be available for the component to be functioning. 3) Components may make use of optional services, i.e. the service is not required for the functioning of the component, but it can provide added functionality.

Introspection: A builder environment can introspect components to discover their public interfaces and the set of services they provide, require, or optionally use.

Coupling: Components can be coupled (“plugged together”) if the services required by one are provided by the other.

Selectors: Given the knowledge about the required services of a component, a builder environment’s selector searches for other components with matching interfaces.

A number of component architectures providing or supporting such features are available today, including OpenDoc from Apple and IBM; CORBA from the Object Management Group; COM, DCOM, OLE and ActiveX from Microsoft Corp.; JavaBeans from Sun Microsystems; and others. Because of our chosen mobile code language, Java, it seemed natural for us to adopt JavaBeans as our software component architecture. The following discussion, however, is kept at the general level, avoiding references to JavaBeans-specific concepts and mechanisms where possible (we also continue using the generic term “component” rather than the JavaBeans-specific term “bean”).

4.2 An Example

Let us illustrate the idea of plugging together a development environment through an example. Suppose a simple case where a developer needs only
a text editor, a compiler, and access to the source files. In this case, three components could satisfy the developer’s needs: an editor component, a compiler component, and a repository component. Different possibilities arise how these components may be plugged together, depending on the sophistication of their interfaces and the services provided. For simple tools, the editor and compiler each access the repository, which may be just makes use of an area in the local file system: the editor writes a file and the compiler reads that file. There is no interconnection between editor and compiler—see figure 6(a). On the other hand, where both editor and compiler possess a greater degree of sophistication, they may be able to interoperate more closely. The editor may be able to directly invoke the compiler, passing the program source along without the need to store it in the repository, while the compiler, upon finding an error, may alert the editor about this, positioning the editor’s cursor on the line in the program where the error was found. In this case, editor and compiler make use of each others’ services, and the repository component is only involved for the initial read and final write of the source file—see figure 6(b). In yet another case, editor and compiler may possess a different degree of sophistication. The editor, for example, may provide a service to position the cursor on a line of text, however the compiler is not designed to take advantage of that possibility and the editor’s greater (optional) functionality remains unutilized—this case ends up with a system identical to that in the first case—see figure 6(c).

4.3 Obtaining Component Information

As mentioned earlier, the plugging illustrated in the example just now would in practice be achieved through a builder environment that would have available all the required tools (and more), and would be familiar with the semantics of each tool to support the user in plugging together a meaningful combination of them. It uses introspection to discover the public interfaces of all components. Standard introspection only discovers a component’s general interface characteristics—properties, events and methods supported by it. However, a builder requires more information: namely the capabilities and the requirements of a component. Capabilities refer to “what kind” of component this is, and what services it provides. Requirements refer to “what kind” of other components this one relies on, and what services it needs from them (together with information whether these are required or optional). To obtain this information, the developer of the component must either encode these in the component itself, in its public properties, or may choose to provide a separate class that can make this information available on request to a builder environment. The representation of this information is an important issue—the goal being that a builder environment should automatically discover matching components, i.e. those that can be plugged together, without human intervention. There are two main approaches:

1. Semantic analysis of a class and representation in a semantic net or other suitable notation. A builder would need to decide whether two semantic descriptions are equivalent in order to determine whether the related components are compatible. Techniques to do so exist, but tend to be imprecise and domain-specific [13, 10].

2. Assigning descriptor terms from a controlled dictionary of descriptors. Component developers consult the dictionary to identify descriptors terms matching their components’ capabilities. The advantage is that at introspection time, a builder only needs to look for precise matches of descriptor terms to determine whether components are compatible. The disadvantages are that it may be difficult to create a standard dictionary that will be widely used, and that the procedure of adding new terms may be inflexible, e.g. having to pass a submission and review process by a standards body.

\footnote{This is in the case of JavaBeans.}

\footnote{In JavaBeans, a class implementing the JavaBeans.Beaninfo interface.}
4.4 Adaptors and Wrappers

The above plugging example made a simplification, as we assumed that two components can be plugged directly into each other, meaning that the components can directly interact with each other. However, in practice it would be desirable if different semantically equivalent components could have different class names, without requiring other components that wish to utilize these components to know about their real name. For example, suppose a component wishes to use an editor component whose semantic descriptor simply is editor. A component provider may have several editors that match this descriptor, which are combined in one package; e.g. the package umac.iccs.editors may contain the umac.iccs.editors.viEditor and umac.iccs.editors.emacsEditor. Since a component will just refer to the editor component by its semantic descriptor editor, there is a need to reconcile the different class names at build time. To achieve this, we propose the use of adaptors. An adaptor would be created by the builder on-the-fly at build-time and placed between two components to map generic object references (such as editor) to specific ones (such as umac.iccs.editors.viEditor). Figure 7 illustrates this point on the compiler-editor-repository example shown earlier—the adaptors are placed between the components to be plugged together.

A different situation exists if we wish to integrate external legacy and non-Java tools. Although in this case we would not be able to achieve code mobility, it may be necessary to integrate such tools if they are of purely local use and if there are no equivalent tool components. The main problem in this case is how to obtain the semantic descriptors of the tool in question, and how to enable it to be plugged together with other tools in the builder environment. We propose the use of wrappers. A wrapper is a simple component which in itself does not implement any services, but which serves as a vehicle to access services that the external tool provides. A wrapper would need to be created (i.e. programmed) manually for the tool to be integrated. Although simple tools would only require a simple wrapper, in the case of complex

---

3Related to this, CORBA service specifications are industry standards that are universally used. However, they are "always independent of application domains" [29] and are only updated two or three times a year—not suitable for our purposes.
tools which provide their own APIs, larger effort may be required to implement the wrapper. Many external tools were not designed with interoperation in mind, so the services such tools can offer are necessarily few and simple. A compiler, for example, may only be able to be started, supplying the name of a file to compile, but may not be able to position an editor on the line in the program where an error was found. In this particular case, the wrapper could provide such functionality based on the error message output from the compiler, but such may not always be the case with other tools. This example is illustrated in figure 8 where the external compiler is enclosed in a wrapper, while the editor is a proper component that is plugged into the repository through an adaptor. As mentioned, some external tools provide APIs for application programmers, in which case the wrapper could create services corresponding to API functions and the tool would offer a richer set of services to other tool components.

5 Future Work

In the current state of the ICCS project, the main concepts have been defined and specified: the component architecture and interoperation model, and the repository services. An initial repository has also been implemented in Java, which is already functional. There are, however, several areas in which we plan further work, as follows:

Extended repository services: The repository in its present incarnation has only implementations of the major services in the communication, repository, and access layers. We plan to extend this by adding more services in these layers, and by also implementing several policy layer services, starting with the check-out/check-in services.

Detailed component interaction specifications:
While the main concepts of component interaction have been defined, we still need to specify in greater detail how components can interact through the adaptor and wrapper mechanisms, including: object creation, method invocation, event listener registration and event handling. We also need to define how to implement concurrency control of common resources used by different components.

Implementation of builder environment:
Current component builder tools, such as the JavaBeans Beanbox, do not have the ability to inspect service capabilities and requirements in the way we require, and to assist the user in finding matching components. Therefore, we need to implement a builder environment that can provide this functionality.

Definition of semantic descriptors: We need to develop a suitable scheme of semantic descriptors for components and their services, and to define an initial set of descriptor to apply in our prototypes. Questions to investigate are whether to support inheritance of descriptor terms, and if so what kind of inheritance would be appropriate.

Implementation of tool components: So far, the repository is the only component implementation. In order to test our ideas, we need to create more software tools. We may start by collecting existing tools and converting them to ICCS components, or in the case of non-Java tools, to write wrappers for them.

6 Conclusion

This paper has presented an overview of the ICCS project, which is currently underway at the University of Macau and whose objectives are to study principles and to develop prototypes of systems that can facilitate distributed cooperative software engineering.

Thus far the project has produced specifications for a component-based software engineering metamodel, which can aid a software developer in plugging together a set of tool components that can provide a match to the requirements of a given project or task. A central and important component to support distributed developers sharing their objects of work, a repository, has been specified and implemented. Currently, work is proceeding on extending the functionality of the repository and
on starting the implementation of the component builder environment.

We believe that in the future, distributed collaboration will be commonplace not only in software engineering, but also in other disciplines. As the ideas presented in this paper are not limited to the domain of software engineering, they should also be of interest to researchers in other areas.

Acknowledgments

This work was sponsored in part by the University of Macau Research Committee under research grants RC004(96/97) and RC063(96/97). The support is gratefully acknowledged.

Thanks also for the hard work of Tou Chi Pio (Bryan), our local Java expert, who implemented the ICCS repository.

References


Robert P. Biuk-Aghai is a lecturer in the Software Engineering Programme of the Faculty of Science and Technology, University of Macau, where he has been working for the past four years. He is also a group leader, project manager and researcher with INESC-Macao. Prior to coming to Macau, he was a member of the corporate database group of a large bank in Frankfurt, Germany. He received an M.Sc. degree in Information Systems from the University of London, U.K. in 1990, and is member of the IEEE Computer Society and the ACM.