Component-Based Groupware Tailorability using Monitoring Facilities

Cléver Ricardo Guareis de Farias, Nikolay Diakov

Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{farias, diakov}@cs.utwente.nl

ABSTRACT

Tailorability has long been recognised as a key issue concerning groupware applications in general and component-based groupware applications in particular. Tailoring activities are usually classified according to three levels, viz., customisation, integration and extension. This paper presents an approach to component-based tailoring based on the use of monitoring extensions. Our approach allows the extension and integration of new components into a legacy groupware application without the need for changes in the existing components.

Keywords

Software components, groupware, tailorability, monitoring framework, middleware.

INTRODUCTION

One of the characteristics of Computer Supported Cooperative Work (CSCW) is the diversity of situations involved in an everyday project. For example, within the same project, the members of a group can combine and alternate between synchronous and asynchronous work, centralised and distributed work and collaborative and individual work. This is reflected in a variety of groupware applications that are used to provide support to cooperative work, each one usually designed having a specific situation in mind.

It is also very unlikely that an application targeted to a group of users will equally satisfy everyone's preferences and needs. Some contingencies typical of cooperative work, such as a group's individual preferences, different modes of collaboration, the opportunistic nature of work and different contexts of use, contribute to make this situation worse.

We argue that likely there will not be a single groupware application capable of providing the necessary support to a typical cooperative work situation. Instead, we expect that a typical groupware system will consist of a set of individual applications or applications components that can be combined to provide an integrated support.

Tailoring is defined as the activity of modifying a computer application within its context of use [8]. Tailoring takes place after the original design and implementation of the application; it can start during or right after the installation of the application. Tailoring is usually carried out by individual users, local developers, helpdesk staff or groups of users. The level of tailorability varies a lot, from simple user interface tailoring to more sophisticated forms involving the provision of a complete new set of functionality, which is commonly regarded as radical tailoring [10]. In the literature [11], different levels of tailorability are defined:

- customisation, which deals with the selection of configuration options among a predefined set. Examples of customisation in groupware include the selection of different floor control policies among the users of a shared whiteboard or the selection of individual interface preferences;
- integration, which deals with the composition of a set of predefined components into one unified application. The composition of the components of a groupware toolkit is an example of integration;
- extension, which deals with the addition of functionality to an application and its components by adding new program code. Examples of extension include the implementation of a new floor control policy in a shared whiteboard. Scripting languages are often used for both the integration of components and the extension of existing code without the need for recompilation.

Two basic properties that a groupware application must have in order to provide tailorability are extensibility and composability [6]. Extensibility represents the capacity of adding new functions without interfering with existing ones, while composability represents the capacity of composing a function by selecting and combining more basic functions.

Stiemerling and Cremers [12] point out three different forms in which a CSCW system can be tailorable using components, viz.: changing the parameters of single components, changing the composition of components and changing or extending the implementation of components. These forms of tailorability keep some correspondence with the levels of tailorability defined previously.

Most of the work on groupware tailorability concentrates on second level of tailorability, i.e., the integration or combination of components to provide the desired functionality, c.f. [15,17]. A possible explanation for this phenomenon can be drawn based on the amount of skills required from an application user in order to tailor this application according to the different levels of tailorability. As observed

in [9], there is a large gap between the skills required to tailor an application by customisation (less skills) and by extension (more skills). Tailoring an application by integration requires an intermediate level of skills.

This paper presents an approach for component-based tailorability that concentrates on integration and extension of legacy groupware applications. Collecting information can play a significant role in the process of extending the functionality of a CSCW system. Information about the current configuration of application components, about their communication behaviour and lifecycle, is often a significant part of the input to a software extension. Although component technology is capable of providing information about component state and the service they offer, little information is available at runtime about the behaviour of the instances of application components. A flexible monitoring framework similar to [1] can provide such monitoring facilities. Since this approach relies on observation, it is complementary to the existing intrusive techniques for extensibility and composability that can be used to implement new functionality in a distributed CSCW system.

The remaining of this paper is structured as follows. In the following section some concepts regarding component-based groupware are introduced. Then, in the next section our monitoring-based tailoring mechanism is presented. Subsequently, some related work concerning component-based tailorability is discussed. Finally, in the last section some conclusions are drawn and future work is outlined.

COMPONENT-BASED GROUPWARE

Most contemporary consumer products are modular, often assembled of components manufactured by different vendors. The end product is often deployed to meet particular regional needs and other factors that may influence the success of a product on the global market. Combining such diverse modules together would be impossible without compatibility requirements outlined in international or company standards for a family of products.

The experience from the consumer products industries is quickly entering the dynamic world of software industry, resulting into a move towards development of new solutions for component-based software design and implementation.

Component-based groupware development has gained increasing support in the past few years (see for example [6]). This discipline aims at constructing groupware systems and applications by assembling prefabricated, configurable and independently evolving building blocks, called software components.

There are several definitions for a software component. However, in this work we adopt a definition based on Szyperski's component definition [16]. We designate a software component a self-contained and reusable binary unit that provide a unique service and can be used either individually or in composition with the service provided by other components.

The development of a component-based application is usually independent from tailoring activities, since tailoring is performed after the design and implementation phases. However, depending on the mechanisms and techniques that will be adopted afterwards to tailor the application its design and implementation may have a significant impact.

A methodology for designing component-based groupware applications is presented in [5]. The main elements of this methodology are abstraction levels and views. According to this methodology the development of an application is split into separate abstraction levels, viz., enterprise, system, component and object. At each level different views are used to capture structural and behavioural aspects of the application.

This methodology is independent of any particular component model that can be used to implement the components, such as Java Beans, Enterprise Java Beans, CORBA and DCOM. So is our approach for tailoring by extension presented in the next section. A deeper discussion concerning this methodology is beyond the scope of this paper.

Components interact with each other through contractually specified interfaces. A contract, such as the OMG IDL specification, establishes a set of services (IDL operations) that one of the entities explicitly provides (implements) and the other entity uses (invokes). A middleware provides the necessary support for component interaction (see illustration in Figure 1).

After assembling a groupware application based on a set of available components to comply with the desired user requirements, the application becomes a "legacy" application. Nevertheless, the user needs may change and new functionality may be required from the application. Depending on the functional requirements, adding a new software extension to the application may require changes in the components and even replacement of components. These changes often lead to high development costs. Further, adding a new functionality to a legacy component may jeopardize its understanding and its combined use in other applications.

In order to solve these problems, we propose the use of a new class of software extensions, called monitoring extensions, that perform observations over the behaviour of the application components in a groupware application. If general monitoring support is introduced into every original application component and the communication middleware responsible for the distribution transparency, the developer of the monitoring extensions does not need to change the existing application in order to implement the observation functionality for this extension.

Figure 1 illustrates the use of monitoring extensions to extend a component-based application. The use of monitoring extensions does not require changes be made in the existing components to provide the required functionality.

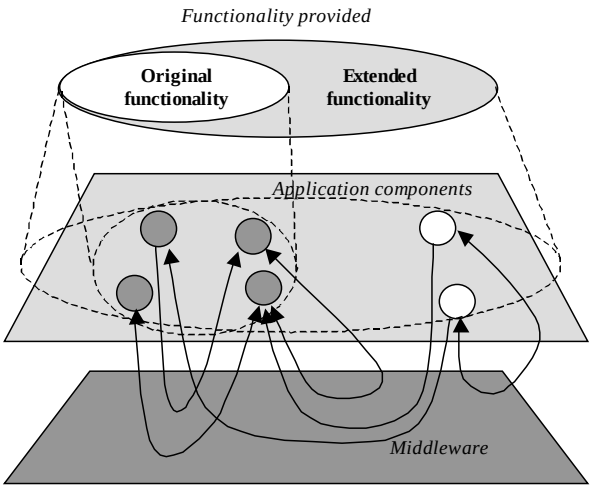

Figure 1. Components and component extensions.

A MONITORING APPROACH

We will construct a generic scenario for building typical extension to a legacy distributed software system. Since CSCW applications in general can be characterised as distributed software systems (often component-based), we will be able to apply the generic scenario for these applications as well.

Let us assume that a legacy application is being modified to facilitate a number of new functionalities. The application is build of a number of components. In the ideal situation the implementation of the legacy components would not need to be changed at all. Such solution would achieve the goal by applying customisation of the component parameters and by integrating several new components implementing the new functionality. The reality however, dictates that such seamless and cheap solutions are very limited to the variety of functionalities they can implement, which motivated us to is to look for a different approach.

We distinguish three phases in the runtime operation of a software extension:

- Phase 1 - Observation. The new functionalities in the software extension require information about the state of the legacy components. This information may involve internal as well as external state of these components, event notifications about state changes, communication behaviour, lifecycle behaviour, etc;
- Phase 2 - Reasoning. Upon receiving information from the legacy components, the software extension analyses it and changes its internal state, typically meaning "decision is being made";
- Phase 3 - Control. The software extension achieves an internal state that requires manipulation on the state of legacy components;

The execution of a software extension typically follows cyclic repetition of the three phases.

We model the behaviour of a monitoring extension using these three phases, thus each activity performed by the extension can be related to observation, reasoning or reaction.

Our approach is based on a monitoring framework described in [1] that provides extensive support for phase 1 type functionalities. The services of the monitoring framework can be considered part of the communication middleware that handles the interaction between the application components.

The monitoring framework provides generic and configurable services for observing the communication and lifecycle behaviour of the legacy application components. The extension components can configure and use these services. The monitoring framework also provides runtime support for configuring the granularity and type of the information coming from the legacy application components (see Figure 2). The information is then collected from the monitoring framework and is delivered to the components of the monitoring software extension.

Naturally, a question arises whether this information is sufficient to implement arbitrary functionality in the software extension. The model (format) of the monitored information is very close to the programming model of the middleware technology used to hide distribution transparencies, and the programming model of the component model used to implement the distributed application. For example, the monitoring framework in the legacy application can be reconfigured dynamically to start monitoring on a particular set of operations from an interface of a particular legacy component. When this is done, monitoring information will be collected by the framework at the rate of invocation of these operations, containing all the information regarding the invocations as parameter values, state, etc. This information is encapsulated into asynchronous (or synchronous[1]) events and is delivered through the monitoring facilities to any interested parties that can further perform phase 2 and 3 functionalities. There are a number of crucial issues like the overhead of the monitoring framework to the legacy application components, the order of the events (execution order vs. receiving order), synchronous and asynchronous events (i.e. the execution of the action that is responsible for the event notification is executed synchronously or asynchronously with the delivery and processing of the event to all interested parties), reliability of the delivery of the events. These issues are discussed in separate research papers [1,2].

[1] The latest design of the monitoring framework allows synchronous notifications. In other words, the execution at the source of the monitoring information is suspended until the designated event receiver analyses and decides for a reaction to this notification. The execution then can be resumed (explicitly) and the legacy application component continues its normal operation.

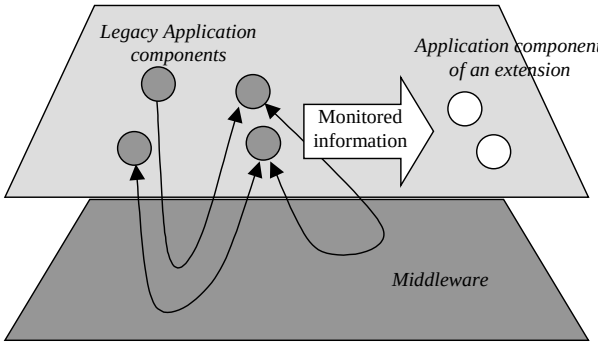

Figure 2. Observation phase and the flow of the monitored information.

During the phase 2 of the execution cycle the information coming from the monitoring framework must somehow be interpreted, and then the resulting reaction carried out in phase 3 type functionality. Our monitoring approach does not provide special support for phase 2 activities however, since the interpretation of the information is done here, we offer a scheme for doing it (see Figure 3). A static mapping translates the information model of the monitoring framework into the information model of the domain where the functionalities of the software extensions lie.

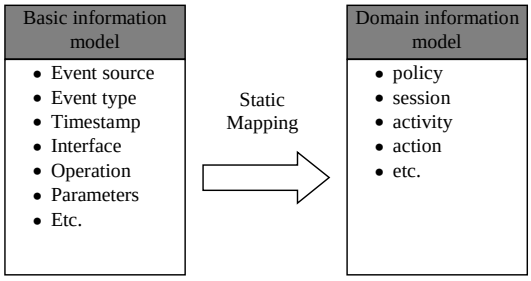

Figure 3. Reasoning guidelines. The step from the basic monitoring information toward domain information that makes sense.

Once a static mapping has been outlined, the designer of the phase 2 functionalities can define the finite state machine of the decision-making process.

Phase 3 is where reaction is conducted according to the decisions made during phase 2 (see Figure 4). Reactions may vary from calling operations on interfaces of the legacy application components, to instantiating/destroying instances of components, and other activities. Since the monitoring framework provides only observation on the application behavior, other, intrusive techniques must be used to inject phase 3 functionality into the software extension, such as techniques employing component tailoring [7, 15, 17].

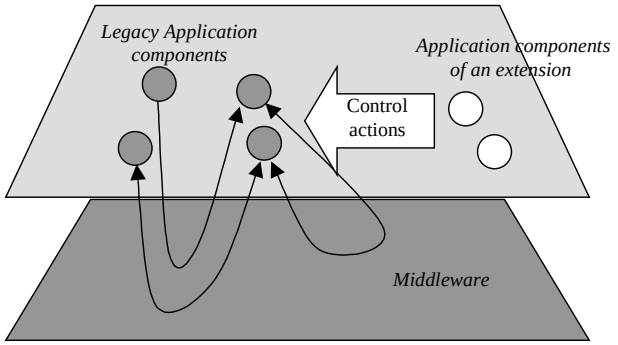

Figure 4. Control phase with the reaction from the extension component towards the components of the legacy application.

RELATED WORK

Hummes and Merialdo present an approach for component-based tailoring by extension [7] based on the extensibility pattern. This pattern allows one to make some changes at run-time, provided that these changes conform to an interface. Some existing behaviour can be replaced or some new behaviour can be added at certain points called “hot spots”. These hot spots must be discovered in the design phase and then implemented according to the pattern.

The extensibility pattern is implemented using the Java Beans component model. This allows one user to insert some new piece of code in a groupware application and distribute this code among the other users. Since Java is an interpreted language, the new code can be immediately executed without the need for recompilation of the application. However, due to the limitations of Java Beans event notification mechanisms, such mechanism had to be extended in order to allow the distribution of events across different virtual machines.

Syri uses another pattern, the mediator pattern, to provide component-based tailorability [15]. There are three basic object types according to this pattern, viz., target objects, enablers and mediators. Target objects provide the intended cooperation support. Enabler objects encapsulate functionality for basic cooperation support, such as communication, coordination and sharing, and provide this functionality to target objects. The interactions between a target object and its associated enablers are mediated by a mediator object.

This approach provides a “fine-tuning” of the functionality provided by the components by decoupling the interactions between a target object and its respective enablers. Therefore, tailorability is achieved in three different ways: a user can replace the behaviour of a target object, add or remove enablers to/from a target object and customise the methods of the enablers themselves.

Teege proposes yet another approach to achieve component-based tailoring, particularly tailoring by composition [17]. This approach concentrates on the use of features and parts. A feature represents a system component (part)

whose properties or functions can be integrated with other components by simply selecting its presence in a set. This characteristic is regarded as pure integration.

Teege argues that feature composition presents some unique characteristics that made it suitable for composition. For example, features can be added in any order and the use of each feature should be independent of the use of other features; however, features can be related. In feature composition the perception is of the whole, not the single parts. Therefore, any feature composition can be abstracted into a single feature.

The system should be decomposed in both parts and features. First the system is decomposed into parts. Then those parts are further decomposed into features. Every feature consists of a description and an implementation. Based on the feature's description a user may select different features for composing his application, each feature representing a different functional property.

Stiemerling et al. propose another approach for component-based groupware tailoring in [13,14]. According to this approach, components are implemented using Flexibeans, an extension to the Java Beans component model. Flexibeans extends Java Beans with the concepts of named ports, shared objects and remote interactions. Component compositions are described using the so-called CAT-files. CAT-files describe the interconnection of components locally, i.e., components located in a same machine. The interconnection of components located in different machines is described using remote-bind files.

The deployment and tailoring of the applications is supported by a component-based platform, called EVOLVE. This platform provides the necessary support for tailoring by changing at run-time the composition of the components that form the application.

This solution relies on many proprietary designs and implementation (EVOLVE, Flexibeans, CAT-files). There is a number of international standards like CORBA Components and EJB that offer similar functionality. That is why we feel this approach has to be revisited and updated on technological level.

In [3] a multi-layer monitoring framework is proposed supporting distributed environments. Under certain conditions this framework can be used for building software extensions however, it is not flexible enough and does not provide room for extending its reliability with respect of the order of events.

Incorporation of reflective middleware in a project can offer good support to future software extensions. DynamicTAO [4] is a good candidate because it includes facilities that enable implementation of a monitoring framework and is build around a standard CORBA-compliant request broker. Furthermore, it provides facilities for adaptive changes in the communication and security strategies and on-the-fly safe reconfiguration.

Nevertheless, the support for monitoring in dynamicTAO is rudimentary and needs to be significantly extended.

CONCLUSION

We propose an approach for component-based tailoring based on the use of a monitoring framework. The monitoring framework is generic enough to support the majority of the activities categorized as observation of the behavior of a distributed application. In particular, having such framework will reduce the cost for implementing notification hooks by opening the code of legacy application components making them report information to the components of the software extension.

Our approach provides the means to extend and integrate a groupware application through component extensions. However, the design of the new components is not prescribed. This activity is carried out separately using, for example, a component-based groupware development methodology as proposed in [5]. The proposed approach is also independent from any particular component model.

We offer a basic guideline for design and implementation of functionalities involving reasoning about the information received from the monitoring framework. This way the basic information model would effectively be mapped into the domain model of the software extension, where decision can be made for reaction towards the components of the legacy system. When reaction has to be performed on the legacy application components, we advise the designers to select from a set of available solutions for tailoring the reaction functionality with the existing groupware application and/or supporting platform.

Our monitoring framework has been applied during implementation of a generic distributed debugger in the FRIENDS project [18]. Another application of the framework is planned within the AMIDST project [19] where monitoring information will be used to trigger Quality of Service related control cycles.

We also intend to further investigate the static mapping design guideline between the basic information model and the domain information model, hopefully providing some aid for groupware domain mappings in particular. More detailed mappings can not be provided since every mapping is domain and application dependent and thus unique.

We are currently working on a prototype of a generic monitoring framework that can supports wide variety of distributed applications, including: distributed debuggers, automatic conformance verification tools (conformance of a prototype to its process model), management tools (performance analysis, load-balancing, administration and logging), and groupware applications.

REFERENCES

1. Diakov, N., van Sinderen, M. and Quartel, D.: Monitoring extensions for component-based distributed soft-

ware. Accepted for the *5th International Conference on Protocols for Multimedia Systems (PROMS 2000)*. Cracow (Poland), October/2000.

2. Diakov, N.K., Batteram, H. J., Zandbelt, H., Sinderen, M. J., " Design and Implementation of a Framework for Monitoring Distributed Component Interactions", accepted for the *"Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services" (IDMS 2000)*, October/2000, Enschede, NL.
3. Rackl, G., Lindermeier, M., Rudorfer, M., Süss, B. "MIMO --- An Infrastructure for Monitoring and Managing Distributed Middleware Environments", *Middleware 2000 --- IFIP/ACM International Conference on Distributed Systems Platforms*, volume 1795 of *Lecture Notes in Computer Science*, pages 71-87. Springer, April 2000.
4. Kon, F., Román, M., Ping Liu, Mao, J., Yamane, T., Magalhães, L.C., Campbell, R.H., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000
5. Guareis de Farias, C.R., Ferreira Pires, L. and van Sinderen, M.: A component-based groupware development methodology. *Proceedings of the 4th International Enterprise Distributed Object Computing Conference*, Makuhari, Japan, pp. 204-213, September/2000.
6. ter Hofte, G. H.: *Working Apart Together: Foundations for component groupware*. PhD Thesis, Telematics Institute, Enschede, the Netherlands, 1998.
7. Hummes, J. and Merialdo, B.: Design of Extensible Component-Based Groupware. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 53-74, 2000.
8. Kahler, H., Mørch, A., Stiemerling O. and Wulf, V.: Introduction of the Special Issue on Tailorable Systems and Cooperative Work. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 1-4, 2000.
9. MacLean, A., Carter, K., Lövstrand, L. and Moran, T.: User-tailorable systems: pressing the issues with buttons. *Proceedings of the 1990 Conference on Human Factors and Computing Systems (CHI'90)*, pp. 175-182, 1990.
10. Malone, T.W., Lai, K.-Y. and Fry, C.: Experiments with Oval: a radically tailorable tool for cooperative work. *ACM Transactions on Information Systems*, 13 (2), pp. 177-205, 1995.
11. Mørch, A.: Three Levels of End-User Tailoring: Customization, Integration, and Extension. In M. Kyng and L. Mathiassen (eds.): *Computers and Design in Context*, Cambridge, MA: The MIT Press, pp. 51-76, 1997.
12. Stiemerling, O. and Cremers, A. B.: Tailorable Component Architectures for CSCW-Systems. *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Programming*, Madrid (Spain), pp. 302-308, 1998.
13. Stiemerling, O., Hinken, R. and Cremers, A. B.: Distributed Component-Based Tailorability for CSCW applications. *Proceedings of the Fourth International Symposium on Autonomous Decentralized Systems (ISADS '99)*, IEEE Press, Tokyo (Japan), pp. 345-352, 1999.
14. Stiemerling, O., Hinken, R. and Cremers, A. B.: The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware. *Proceedings of Third International Enterprise Distributed Object Computing Conference (EDOC'99)*, IEEE Press, Mannheim (Germany), pp. 106-115, 1999.
15. Syri, A.: Tailoring Cooperation Support through Mediators. In *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work (ECSCW '97)*, pp. 157-172, 1997.
16. Szyperski, C.: *Component software: beyond object-oriented programming*, Addison-Wesley, USA, 1998.
17. Teege, G.: Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems. In *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9 (1), pp. 101-122, 2000.
18. The FRIENDS project, see http://friends.gigaport.nl/
19. The AMIDST project, see http://amidst.ctit.utwente.nl/