Available online at www.sciencedirect.com

SCIENCE DIRECT

Knowledge-Based Systems 17 (2004) 303–310

Knowledge-Based SYSTEMS

www.elsevier.com/locate/knosys

High-level design for user and component interfaces

Gregor v. Bochmann*

School of Information Technology and Engineering, University of Ottawa, 800 King Edward Ave, Room 5082, P.O. Box 450, Stn A, Ottawa, Ont., Canada K1N 6N5

Received 21 June 2004; accepted 9 July 2004
Available online 7 October 2004

Abstract

Component-based software architecture is very important for current software engineering practice because (a) it is the basis for re-use of software at the component level, and (b) in distributed systems, the physical distribution of an application over separate computers represents a decomposition of the application. Typical e-commerce applications consist of various components sometimes belonging to different organizations, and presenting different user interfaces to various categories of users. We review in this paper the current trend in standards for inter-component communication in distributed systems, including various forms of remote procedure calls (RPC) and message passing, and paradigms for describing and implementing user interfaces in the Web environment. We discuss whether the user interface can also be described, at an abstract level, by RPC primitives. In the second part of the paper, we discuss the importance of indicating which party is responsible for making certain decisions for selecting control flow alternatives and certain parameter values. This leads to some guidelines for describing system behavior scenarios at the requirements level. We also discuss how this approach can be integrated with screen-oriented behavior definitions.

Keywords: Component-based development; Screen-oriented interface; Requirements modeling

1. Introduction

Complex computer systems are usually built as a composition of several components. There are two main reason for decomposing a system into components: (a) the whole system may be easier to understand if it is described as a composition of several components where each of these components has a comparably simple structure and behavior; and (b) the system may be implemented as a distributed system, that is, different parts of the system run on different computers which communicate with one another through a telecommunications network. The geographical distribution of different system functions is often dictated by the requirements. Furthermore, during the design phase, a system is often decomposed into a large number of separate components, and the allocation of these components to computers located at different geographical locations may be one of the subsequent design decisions.

It is clear that the different system components must communicate with one another in order to provide a system that satisfies the user requirements. In order to define the behavior of each component independently of the other components, one has to establish well-defined interfaces for communication. The primitives used for this communication should be implementable locally within the same computer as well as over distance using networking protocols. There are two basic communication primitives that can be used here: asynchronous message passing and (synchronous) remote procedure call (RPC).

In the first part of this paper, we give a review of the major technologies proposed for the realization of RPC communication within distributed systems, such as CORBA, Java RMI and SOAP. While they have essentially the same control structure, namely a procedure call, they employ different encoding schemes for transmitting the input and result parameters of the procedure calls. We also discuss related directory structures for finding service objects in distributed systems, and available software development tools and infrastructures. We note that

* Tel.: +1 613 562 5800x6205; fax: +1 613 562 5664.
E-mail address: bochmann@site.uottawa.ca.

asynchronous message passing may be considered a simplified version of RPC were the caller does not wait for the return of the procedure call.

Then we discuss whether the RPC paradigm is not only suitable for describing component-to-component interactions, but also interactions with a (human) user. In fact, it turns out that there are two paradigms that could be applied: (1) the user as caller, and (2) the user as being called. We also consider the concept of a Servlet, which is often used for designing user interfaces in the context of Web services. We then discuss how more complex user interaction patterns, possibly represented as Use Case Maps (UCM) [1] could be translated into implementations of corresponding user interfaces.

In the last part of the paper, we then discuss the importance of indicating which party (or system component) is responsible for making decisions for selecting control flow alternatives and choosing parameter values. This leads us to propose some guidelines for describing behavior scenarios at the requirements level. Our approach is in line with screen-oriented requirement specifications as proposed in the literature. The described methodology can be used for describing the global system behavior as viewed from the user perspective, as well as for describing the behavior of a particular system component, as viewed through the interactions with its environment. It applies to the description of workflow systems involving many system components, as well as to monolithic systems as seen through the user interface. Some implementation issues are also discussed.

The first part of this paper is based on a conference presentation [3]. This work is inspired by the screen-oriented approach to requirement specification as described in [4,10,11].

2. Standards for inter-component communication in distributed systems

The concept of procedure call is a basic tool for abstraction in software engineering. It is also the basic communication paradigm in object-oriented system design, since it represents the execution of a method on an object instance. The latter plays the role of a server providing the service identified by the method name. The details of how the service is provided, i.e. the body of the method implementation, is not of interest at this level of abstraction. However, the object instance providing the service may have an internal state that may determine the result provided by the execution of some service methods, and may also be changed during the execution of some of these methods. It is important to document such state dependencies, because the result of a method call may then depend on previous message calls and their parameters. Unfortunately, there is no standard way to document such dependencies.

While originally the called object and the calling process were usually implemented within the same program executable, it soon became obvious that the procedure call mechanism can be adapted to the situation where the called object resides in a different computer. This leads to the concept of Remote Procedure Call (RPC), which was introduced in the 1980s. In the case of an RPC, the information concerning the method name and input parameters must be passed in the form of a message to the site of the called server object, and the results of the operation must be passed back to the calling party. This requires a well-defined protocol for exchanging this information between the two parties.

Different RPC protocols have been defined and implemented. Such a protocol needs a reliable message transmission medium (for instance TCP or a secure session could be used for this purpose) and must foresee at least two protocol messages:

(1) The RPC request which contains the following information:
    (a) Identification of the server object that should execute the procedure call. *Note*: in the case that TCP is used as underlying transport protocol, the IP address identifies already the computer on which the object resides; the port number used by TCP may identify an application process in that computer which understands the RPC protocol; the RPC request message sent to this application should include enough information to identify the server object within the context of that application.
    (b) The name of the method (procedure) to be called.
    (c) The values of the input parameters.
(2) After the procedure is executed, an RPC response message is returned which must include the result parameter(s) of the operation and any exceptional conditions that may have occurred.

While the nature of the information exchanged in an RPC protocol is always similar to what is said above, the way this information is encoded depends very much on the particular protocol. In fact, the different RPC protocols in use today are usually associated with a number of support tools that are provided as part of a distributed processing environment. The most important environments are the following:

(1) The IIOP protocol used in the CORBA environment,
(2) Remote Method Invocation (RMI) provided in the Java environment,
(3) SOAP, often used for e-commerce applications, possibly using WebServices.

CORBA (see for instance [2]) was designed to realize RPC for object-oriented systems where the software components within the different computers could be written in different languages (language heterogeneity). It is based on the standardization effort on Open Distributed Processing (in the early 1990s), which had the aim of facilitating the interworking between heterogeneous systems, and considered that the early system design stages should be transparent to several dimensions, including the physical distribution of the different system functions (see for instance in [16]).

CORBA introduced the notation of an Interface Definition Language (IDL) for writing abstract class interfaces. Such interface definitions can be automatically translated into equivalent interfaces in the programming language that is used for the implementation of one of the CORBA component implementations, such as C++ or Java.

In this context, the concepts of stub and skeleton became popular. The stub (sometimes called proxy) is an object that represents the remote server object in the context of the calling party. It accepts local calls of the methods defined for the server, but instead of performing the operation locally, it sends an RPC request message to the server object and waits for the corresponding RPC response message and presents the results to the caller through the local procedure return mechanism. The skeleton is the software in the remote site that accepts the RPC request message, decodes the message and prepares a local method call on the server object. The skeleton also receives the results from the server object and encodes them into the RPC response message.

Java RMI (see for instance the SUN tutorial on the Web) is similar to CORBA RPC, except that it assumes that both, calling and called party are implemented in Java.

SOAP (Simple Object Access Protocol) was developed by the W3C consortium and uses the XML standards for the encoding of the RPC request and response messages (see for instance http://www.w3.org/TR/SOAP/). In contrast to the above two protocols, the XML encoding results in messages that can be understood by an (expert) human reader, however, the encoding is less compact. The format of XML messages can be specified in the form of an XML Schema (or a so-called DTD). Consequently, the interface provided by a server object through the SOAP protocol is normally described by such a schema.

SOAP does not use TCP as the underlying transport mechanism. Since it evolved in the context of Web applications, the Web server access protocol HTTP (which runs on top of TCP) was adopted for this purpose. This also means that the server object instance to be called is identified by an URL, which includes the host name, which is used to derive the IP address of the Web server. One advantage of using the Web access protocol is that it is less affected by security firewalls than TCP connections using different port numbers. Therefore, SOAP and XML technologies are used for the development of the so-called Web Services (see for instance http://www.w3.org/2002/ws/) and Grid Applications [12].

As mentioned earlier, various tools and platforms are available for using these different RPC environments. For instance, the CORBA environment provides for translators that translate RPC interface specifications written in IDL into corresponding interface definitions in the implementation language (e.g. C++ or Java) and also automatically produce the code for the corresponding stubs and skeletons. In the case of Java RMI, the RPC interface definition is directly written in Java, therefore not requiring any translation. The code for stubs and skeletons is also automatically obtained, at least for simple parameter types.

Certain implementation support environments for SOAP, such as [12], provide the automatic generation of the XML interface schema from the interface definition given in the Java programming language. This is very convenient when the distributed application is written in Java. In addition, the encoding and decoding functions included in the stubs and skeletons are also automatically generated.

Another important aspect of these distributed computing environments are the directories that allow a calling party to find appropriate server objects within the distributed environment. In the case of the CORBA and Java RMI directories, the server objects register themselves in a directory under a given name. The name, as well as the directory where this name is registered, must be known to the searching party. More sophisticated directories are provided by the Java Jini environment (an extension of Java RMI, see for instance SUN documentation) which provides a so-called Lookup service where a searching party can find a registered server object by providing as search parameters the type of interface offered by the server and possibly values of certain characteristic attributes of the service object that were specified during the registration. Similar directories are also planned in the context of Web Services and Grid applications [6]. The Web Service Description Language (WSDL), which is basically an XML Schema, can be used to define the interface provided by a given service.

3. Paradigms for defining user interfaces

As discussed in the previous section, the communication between different system components is usually organized as sequences of method calls, in the distributed context, remote procedure calls or message transmissions. In this section we discuss whether these communication primitives are also natural units of communication for the interactions between a user and the system. We consider first a simple example to make our discussion more concrete. We then consider three approaches to structuring the user interactions: (a) command language interface; (b) screen-oriented interface; and (c) Servlets.

3.1. An example application: room booking

This room booking application, originally presented in [13], uses a database that contains information about hotel

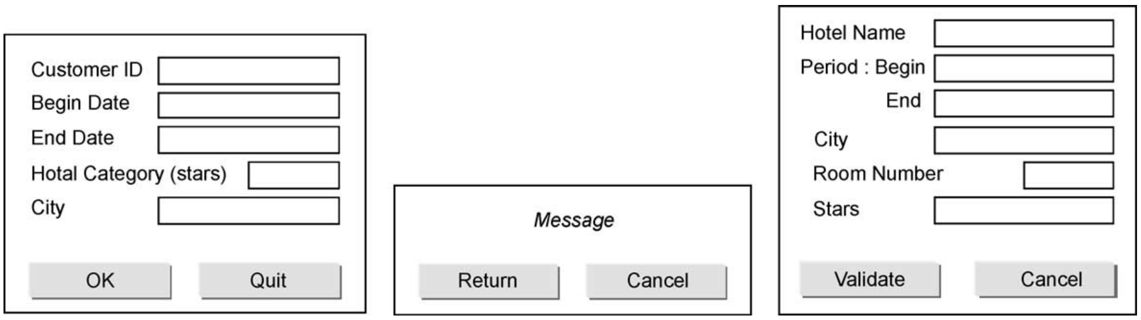

Fig. 1. GUI for hotel booking application (from [13]): (a) initial screen, (b) message window, (c) validation.

rooms available in different cities. It interfaces with the user and lets a user select a room for a certain period and make a reservation. A sketch of the user interface is provided in Fig. 1(a)–(c). Fig. 1(a) shows the initial screen where all the defined fields are input by the user. After pushing the 'OK' button, the user sees either the screen of Fig. 1(b) (where the *Message* may either read 'The customer xxx does not exist in the database' or 'No room is available for yyy' and xxx is the customer identifier and yyy is the customer name) or the screen of Fig. 1(c), where all fields are output by the system. The user may then confirm the reservation by pushing the 'Validate' button, or cancel the reservation and come back to the initial screen.

The dynamic behavior of the Room Booking system can be described by the Activity Diagram shown in Fig. 2. The activity *Prepare booking* displays first the initial screen of Fig. 1(a) and lets the user fill in the fields. The four subsequent actions correspond to the following four cases:

(1) The user pushes the *Quit* button.
(2) The customer identifier is not in the database.
(3) No suitable room is available.
(4) A room is available and may be reserved.

Case (4) leads to the activity *Confirm booking*, which displays the screen of Fig. 1(c) and lets the user validate the reservation or go back to the initial screen.

We note that, in this example, each activity represents a screen of the user interface, and has the same generic sequence of interactions with the user: first a screen is presented to the user including values in certain information fields, and then the user may enter data into certain fields and push one of the buttons.

We also note that this activity diagram describes the behavior of the whole Room Booking system (including the database). In the original description of the Room Booking application in [13], the user interaction sequences were not defined by an Activity Diagrams, as shown in Fig. 2, but through a so-called Process Route Diagram (PRD) according to the Lyee methodology [11], and the application had an explicit interface with a database supporting SQL queries.

3.2. Command language interface

With a command language interface, the user types a command, waits for a response from the system, and then writes the next command. In the simplest case, each command formulated by the user corresponds to the name of a method provided by the system, which plays the role of a passive object that accepts a certain number of method calls in the form of user commands. In the simplest case, the user will type the name of the command (name of method to be invoked), and the values of the required input parameters of the method. Since the user waits for the answer, this sequence of sending the request and receiving the response from the system is logically equivalent to an RPC where the user is the calling party while the system is the server. This situation is therefore the same as for inter-component communication.

In the case of our example system, a simple interaction scenario, corresponding to two RPCs, may have the following form:

RPC1(a)–The user types: PrepareBooking (cust_id, begin_date, end_date, category, city)

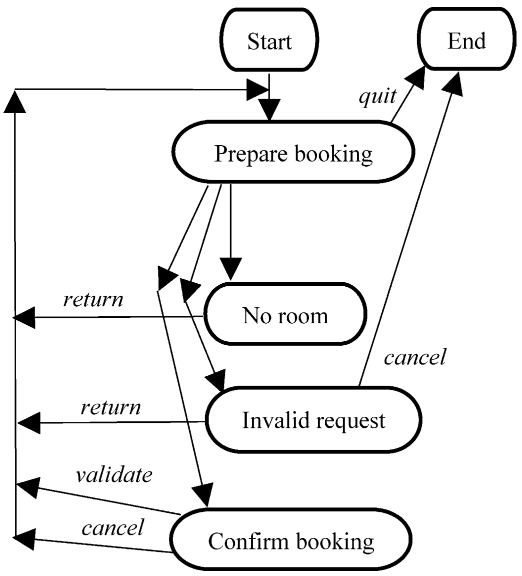

Fig. 2. Activity diagram for the hotel booking application.

RPC1(b)–The system responds: Booking candidate (hotel_name, room_number, stars)
RPC2(a)–The user types: ConfirmLastBooking
RPC2(b)–The system responds: OK

3.3. Screen-oriented interface

With a screen-oriented GUI, one would like to consider a screen as a basic unit of interactions. This is an approach proposed in many papers, e.g. [4,10,11]. While in the command language interface, each unit of interaction (each RPC) is realized by two messages, first from the user to the system, and then from the system to the user, in the case of a screen-oriented interface, the order of the messages involved are in the opposite order: first the screen display content is sent by the system to the browser (which represents the user as far as the application running in the server is concerned), and then the user fills in some data and clicks a button which results in the sending of a message to the application.

For the example discussed above, we would have the sequence:

Screen 1(a)–System sends: initial screen (see Fig. 1(a))
Screen 1(b)–User sends: cust_id, begin_date, end_date, category, city
Screen 2(a)–System sends: validation screen (see Fig. 1(c))
Screen 2(b)–User sends: click-Validate
Screen 3(a)–System sends: message screen (see Fig. 1(b)) with message 'OK'

If we consider each screen as a unit of interaction, we may again consider each unit to be an RPC. But in contrast to the case of the command language interface, each RPC is initiated by the application server and the user plays the role of the RPC server side. This, in fact, corresponds to what is really going on. The application is in charge of the sequencing of the screens, but it takes the responses from the user into account for making certain decisions.

3.4. Defining the interface with servlets

The concept of a servlet was introduced in the context of Web application programming. A servlet represents a fraction of an application that corresponds to the code that should be executed in response to a single message (HTTP Request) from the user. Cutting the application into such servlets simplifies the task of the Web server that has to manage concurrent HTTP Requests from various users. If the request corresponds to a servlet, it may create a new instance of the servlet, execute it in parallel with other requests, and destroy it afterwards. A servlet has local state, but it may also refer to state variables of the application, called session parameters. Sometimes the concept of a servlet is combined with facilities for assembling Web pages in HTML format (e.g. Java Server Pages, or Microsoft's Active Server Pages).

If we assume that our servlet implementation of the GUI presents the same screen-oriented interface as discussed above, we would have the same exchange of message between the user's browser and the Web server. There would be two servlets, one that searches the database to find a candidate booking and one that confirms the booking. They would be executed in the Web server after the reception of the messages Screen1(b) and Screen2(b), respectively. The first servlet will generate the Screen2(a) message and the second the Screen3(a) message.

From the point of view of these servlets, each servlet sees first a message from the user to which it then responds. If we take a servlet as the unit of interaction, each unit can therefore be considered to be an RPC initiated by the user, as in the case of the command language interface.

3.5. Discussion

As we see in our example, a single service, for instance the 'Find Booking Candidate' method of the Hotel Booking application, needs normally at least two screens, one for collecting the input parameters that are passed to the service method, and one for displaying the results to the user and to allow the user to select the next step of the interactions. Through the different responses that the application may provide to a given user request and through the different options for the next step provided to the user, the application allows for a large number of different interaction scenarios.

It is not immediately clear whether the screen-oriented or the servlet-oriented approach to defining the interactions at the GUI is more natural for the designer. The servlet-oriented approach is closer to the logical functions that are provided by the application to the user, while the screen-oriented approach is closer to the way the user will see the system. Further study should determine their relative advantages and shortcomings.

While RPC and screens are relatively small units of interactions within a longer use case scenario, there are various notations that have been proposed to describe such scenarios on a larger scale at the logical level (independent of screen layouts). Such notations are Use Case Maps [5], UML's Activity Diagrams [14], and Live Sequence Charts (LSC) [7,8] (to mention just a few). We mention that a tool is described in [8] that allows capturing requirements in relation with screen-oriented GUIs and automatically store them in the form of Live Sequence Charts. This approach appears to be quite similar to the Lyee methodology [13]. In the context of workflow modeling using Web Services, the so-called Business Process Execution Language (BPEL), an extension of WSDL, is proposed for describing the allowed execution sequences for activities or service calls.

4. Responsibilities of actors and components

4.1. Responsibilities for control flow decisions

It is quite common that a single scenario definition, such as the one given in Fig. 2, covers different alternate execution sequences. In the example of Fig. 2, the activity *Prepare booking* has three normal outcomes (*No room*, *Invalid request*, and *Confirm booking*) plus the *quit* alternative possibly initiated by the user, while the activity *Confirm booking* has two possible outcomes. Most notations for describing such scenarios do not explicitly indicate the responsibility for such control flow decisions. However, for the understanding of the scenario, this information is quite important. In the case of our example, the decision about the three normal outcomes of the *Prepare booking* activity is made by the booking application, while the decision about the outcome of the *Confirm booking* activity is made by the user. If it were the opposite, the behavior of the booking application would be quite different.

We note that with the notation of the activity diagram of Fig. 2, and in other similar notations intended for describing the black-box behavior of a system, the detailed realization of an activity is not defined. If an activity represents interactions with the user, or some other component outside the system boundary, it is not indicated which initiatives are taken by the user (or the environment of the system) and which initiatives are taken by the system itself in order to realize the execution of the given activity. At this high level of abstraction, an activity may be represented as a rendezvous between the system and its environment, as defined in the specification language LOTOS [9].

However, in order to define the system behavior in terms of requirements that must be satisfied by the implementation of the system, it is important to distinguish between the properties that must be satisfied by the system implementation and the assumptions that can be made about the behavior of the system's environment [15]. This means, we have to specify which party (the system or the user) makes the control flow decisions and which party provides input parameters for a particular procedure call.

We, therefore, suggest that a scenario definition of the behavior of a system should include for each activity that admits several possible outcomes, an allocation of responsibility for the control flow choice to either the system to be built, or to the user. And if certain parameters are determined during the execution of an alternative, then similarly, the responsibility for determining the value of each parameter should be stated.

In the case of the example of Fig. 2, the activities *No room* and *Invalid request* have only one outcome and no parameters, the activity *Confirm booking* has two alternate outcomes determined by the user, and the activity *Prepare booking* has three possible outcomes determined by the system, one initiated by the user, and several query parameters determined by the user. Therefore, we cannot allocate the responsibility for the latter activity to a single party. In fact, it may be a good design practice to separate this activity into two sub-activities, each with the responsibility of a single party: (1) the sub-activity *Prepare request* which determines the query parameters, a responsibility of the user, and (2) the sub-activity *Query database* which has three possible outcomes and is the responsibility of the Room Booking system.

4.2. Guidelines for designing action scenarios

We may generalize the above discussion and come up with the following steps for elaborating the description of the behavior for a given system, as an extension of the screen-oriented interface design approach described in Section 3.3:

(a) Define the different activities in which the user is involved; they correspond to the different user interface screens.
(b) For each activity, identify all possible activities that could directly follow.
(c) Determine responsibilities:
   (i) If for a given activity, there are several possible follow-up activities, find out whether the user or the system is responsible for making this decision.
   (ii) If for a given activity, there are parameter values that must be determined, find out whether the user or the system is responsible for selecting the values.
(d) If for a given activity, responsibilities from different parties were identified under the previous step, then the activity should be decomposed into sub-activities, such that the responsibility for each sub-activity lies only within a single party.

Applied to the example of Fig. 2, this approach results in the scenario definition shown in Fig. 3.

We note that the resulting sub-activities and the activities that do not have to be decomposed, each is characterized by

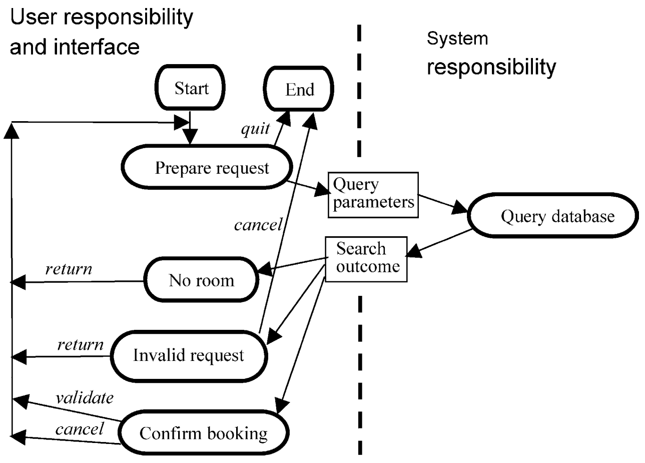

Fig. 3. Activity Diagram of Hotel Booking application showing two 'parties': (i) user responsibility and interface and (ii) system responsibility.

some actions to be performed by the responsible party and a certain number of (at least one) 'continuations', or follow-up activities. In most cases, a continuation is not only the identification of an activity (and its responsible party), but also some input parameters that are passed from the originating activity to the follow-up activity (e.g. the query parameters, and the search outcome shown in Fig. 3).

In general, we may assume that the input parameters for an activity provide all the information that is required from the environment of the responsible party for the execution of that activity. This is for instance the case for the *Query database* activity in our example. In the case of longer user interactions with the same application, it is often convenient to establish a 'session', which means that both parties involved keep the values of past activity parameters in local memory; it is therefore not necessary to transmit them again for initiating subsequent activities, since it is sufficient to transmit the session identifier and the other party can retrieve the previous parameter values from local memory, if they are required for the processing of a subsequent activity.

4.3. Implementation issues

We note that the above discussion is at a relatively high level of abstraction. In Fig. 3 only two 'parties' are identified: (1) *UserResponsibility and Interface*, and (2) *System Responsibility*. If we consider an implementation of the Room Booking application in a typical client–server architecture based on current Web technology, the *UserResponsibility and Interface* part of the system would be implemented in the user's workstation within an Internet browser, and by the user interacting through the GUI provided by this browser. The *System Responsibility* would be implemented in some server computer that would also contain the database of hotel rooms, or would be able to access such databases over the network.

We note that nothing has been assumed at this point about the location where the GUI information of the activities allocated to the *UserResponsibility and Interface* party and their continuations are stored. Different implementation strategies may be considered, such as the following:

(a) A single applet: the GUIs of these activities and their continuations (representing the control flow of the application) are included in a Java applet which is down-loaded when the application starts (see *Start* activity in Fig. 3). This applet may be stored in some HTTP server. The common security conventions for applets imply that the system component realizing the *System Responsibility* must reside in the same server computer.

(b) An application program: instead of being down-loaded as an applet, the system component providing the *UserResponsibility and Interface* may be realized as an application program running in the client's workstation. It may, in fact, have the same functionality as the applet considered under point (a), but it would normally have no restrictions as to the remote services it may access.

(c) HTML files plus servlets: the *Prepare request* activity could be realized as an HTML file stored in an HTTP server, which would have a continuation pointing to a servlet realizing the database query, and possibly running on a different computer. The response of the servlet will normally include an HTML string representing the GUI of the follow-up activity (either *No room, Invalid request* or *Confirm booking*). The *Confirm booking* HTML string would include a continuation pointing to another servlet that is responsible for committing the database transaction if the user enters the *validate* response. Here it could be useful to keep session state information from the execution of the first servlet to the execution of the second.

Clearly these different implementation choices can lead to very different software structures. We note that the second implementation strategy above corresponds to the paradigm of the 'command language interface' discussed in Section 3.2 in the sense that each of the two servlets corresponds to one remote procedure call from the client to the server.

Nevertheless, it appears that for the high-level conceptualization of a new application, the screen-oriented strategy discussed in Section 3.3 and represented by the diagrams in Figs. 2 and 3 are quite useful. Therefore, an interesting objective for further study seems to be the development of methods and tools that allow the systematic development of system implementations from abstract screen-oriented requirements definitions, allowing as input the various architectural and implementation options that may be desirable in practice.

Among these implementation options is also the choice of one of the RPC protocols, discussed in Section 2. The names and types of the input parameters for each of the different activities would typically be defined using the interface definition facility associated with the RPC protocol. In the case of CORBA, this would be an interface specification in the IDL language; for SOAP, this would be an XML Schema; and for Java RMI, this would be a Java interface definition. As mentioned in Section 2, the support tools associated with these RPC infrastructures may be useful in this context.

5. Conclusions

We conclude from this study that the Remote Procedure Call (RPC) paradigm is a very general communication primitive which is suitable for inter-component communication in distributed software systems, as well as for describing the interactions with a user at an abstract level (independent of the layout of the graphical user interface). There are two ways the RPC paradigm can be applied to the user interactions (1) in the traditional way of letting the user call the services of the application, and (2) in line with screen-oriented GUI interface design, by considering that the user answers questions and selects choices that are presented by the system to the user through a given screen layout.

We pointed out the importance of identifying the party that is responsible for making the choice of alternative control flows, whenever such a choice exists. This leads to guidelines for describing system behavior scenarios at the requirements level, in line with the screen-oriented requirement specifications proposed in the literature. This methodology can be used for describing the global system behavior as viewed from the user perspective, as well as for describing the behavior of a particular system component, as viewed through the interactions with its environment. It applies also to the description of workflow systems involving many system components. Further work is required for building tools that could support the implementation process that leads from these abstract system requirements to an implementation within a given software architecture.

References

[1] D. Amyot, A. Eberlein, An evaluation of scenario notations for telecommunication systems development, Ninth International Conference on Telecommunications Systems (ICTS'01), Dallas, USA, 2001.

[2] S. Baker, CORBA Distributed Objects: Using Orbix, Addison-Wesley, Reading, MA, 1997.

[3] G.v. Bochmann, Describing requirements in Lyee and in conventional methods: towards a comparison, in new trends in software methodologies, tools and techniques, Proceedings of Lyee_W02 Conference, Paris, IOS Press, 2001. pp. 239–253.

[4] D. Brown, M. Burnett, G. Rothenmel, End-user testing of Lyee programs: a preliminary report, in new trends in software methodologies, tools and techniques Proceedings of Lyee_W02 Conference, Paris, IOS Press, 2001. pp. 239–253.

[5] R.J.A. Buhr, Use case maps as architectural entities for complex systems, IEEE Transanctions on Software Engineering 24 (12) (1998) 1131–1155.

[6] Furmento, N., Lee, W., Mayer, A., Newhouse, S., Darlington, J. ICENI: an open grid service architecture implemented with Jini, in Proceedings of SuperComputing, Baltimore, USA, Nov. 2002.

[7] D. Harel, Can behavioral requirements be executed? (and why would we want to do so?), Proceedings of EMSOFT 2002, Lecture Notes in Computer Science, vol. 2491, Springer, Berlin, 2002. pp. 30–31.

[8] D. Harel, R. Marelly, Specifying and executing behavioral requirements: the play in/play-out approach, Software and System Modeling (SoSyM) 2 (2) (2003) 82–107.

[9] T. Bolognesi, E. Brinksma, Introduction to the ISO specification language lotos, Computer Networks and ISDN Systems 14 (1) (1987) 25–59.

[10] J. Landay, B. Myers, Sketching interfaces: toward more human interface design, Computer 34 (2001) 56–64.

[11] Negoro, F. Intent operationalisation for source code generation, in Proceedings of SCI, Orlando, FL, USA, July 2001.

[12] Open Grid Service Intrastructure, see http://www.gridforum.org/ogsi-wg/.

[13] C. Salinesi, M. Ben Ayed, S. Nurcan, Development using Lyee: a case study with LyeeAll, Internal Technical Report TR1-2, University of Paris I and ICBSMT, Oct. 2001.

[14] J. Rumbaugh, G. Booch, I. Jacobson, The Unified Modeling Language, User Guide, Object Technologies/Addison-Wesley, 1999.

[15] M. Abadi, L. Lamport, Conjoining specifications, ACM Transactions on Programming Languages and Systems 17 (3) (1995) 507–534.

[16] G. Blair, J.B. Stefani, Open Distributed Processing and Multimedia, Addison-Wesley, 1998.