Design of universal personal computing using SDL

M. Törö, J. Zhu, V. Leung*

Department of Electrical and Computer Engineering, The University of British Columbia, 2356 Main Mall, Vancouver, BC V6T 1Z4, Canada

Abstract

Universal Personal Computing (UPC) is a novel concept to support nomadic computing on the Internet. UPC enables mobile users to access their personalized computing environment anywhere on the Internet using any available terminals. The system is modeled as distributed objects using Common Object Request Broker Architecture (CORBA) to facilitate their distribution, interfacing, and integration. We have chosen the Specification and Description Language (SDL) for our system design. Through several iterations, we refine the specification and verify the correctness of the design. The final version serves as the basis for system implementation. The SDL specification allows a direct derivation of object interface description in Interface Definition Language (IDL). Finally some implementation issues are discussed. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Common Object Request Broker Architecture; Formal specification; Formal verification; User mobility; Specification and Description Language

1. Introduction

Nomadic computing on the Internet has emerged as a new research area due to the widespread use of portable computing devices such as personal digital assistants and laptops, and the demands of users of these devices for Internet access. Nomadic computing is characterized by independence of location, motion, and computing platform, with ubiquitous access to local and remote files, systems and services [5]. In other words, the essence of such a computing paradigm is the ability to transparently access one's personal computing environment wherever one happens to be. Nowadays this is achieved by physically carrying the user's computing device such as a laptop furnished with the appropriate Internet connectivity and by the support of terminal mobility using protocols such as mobile IP (MIP) [12].

In a more elegant solution for nomadic computing, the user's personal computing environment is the one that is mobile. A personal computing environment consists of resources, services, and personal preferences as configured by the user. When the user moves, this set of facilities is delivered automatically to the user. We envisage that in the near future there would be many public terminals deployed for customer use (just like public phones today) that are connected to the Internet. One can use any idle terminal to access any of the computing resources and yet the familiar

personalized computing environment would also be locally available at the same time. One need not worry about what hardware or software platforms are used, just like one does not need to wonder about the brand or vendor of a public phone. We call this form of mobility user mobility, as opposed to terminal mobility where the physical device migrates [4,13]. A system that supports user mobility is called a Universal Personal Computing (UPC) system. The public or leased terminal is called a foreign terminal and the subnetwork to which it is attached is the visited network. Under such a system, users employ their globally unique logical identifiers to access their computing resources anywhere on the Internet, using any available terminal, mobile or stationary, that are attached to networks supporting UPC.

The concept of UPC is adapted from a similar concept of Intelligent Networks supporting Universal Personal Telecommunication (UPT) [3], in which users can access subscribed services on the basis of unique network transparent personal UPT numbers regardless of their current geographic locations. We applied this idea to the area of Internet computing [6,7] where users can access their personal computing environments adjusted to the local facilities at any locations. When a user travels to a new location, user's personal computing environment will be brought to the location, and the user can access both resources and services as if at home.

To provide this continuous personal computing environment on any available networked machine, a virtually homogeneous distributed processing environment is required

* Corresponding author. Tel.: +1-604-822-6932; fax: +1-604-822-5949.
E-mail address: vleung@ece.ubc.ca (V. Leung).

Fig. 1. An example UPC service.

which is independent of the computing platforms. Fortunately, the Common Object Request Broker Architecture (CORBA) of the Object Management Group (OMG) provides such a framework [2,9,14,17]. We favor CORBA over Microsoft's Distributed Component Object Model (DCOM) because CORBA is an open standard supported by the majority of the industry and is available on most platforms.

To design the UPC system we have selected the Specification and Description Language (SDL) of ITU-T [1,10]. This allows us:

- to design the system in several iterations with increasing level of details, supported by special purpose tools such as SDL simulator and verifier;
- to fully understand the system behavior by simulation at each stage of the design;
- to check the design against global and specific system requirements from the earliest stage;
- to find and correct errors before implementing the system;
- to make available documentation defining the implementation.

The latest version of the primarily asynchronous extended finite automata based SDL is object oriented and extended with synchronous communication. We used ObjectGEODE [8], a commercial SDL tool by Verilog for our work.

The rest of the paper is organized as follows. Section 2 gives a more detailed description of the UPC system and its object-oriented model. Section 3 briefly reviews SDL and a possible way of modeling CORBA objects, and Section 4 presents the SDL model of UPC with some highlights of the verification of this model. The definition of the major UPC objects in Interface Definition Language (IDL) based on the SDL specification follows in Section 5. Section 6 presents the prototype implementation. We conclude by discussing some directions for further work in this area.

2. Universal personal computing

Similar to UPT, UPC allows global access to the Internet regardless of the user's point of attachment to the network, and with the capability to use the user's Personal Computing Environment (PCE) to the extent of the terminal's capabilities. With the development of network-centric and server-centric computing, the point of attachment has become less important as the computing paradigm itself supports user mobility. However, it is still desirable for most users to be able to set up their own PCEs to include services that are available at home but not server based, e.g. a service that is provided by an application, and access them on any foreign terminals. We demonstrate this by some examples.

An imaginary example is when a businessperson travels and happens to be at a hotel or at an airport. The executive might need to use an in-house-designed inventory tracking system. In this scenario as part of the PCE the user has defined a need to use the inventory application service and its associated configurations or preferences. The businessperson could use a public terminal to bring up this PCE, download the objects that implement the client part of the inventory tracking system, and run the application. Once a user chooses to use this service, the UPC system automatically transfers the application's client part and its configuration to the foreign terminal (provided the terminal is able to support the required GUI) and fires up the application. The application runs as a client that provides a GUI interface and interacts with the server object to accomplish the functionality. Fig. 1 illustrates this example. The required calls are generated and processed by two agents: the user and the terminal agents.

Another scenario is using an e-mail application. The e-mail application itself may be available at a foreign terminal, but the user would prefer to have local access to the e-mail-related part of the PCE, such as address book, aliases, folders, preferences and other options. UPC can provide this PCE based setup by downloading them at the

Fig. 2. Object modeling of the UPC system.

start of the application, converting them to the proper format, and synchronizing any changes afterward.

It should become clear at this point that a distributed object architecture (such as CORBA) and a portable object implementation (e.g. using Java) are essential for UPC to work because objects are distributed and need to be easily movable on the network. A foreign terminal could also be running any operating system; hence the platform independence is essential.

To support UPC, naturally there is a need to identify the user uniquely. The identifier can be anything that is globally unique (such as Social Insurance Number), but it makes sense to use unique *names* that facilitate identification. The Logical Universal Identifier (LUI) was proposed in [6,7], which consists of the user's home domain name and her unique name in that domain. For example, an e-mail address satisfies this requirement. This allows us to use of directory services such as the CORBA Naming Service, an extended Domain Name Server, or X.500 in order to retrieve the user's PCE.

There can be different circumstances under which UPC should handle user mobility. The terminal capabilities may vary over a wide range; there may be terminal mobility. One may prefer to carry a lightweight device with possibly limited capabilities, in which case certain UPC functions fall onto the network side. However we currently focus only on the case where the UPC capable terminal device is CORBA and Java enabled. Terminal mobility is also considered.

There are two domains involved in UPC: the user's home domain; and the visited domain for those users away from home. At the visited domains, the terminals and the networks are logically separate, as mobile terminals could be associated with different networks at different time. We define for the UPC system the following major objects (Fig. 2):

- the User Agent and the User Profile (or PCE) located at the home domain;
- the Initial Agent, the User-Terminal Agent (UTA), and

the Terminal Profile located in the terminal at a visited domain;

- facilities and resources distributed among the terminal, the visited and the home networks.

The Initial Agent provides an initial interface for the mobile user to login at a terminal. The User Agent resides in the user's home network and keeps track of the User Profile. When a user logs into a terminal with her LUI and password, the Initial Agent locates via CORBA Naming Service the User Agent according to the LUI. Subsequently it authenticates the user by submitting the password to the User Agent. Next the Initial Agent submits the Terminal Profile to the User Agent to obtain relevant part of the User Profile (i.e. description of services registered in PCE in advance). To provide this, the User Agent compares the User Profile with the Terminal Profile and determines the services that this terminal is capable of supporting, in terms of, say, terminal hardware constraints such as colors and resolutions. Based on the returned terminal specific User Profile the Initial Agent starts the UTA with the correct parameters such as the requested screen layout, with or without the support of mobility, etc. From this point the UTA provides the user's familiar interface.

The UTA starts services requested by the user. In order to do this it checks the User Profile first to determine where to obtain a specified service. If the service is obtainable locally, the UTA checks with the local service provider to see if service access is permitted, and gives the user a choice of accessing the service locally. Services specified in the User Profile by default can be accessed (remotely) at the home domain. We distinguish between the mobile and the stationary case in this respect.

For the stationary case all service comparisons are done once, instantly after login, for the whole profile. The user interface will display services supported by the foreign terminal in two sets: (1) those available at home; (2) those available at the foreign network locally. The user can then choose the service she wants, and the system will bring in the related objects (e.g. application client, preferences, etc.)

as required and run the application. The user interface also allows the user to query the local service provider for any other locally available services or facilities, based on the service types and attributes specified in the inquiry.

The situation is more complex in the case of a mobile terminal. The set of home provided services could be determined after login immediately. However, either the terminal or the visited network might provide the local services. Those provided by the terminal can also be selected after the login, while those provided by the visited network may vary due to terminal migration, and may become obsolete as a result. Therefore for these services the UTA acts only on demand to determine accessibility when the user actually requests a given service. For this, first of all, the UTA should be aware of the migration, which is hidden by MIP [12] from the layers above. Therefore we introduce a signaling from MIP to the UTA.

We refine some parts of the system based on our choice of the CORBA platform. Applications are split into the client part and the server part. The client part provides the user interface and has an interface toward the server part appropriate to the IDL specification. The server part can be a single object or a chain of objects, which can reside at any location and are accessed through the Object Request Broker (ORB). Since the interface between the server and the client part is specified in a standard way, it is possible to mix and match server and client parts. From the user's point of view, it is enough to keep the client part to provide the appearance of the same environment. Therefore the UPC User Profile refers only to the client parts of the applications, and its IDL type is the search criterion for the server part. At any location the local CORBA Trading Service can be used to match a client with a local server object. As Fig. 2 shows we distinguish the ORBs at the home domain and at the visited location. They may not know about each other. The User Profile maintains the interoperable object reference to the server objects at the home domain. To locate a visited ORB we assume a well-known mechanism (e.g. a well-known port on the serving router to contact the Naming Service).

There is a broad range of issues in UPC, such as location management of the mobile user, user PCE management, and security concerns (see Ref. [7] for the details). We focus in this paper on the distributed object architecture appropriate for CORBA and system design using SDL, which we discuss next.

3. Modeling CORBA with SDL

3.1. Brief summary of SDL

SDL is a language based on extended finite-state machines with asynchronous communication. SDL is standardized by the ITU (International Telecommunication Union) and was originally devised for specification and

description of communication protocols. Hence it is designed for interactive, real-time and distributed systems.

Extended finite-state automata, called processes in SDL, are the basic components of an SDL system. The behavior of the system is the composition of the behavior of these automata in the system. Processes are grouped into blocks, which in turn constitute the system. The means of communication between the components are signals delivered through signal routes between processes and channels between blocks. The behavior of the system environment is also assumed to be SDL-like. All blocks and channels of a system are created at the system start and remain until the system termination. On the contrary, processes can create other processes, and they can terminate any time; this feature provides some dynamism in the system structure.

The communication in SDL is essentially asynchronous. Processes and the system environment create and send signal instances to each other. Each process buffers received signals in its single input queue until consumption. Signal routes deliver signals instantly, while channels are FIFO and may introduce a non-deterministic delay. The 1992 version of SDL introduced remote procedure calls as means of synchronous communication. Another important novelty of this version of SDL is the application of object-oriented methodology for the structural elements of the specification.

3.2. Modeling CORBA with SDL

CORBA and particularly the CORBA IDL specifications define formally the interface between the client and the server parts of CORBA-based applications and services. They do not deal with the behavioral aspect of any object in the system. The standardized interface specification provides the mixing and matching of server and client parts. However, the IDL specification only cannot guarantee that a server object will perform the required actions to serve a request. SDL, on the contrary, is used to specify the behaviors of telecommunication systems, though still at an abstract level to enable platform independence. An SDL specification provides a stronger requirement on the server's behavior. The standardization of SDL resulted in a methodology and availability of commercial tools for system design and verification, which influenced our decision to use SDL to design our system. For this first we have to map CORBA services used by UPC to SDL.

CORBA objects can be modeled as SDL processes. The instantiation of an object would be the creation of a process; consequently we can use the unique SDL process identifier generated at creation as the object reference. This way the SDL interpreter will provide the ORB function of the signal distribution (for other ORB functions, e.g. object creation, one needs to specify special processes). With the object-oriented structures of SDL a process specification may be based on process type specifications. Process types are used in a similar way as C++ classes or IDL interfaces: inheritance can be established between them.

Fig. 3. The simplified SDL structure diagram of UPC.

There are three types of CORBA method invocations. The synchronous method invocation of CORBA can be modeled by a remote procedure call in SDL. In contrary to the simple procedure call of SDL, a remote procedure call may change the state of the exporter process. Also the exporter may block out certain calls in certain states. This is possible because the SDL interpreter maps a remote procedure call into an implicit signal transmission. Whenever the implicit signal to call the procedure is input in the exporter of the procedure, the procedure is interpreted. In states where the procedure is not explicitly mentioned, the exporter process returns to the same state after interpreting the procedure. However, the procedure may be explicitly specified, and after the procedure is interpreted, a transition is entered that can perform some actions and can lead to a new state. Alternatively, the procedure can be explicitly saved, meaning that the consumption of implicit signal to invoke the procedure is blocked. At interpretation the interpreter will schedule the call only at those states of the exporter process where the procedure is not saved, and will block the importer process until the call is completed.

The two other types of CORBA method invocations: deferred method invocation and one way invocation can be easily represented as asynchronous communications with signals, which are native in SDL.

Unfortunately, in ObjectGEODE, the SDL tool we used for our work, there are some restrictions with respect to the 1992 version of SDL. One of them is that it does not allow remote procedure calls.¹ Therefore for synchronous method invocations of CORBA we had to explicitly specify the signal exchange between the calling and the called processes. As we described above the SDL interpreter would provide the signal distribution implicitly (the recently

approved ITU Z.130 Recommendation [11] uses a similar approach since the SDL remote procedure does not support exceptions). However, this gives a consistent way of method invocation for all three types of CORBA invocations in our design. Note also, that the synchronous method of invocation in CORBA is possible only for the static invocation interface, i.e. through precompiled stubs. The dynamic invocation interface allows instead the deferred invocation, which is therefore a more general solution.

The synchronous invocation differs from the deferred invocation by blocking the caller. To provide this we introduce a blocking state for the caller whenever a synchronous invocation is assumed and start a timer to avoid a system deadlock. In Z.130 this is done via a local procedure call which, however, contains only a blocking state. No timer is declared. Consequently all CORBA method invocations are mapped into a single or a pair of signals. The first, and in the case of one way invocation only signal delivers the service request to the intended server process. It has a parameter list composed of the *in* and *inout* parameters of the IDL specification. The second signal is used to return the result and the control to the caller. Its parameters consist of the *out* and *inout* parameters of the invocation.

Finally, an unsuccessful CORBA invocation raises an exception; however SDL does not have such a construction. Therefore we specify the 'exception' signal for all CORBA processes, which may return an exception occurring at request processing. The signal carries the exception specific information as parameters. Timeouts resulted by timers guarding blocking states are interpreted as exceptions.

4. SDL specification of UPC

In the UPC model we define three groups of entities: entities at the user's home domain, entities at the visited

¹ The remote procedure call is supported in the latest version of ObjectGEODE.

domain, and entities in the user's terminal. We group these entities into three SDL blocks as shown in Fig. 3. Since we require CORBA in all parts of our system, a CORBA Naming Service process has been defined in all three blocks as a minimum configuration. In addition, we define Trader Services in the terminal and at the visited domain. We did not specify any of the ORBs (that is, the Home ORB and the Visited ORB of Fig. 2) as an explicit process, since the specification of the signal paths and the signal distribution provided by the SDL interpreter was sufficient for our purposes.

4.1. Specification of CORBA services

As part of system initialization, the Naming Service processes exchange their process identifiers and name bindings to become known in the system. Each Naming Service also registers the name and process identifier bindings in a given block. Since this binding is created by the invocation of the 'bind' method of the Naming Service, we define a 'bind' signal with two parameters: the name and the process identifier. The name is given in the SDL specification, while the process identifier is generated by the SDL interpreter at system startup. The bind method returns no values, so we define no outputs in the state transition except for an 'exception' signal which reports if the binding already exists. The invocation of the bind method is non-blocking.

To find a given object by name one has to invoke the 'resolve' method. Accordingly we define the 'resolve' signal carrying a context and a name. The context is used to identify the Naming Service. The name is the one that is registered with this Naming Service. On reception of a 'resolve' signal the context parameter is first checked and if it is not equal to the process name processing the request, the appropriate Naming Service is looked up and the request forwarded to its address. The invocation of the 'resolve' method is blocking; therefore both the caller and the forwarding Naming Service are blocked until the response arrives. A time-out raises an exception. The response for a 'resolve' signal must represent the returned value of the 'resolve' CORBA method. In our case this is the process identifier and we define the 'location' signal to carry it. We have to take into account the possible exception if no binding is found. Both the 'exception' and the 'location' signals can be generated by the transition triggered by the 'resolve' signal. Using this approach, we also specify the 'LUI' signal with the same structure as the 'resolve' signal. It carries the user's home domain name, which identifies the Naming Service and user name as it is registered during initialization with this Naming Service.

The Trader Service maintains a database of the properties of different available services. Upon request it matches them with the required properties and returns a list of matched objects, referenced by their process identifiers. The database of offered services is predetermined at process initialization.

The Trader Service is invoked at the reception of a 'lookup' signal carrying the requested properties. The data structure for the 'lookup' signal is defined according to the IDL specification of the query interface of the service. To enable requests to be evaluated, appropriate operators are specified for each property defined and used in the system. The result of a service comparison is returned in the 'object' signal, which carries the return value of the CORBA method: a list of services satisfying the request. We restrict the list to a single item in the design for technical reasons. Since the Trader Service can also raise an exception, we have added this possibility to the specification as well.

While the specification of the behavior of the Naming Service is straightforward, at the specification of the Trader Service it was obvious for us that the information provided in the current CORBA standard is not enough for proper functioning. Different Traders may maintain different sets of properties for the same type of servers, or just refer to them by different names. Even a comparison can be interpreted many different ways. To agree on all these is inevitable not only for UPC.

With the abstract data type of SDL we could specify all the necessary information. For UPC the Trader Service query interface serves as a model also for the User Profile. It was extended with other UPC specific information such as from where a service can be obtained. This way services registered for the user in the User Profile can be inquired through the Trader Service. For the same reason the Terminal Profile describing the terminal capabilities uses a similar data structure.

4.2. UPC objects

To provide synchronous calls of CORBA in our UPC specification, any time a synchronous invocation occurs, a blocking state is introduced in the caller, from which only the inputs received in response to the invocation (i.e. the returned result and the exception) are processed. Any other input is saved for this state. The blocking state is also guarded with a timer from indefinite blocking of the system.

Beside the CORBA components we have the following processes in each block shown in Fig. 3.

- User's home: User Agent process.
- User Terminal: Initial Agent process, UTA process, Application Manager process.
- Visited domain: none.

The User Profile (or PCE) does not appear as a separate object, it is declared as a variable belonging to the User Agent process. Similarly the Terminal Profile is declared in the Initial Agent process. The abstract data types for both are defined at the system level.

The User Agent process is created at the system start and remains active until the system termination. Its behavior is straightforward. After initialization, which includes the request for the registration of the user's name and the

Table 1
Summary of the SDL specification

Process/procedure	Number of states	Number of inputs	Number of timers	Number of procedures
Naming Service	3	4	1	2
Trader Service	2	2	–	1
User Agent	3	6	–	3
Initial Agent	8	6	2	1
UTA	5	9	1	7
Application Manager	3	4	1	–
New Service of UTA	4	5	1	–
Update of UTA	2	1	–	–
Stop Local of UTA	3	2	–	–

User Agent's process identifier binding with the Naming Service, and the initialization of the User Profile data structure, the User Agent reaches the 'off-line' state. It stays there until a login is initiated from the User Terminal by a 'login' signal carrying a string parameter. The User Agent authenticates the user based on this submitted password. If the authentication is successful, the User Agent moves to the 'online' state, where it stays until a 'logout' signal is received. While in the 'online' state, upon request the User Agent returns the User Profile, updates the profile, or deals with exceptions as necessary.

The Initial Agent of the User Terminal accepts the user login from the environment. It is initiated by two signals, which are the user's 'LUI' and the 'password'. With the LUI, the Initial Agent finds the User Agent via the Naming Services, and then it verifies the provided password with the User Agent. If the authentication was successful, the Terminal Profile is sent to the User Agent in order to get the User Profile adjusted to the terminal capabilities. Upon the receipt of the User Profile, the Initial Agent creates the appropriate User-Terminal Agent for the user, and then terminates.² It will be recreated again by the UTA after the user logs out.

The type of UTA created depends on the characteristics of the terminal and information contained in the User Profile. The UTA also receives the User Profile as a parameter at creation. The UTA represents a given user-terminal binding, and serves the user requests in this context. The user may request the UTA to update the User Profile, to logout, or to start a given service. The UTA also handles any exception, which may occur during these invocations.

To start up a service, the UTA first checks the User Profile to determine where to look up the service. If it should be started at the user's home, the UTA simply creates a new Application Manager for the service and passes the parameters given in the User Profile required to start up the service. These parameters provide references to the client object and to the configuration and preferences. If the service can be provided locally, the UTA checks first

with the User Terminal's Trader whether the service is available from the terminal, and if not, it contacts the Trader of the Visited Domain. Information about a potential server object received from any of the Traders, and the related data from the User Profile are passed to a newly created Application Manager. Services provided by the visited domain are restarted any time the user and terminal moves to a new domain.

To model mobility the UTA contains a spontaneous transition, which represent the signalling of the change in the point of an attachment. Since UPC itself does not detect the migration, it only reacts upon its detection by MIP, this feature of SDL to specify non-determinism in the system is sufficient for the model (to specify MIP, which detects the migration itself, we modified the channel concept of SDL [15]). We assume that the Naming Service of a new provider can be found at the address returned by this signaling via some default or well-known way. Therefore the UTA requests the new Naming Service to resolve the Trader's identifier, and then it contacts the Trader to locate the requested services.

The Application Manager administers a given service for the user. At service termination it terminates and notifies the UTA about the termination. If the user requests for an update of the User Profile, the UTA collects the necessary information from the Application Managers and sends it to the User Agent. At user logout the UTA sends a termination request (one-way request) to Application Managers, notifies the User Agent about the logout, and after creation of a new Initial Agent instance the UTA also terminates. The Application Manager process is a super type for any service.

To give an impression of the size of the current specification we summarized the number of states, inputs and timers of processes and procedures of our UPC design in Table 1. The whole specification is about 40 pages of SDL diagram, which is about 36 KB in text format (excluding the Common Interchange Format information).

4.3. Verification of the design

We take a stepwise approach in specifying UPC; i.e. we introduce new features and changes gradually into the specification. After each modification we verify that the

² For cases when the network provides the Initial Agent, it does not terminate but becomes ready to serve the next user.

Table 2
Summary of verification results

	(a)	(b)	(c)	(d)
Method	Breadth	Depth	Breadth	Breadth
Number of states	262 142	262 142	262 142	524 286
Number of transitions	684 944	719 355	723 606	1 566 044
Maximum depth reached	37	1000 (truncated)	35	35
Maximum breadth reached	90 506	n/a	86 714	148 866
Duration	6 min 16 s	17 min 50 s	13 min 56 s	18 min 28 s
Number of exceptions	0	0	18 991	0
Number of deadlocks	1	0	0	0
Number of stop conditions	0	0	0	0
Transitions coverage rate	62.94	62.94	59.21	56.60
Transitions not covered	53	53	62	69
States coverage rate	94.00	94.00	88.68	88.68
States not covered	3	3	6	6

system is free of deadlocks and exceptional situations. A *deadlock* is a system state when no transition can be fired in the system; an *exceptional situation* refers to a dynamic error during execution. The checking is done using breadth-first and/or depth-first analysis of model. To compare different versions of the specification we generated Message Sequence Charts (MSC) for the specification and used them as requirements for consistency. ObjectGEODE allows both automatic generation of MSCs and validation against properties given in MSC.

The first version specified the main components of the system and the basic communications between them. It contained many informal parts. We were able to use the SDL simulator, however the verifier did not produce any useful result. The elimination of the informal parts allowed the verifier to be used, and it reported no deadlocks or exceptions in the system.

In specifying synchronous CORBA calls we accidentally introduced an error into the specification by missing the transition handling an exception signal, which would move the process from the blocking state. This resulted in a deadlock, which was found by breadth-first analysis shown in Table 2, column (a). The same error was not found in depth-first analysis (see Table 2, column (b)). Adding the missing transition removed the deadlock; however the specification still was not complete and resulted in a number of exceptional situations shown again by the breadth-first analysis (see Table 2, column (c)). The reason was that we did not update the current provider address, therefore a request was made to it even though it did not exist any more. After correction of the error we ran the verification once again, and no new problem was found.

However the request–response pairs still could not be matched by the system. We added sequence numbering to the messages and finally after verification we obtained the report shown in Table 2, column (d).

As seen in Table 2, column (d), approximately half a million of system states and one and a half million of

transitions were investigated. The tool has reported 88.68% state and 56.6% transition coverage. After analyzing the detailed state and transition coverage reports, we concluded that we reached the maximum possible coverages for our specification, since all the reachable states and transitions of the SDL processes were traversed at least once. The uncovered states could not occur in our system during the verification. These were states and transitions of CORBA services, which were not used by the given part of the system. Further details on verification can be found in [16].

ObjectGEODE was an excellent tool for the design and verification of this complex system. SDL gave us the possibility from the very beginning to formalize and, consequently, to interpret in the same way our ideas. By using the simulator from the early stage of the design—even when it contained informal parts—gave us a clear understanding of the system behavior. The verification of the design increased our confidence in its correctness, even though we did not use all the capabilities of the tool. The only problem we had with the tool was its handling of time-outs in the system. From our point of view its implementation is not consistent. The consequence of a time-out is evaluated only when alternative signals are external for the system signals. It is not considered when alternatives are signals generated by other system components. As a result certain transitions were not checked at automatic verification and had to be verified manually with the simulator. However since the simulation results automatically update the verification results, these transition were included in the final coverage reports.

5. Derivation of the IDL specification from SDL

As the last step of the design we need to derive the IDL interface specification for our UPC objects from their SDL specification. The SDL specification will guide the

implementation of these objects, and conformity to it ensures that the implementation will behave as desired. However since our system is designed for CORBA, it is required to provide the interface specification in IDL for accessibility.

The derivation of the IDL specification from our SDL specification is the reversed procedure of the one used to specify the CORBA services in SDL. Using this procedure we define an object for each SDL process specified.

In two cases SDL processes are further divided: the User Profile is separated from the User Agent, and the Terminal Profile divided from the Initial Agent. This allows independent handling of these objects in the future, although it also requires additional interface specifications. For the moment we simply chained the appropriate profile-related methods of the agents to the profiles.

The procedures embedded in the different processes have not been separated from the processes because they did not expose a different interface to the environment. The SDL procedure is only a shorthand to structure the behavior description of a process, and the IDL specification does not deal with the behavior part.

Next we define the interfaces. Interfaces of CORBA objects are defined for the server objects. The easiest way to find the interfaces is to check the communication paths between the components. For processes which communicate with the user we separate this communication, since it is the part that handles the user interface, i.e. the client part of the object. Both the Initial Agent and the UTA have an interface toward the user as server objects whose interfaces must be defined. This separation immediately results in the freedom provided by CORBA: these objects may be moved around on the CORBA bus. That means, they can be provided by the network side, and migrated by CORBA, as the user migrates. The Initial Agent is a client in all other communications. The User Agent is a client with respect to the Naming Service, but it is a server for the UTA. On the contrary, the UTA is a client with respect to the User Agent and the CORBA services. However it is a server for the Application Manager. Finally the Application Manager has both client and server roles with respect to the UTA.

After determining the interfaces we define the methods for each interface. Again by reversing the procedure employed for CORBA services, we define methods for each input signal where a process plays a server role. The return value of the method is the signal (or its first parameter) which is sent in response to the signal. Obviously the 'exception' signal results in the exception definition part of the IDL specification rather than in a return value. The signal parameters become the parameter list of the method. The input signal's parameters are mapped into the 'in' parameters of the method invocations, while the parameters of the output signal determine the 'out' parameters except the one which has been used as the return value.

In cases where the signal parameters are derived types or new SDL types, we map the data structure into IDL typedef definitions.

Obviously we did not need to derive the IDL specification for the CORBA based processes. The following gives the IDL definitions of the UPC processes.

```
// User Agent interface
interface UserAgent {
    // toward the Initial Agent
    boolean authenticate (
        in string address,
        in string password)
        raises NO_RESPONSE;
    PCE get_profile (
        in hardware_capabilities
        TerminalProfile)
        raises NO_RESPONSE;
    // toward the User-Terminal Agent
    oneway update(
        in PCE UserProfile);
    void logout(
        in string address);
};

// Initial Agent interface toward the user
interface InitialAgent {
    void LUI (
        in string LogicalUserIdentifier)
        raises NO_RESPONSE, BAD_PARAM;
    void password (
        in string UserPassword)
        raises NO_RESPONSE, BAD_PARAM;
};

// User-Terminal Agent interface
interface UserTerminalAgent {
    // toward the user
    void startservice (
        in string ServiceName)
        raises NO_RESPONSE, BAD_PARAM;
    oneway update_profile()
        raises BAD_PARAM;
    void logout();
    // toward the Application Manager
    oneway terminated (
        in Object Service)
        raises OBJECT_NOT_EXIST;
};

// Application Manager interface toward
// the User-Terminal Agent
interface ApplicationManager {
    ServiceProfile get_description ();
    Void terminate ();
};
```

Table 3 gives a summary on the mapping between SDL and IDL.

Table 3
Summary of mapping between SDL and IDL

SDL	IDL
SDL interpreter	ORB
Process	Object
Structure	Object
Process identifier	Object reference
Signal route	Interface
Signal output + blocking state with a timer from which the only transitions are the return value or an exception or the time-out	Method
Signal input	A method or a return value/exception if it is in a blocking state
Signal output	Method invocation if it leads to a blocking state, otherwise a return value
Signal <i>exception</i>	Exception
Input signal parameters	<i>in, inout</i> parameters
Output signal parameters	<i>out, inout</i> parameters

6. Implementation

We have done a prototype implementation based on the specification. Ideally an SDL specification is also used for code generation, and different SDL tools support C or C++ code generation. However, the UPC system requires Java implementation for portability, which unfortunately was not supported by ObjectGEODE or any other known SDL tool yet. Therefore the implementation from SDL specification has to be done manually.

We have chosen Visigenic's VisiBroker for Java 3.0 as our CORBA platform. JDK 1.1.4 was used for object implementations. Currently the prototype works on Sun Workstations with Solaris 2.5, but the implementation does not depend on a particular operating system such as Solaris as long as an ORB and a Java virtual machine are available. We are working on the extension of the prototype to other operating systems such as Windows 95 and Windows NT to verify the object portability as well as the ORB interoperability.

To illustrate the prototype implementation, Fig. 4 shows the interface for the Initial Agent before a mobile user logs in. The user needs to type in her LUI and password. This allows the Initial Agent to locate the User Agent for the user to authenticate her.

After a successful login, the UTA (Fig. 5) is started which allows the user to select any services that are available locally or remotely.

Local services are those that will be initiated locally but will be using their configurations and resources as set up at home if applicable. They will be found via the local Trader Service. If a selected application is part of the User Profile and requires a client object, it will be downloaded too.

Remote services currently are provided at home. They are already in the User Profile or located by the home Trader Service. In the former case, when the user wishes to use an application, the client object will be downloaded to the

Fig. 4. The user interface provided by the Initial Agent.

foreign terminal and then started. Certain services can appear in both lists like a printer service which may be located anywhere and the user can request this service with a location constraint, such as the printer should be located in a certain room.

7. Conclusions

We have presented our design of universal personal computing, which provides a continuous and transparent personal computing environment for nomadic users on the Internet. Our design takes advantage of the virtually homogenous computing environment provided by CORBA, and its many standardized services. Since many vendors support the development of CORBA, we expect the widespread availability of this architecture and its standard services in the near future. The CORBA approach allowed us to define the PCE as a set of client objects movable to the user location wherever the user happens to be. In the design we defined the main components accomplishing this task and their behaviors. For the design we used SDL, a formal specification language recommended by ITU. We modeled the user's migration by a spontaneous transition provided by SDL to specify non-deterministic events. In the case of UPC this approach was sufficient.

We have also presented how SDL can be used to specify a CORBA based system. We demonstrated how to map IDL specification into SDL and in reverse, how to derive the IDL specification from a design specified in SDL. Since the IDL specification focuses only on the interface definition, only the communication is used at this mapping. Beside the interface description, the SDL specification also provides the behavior of the objects involved. This serves as a basis for implementation of these objects. We suggest that beside the interface specification, it is important to formally specify the behavior of a CORBA service as well. Only by doing so can we ensure that its different implementations provide the same required service. From

Fig. 5. The interface provided by the UTA.

our experience SDL is an appropriate language to achieve this goal. A further advantage of SDL is that since it is a standard language, commercial tools are available which can assist the designer in better understanding the design and verifying its properties. We used ObjectGEODE to develop the UPC system specification and to verify that our design is free of deadlocks, exceptional situations, and non-successive loops. It also allows the specification of different user requirements in MSC which features we used to check the consistency between the different versions of our design.

We are further refining the specification and our UPC prototype as new features are being considered. An important area we have not taken into consideration yet in our design is security. Since UPC passes sensitive information through foreign, therefore potentially malicious networks, it requires a sophisticated security scheme. Some additional features being considered include allowing multiple user-terminal bindings, specification of the type hierarchy of the UTAs, and refinement of the application manager. In this regard, the prototyping helps to identify unforeseen issues or details, which require further investigations to ensure the completeness of the design.

Acknowledgements

This work was supported by a grant from Motorola Wireless Data Group, Richmond, BC, Canada.

References

- [1] Addendum 1 (10/96) to Recommendation Z.100—CCITT Specification and Description Language (SDL), ITU-T, 1996.
- [2] S. Baker, V. Cahill, P. Nixon, Bridging boundaries: CORBA in perspective, *IEEE Internet Computing* 1 (5) (1997) 140–144.
- [3] Draft Recommendation F.851 UPT-service Description, Version 7, CCITT SG1, 1992.
- [4] B.D. Johnson, Scaleable support for transparent mobile host internet-working, in: T. Imielinski, H.F. Korth (Eds.), *Mobile Computing*, Kluwer Academic, Dordrecht, 1996.
- [5] L. Kleinrock, Nomadic computing—an Opportunity, *ACM SIGCOMM, Computer Communication Review* 25 (1) (1995) 36–40.
- [6] Y. Li, V.C.M. Leung, Protocol architecture for universal personal computing, *IEEE Journal on Selected Areas in Communication* 15 (8) (1997) 1467–1476.
- [7] Y. Li, V.C.M. Leung, Supporting personal mobility for nomadic computing over Internet, *ACM Mobile Computing and Communications Review* 1 (1) (1997) 22–31.
- [8] ObjectGEODE documentation, Verilog Inc., 1997.
- [9] R. Orfali, D. Harkey, D. Edwards, *Instant CORBA*, Wiley, New York, 1997.
- [10] Recommendation Z.100 (03/93)—CCITT Specification and Description Language (SDL), ITU-T, 1993.
- [11] Recommendation Z.130 (02/99)—ITU object definition language, ITU-T, 1999.
- [12] RFC 2002—IP Mobility Support, Internet Engineering Task Force, October 1996.
- [13] F. Teraoka, Y. Yokote, M. Tokoro, A network architecture providing host migration transparency, *Proceedings of the Conference on Communications Architecture and Protocols*, ACM, New York, 1991, pp. 209–220.
- [14] The Common Object Request Broker: Architecture and Specification, V2.0–V2.3, Object Management Group, July 1996–December 1998, available at <http://www.omg.org>.
- [15] M. Törö, Simulation of IP mobility support: an experiment in mobile protocol specification with SDL, in: A. Cavalli, A. Sarma (Eds.), *SDL, MSC and Trends*, Elsevier, Amsterdam, 1997.
- [16] M. Törö, J. Zhu, V.C.M. Leung, SDL specification of universal personal computing: a case study of system specification and verification with ObjectGEODE, in: S. Budkowsky, A. Cavalli, E. Najm (Eds.), *Formal Description Techniques and Protocol Specification and Verification*, Kluwer Academic, Dordrecht, 1998, pp. 267–283.
- [17] S. Vinoski, CORBA: integrating diverse applications within distributed heterogeneous environments, *IEEE Communications Magazine* February (1997) 46–55.