Migratory Applications

Krishna A. Bharat
Graphics, Visualization & Usability Center
College of Computing, Georgia Tech.
Atlanta, GA 30332-0280
E-mail: kb@cc.gatech.edu

Luca Cardelli
Digital, Systems Research Center
130, Lytton Avenue
Palo Alto, CA 94301
E-mail: luca@src.dec.com

ABSTRACT

We introduce a new genre of user interface applications that can migrate from one machine to another, taking their user interface and application contexts with them, and continue from where they left off. Such applications are not tied to one user or one machine, and can roam freely over the network, rendering service to a community of users, gathering human input and interacting with people. We envisage that this will support many new agent-based collaboration metaphors. The ability to migrate executing programs has applicability to mobile computing as well. Users can have their applications travel with them, as they move from one computing environment to another. We present an elegant programming model for creating migratory applications and describe an implementation. The biggest strength of our implementation is that the details of migration are completely hidden from the application programmer; arbitrary user interface applications can be migrated by a single "migration" command. We address system issues such as robustness, persistence and memory usage, and also human factors relating to application design, the interaction metaphor and safety.

KEYWORDS: Application Migration, Collaborative Work, Interactive Agents, Application Checkpointing, Mobile Computing, Ubiquitous Computing, Safety.

1 INTRODUCTION

The goal of the human-computer interaction community is to make powerful applications easy to use, while retaining their full potential. For this purpose metaphors have been devised; metaphors like overlapping windows, direct manipulation, and hypermedia. A successful metaphor hides complexity, and allows users to accomplish their tasks with little effort. Often, a metaphor requires advances in technology before it can be effectively implemented. Conversely, a new technology often needs the introduction of new metaphors to harness it.

As the infrastructure for ubiquitous computing comes into being, new demands will be placed on the way applications cope with the needs of mobile and distributed users. New metaphors will be necessary to cope with these demands.

We introduce a new genre of user interface applications: *migratory applications* can migrate from one host to another, maintaining intact the state of their user interface. After migration, a former host may shut down without affecting the application. We discuss how application migration can be implemented at the programming language/environment level. Our approach places some demands on the programming environment, but almost none on the application programmer. No restrictions are placed, in principle, on the type of the application being migrated. The entire migration operation can be realized by the execution of a single command. The same technique use for transmitting an application can be used to save the running application to file and transmit it over other channels to be resumed at a later time.

Application migration is useful in the context of many agent-based collaboration metaphors. For example:

1. Applications that follow a user across physical locations: the *ubiquitous computing* metaphor. For "eager" behavior, some applications could use a location sensing device such as an "active badge", to automatically follow the user.

2. Applications that serve a group of people by travelling to each person's site in turn (e.g. a meeting scheduler): the *electronic secretary* metaphor.

3. Applications that interact with people on a user's behalf and carry out an agenda: the *interactive agent* metaphor.

4. Communication over email that is interactive and intelligent: the *interactive message* metaphor. Unlike previous implementations of "active mail" [6], the recipient is able to forward the interactive message after interacting with it.

In addition to self-induced migration as described so far, it is equally easy to allow a program to be migrated under external control. We could "drag-and-drop" programs from one machine to another in the same manner that we move files between folders, and windows between screens.

Section 2 describes our approach to programming migratory applications. In Section 3, we show how this paradigm was applied in the Visual Obliq environment [3] to support the migration of (arbitrary) user interface applications. In Section 4 we provide a walk-through of the process of creating a migratory application. In Section 5 we discuss some issues raised by migration. Section 6 lists related work. In Section 7 we draw some conclusions.

2 PROGRAMMING MODEL

Our programming model is based on the facilities available in the Obliq distributed scripting language [7].

2.1 Network Semantics

In Obliq, arbitrary data, including procedures, can be transmitted over the network. A piece of Obliq data can be seen as a graph where some nodes are *mutable* (meaning that they have local state that can be modified by assignment) and where other nodes are *immutable* (meaning that they cannot be modified). For example, the program text of a procedure is immutable and cannot be modified, while fields in an object are mutable because they can be assigned new values.

2.1.1 Network Transmission. When a data graph is passed to a remote procedure, or returned from a remote procedure, we say that it is transmitted over the network.

The meaning of transmitting a data graph is the following (see Figure 1). Starting from a given root, the graph is copied from the source site to the target site up to the point where mutable nodes or network references are found. Mutable nodes (indicated by shaded boxes) are not copied; in their place, network references to those nodes are generated. Solid pointers represent either local or remote references. Existing network references are transmitted unchanged, without following the reference. Sharing and circularities are preserved.

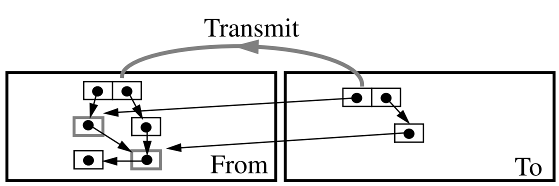

Figure 1: Transmission of a data graph

For example, an Obliq *object* (one of the basic data structures) is never copied over the network on transmission, since objects have state. A network pointer to the object is transmitted in its place. The object can then be referenced remotely through that network pointer; for example, one of its methods may be remotely invoked.

Arrays and updatable variables are similarly not copied on transmission, since they have state.

Obliq procedures are first-class data and, like other data, have a value that can be manipulated and transmitted. The value of a procedure is called a *closure*; it consists of the program text of the procedure, plus a table of values for the global variables of the procedure. Figure 2 shows the closure for a procedure incrementing a global variable x; the variable x denotes a mutable location containing 0.

The closure table contains a single entry, indicated by "where x = ...":

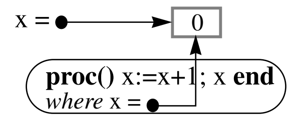

Figure 2: The closure of a procedure

The transmission of a closure (Figure 3) follows the same rules as the transmission of any data graph. When a closure is transmitted, all the program text is copied, since it consists of immutable data. The associated collection of values for free variables is copied according to the general rule. In particular, the locations of global updatable variables are not copied: network references are generated to their location, so that they can be remotely updated.

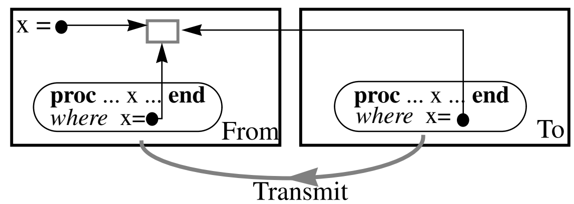

Figure 3: Transmission of a closure

2.1.2 Network Copy. In contrast to the default transmission mechanism, which stops at mutable nodes and network references, a special primitive is provided to perform a *network copy* of a data graph. This primitive makes a complete local copy of a possibly mutable and distributed graph.

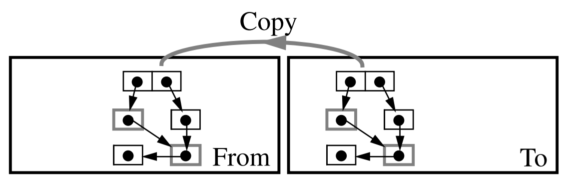

Figure 4: Network Copy

Network copy is useful, for example, when moving a user interface along with a migrating application.

A user interface is normally closely bound to site-dependent resources, such as windows and threads. Since these resources cannot migrate, a stand-alone snapshot of the user interface is first assembled. The snapshot consists of some complex data structure, including a representation of the current state of all the live windows of the application. This data structure, resembling the graph in the picture above, can be copied over to the target site, and then converted back to a live interface.

2.2 Agents

An agent is a computation that may *hop* from site to site over the network [19]. We review the concepts of agents, agent servers, suitcases, and briefings. In Section 2.3, we describe an Obliq implementation of agent hopping.

A *suitcase* is a piece of data that an agent carries with it as it moves from site to site. It contains the long-term memory of the agent. It may include a list of sites to visit, the tasks to perform at each site, and the results of performing those tasks.

A *briefing* is data that an agent receives at each site, as it enters the site. It may include advice for the agent (e.g. "too busy now, try this other site"), and any site-dependent data such as local file systems and databases.

An *agent server*, for a given site, is a program that accepts code over the network, executes the code, and provides it with a local briefing.

A *hop instruction* is used by agents to move from one site to the next. This instruction has as parameters an agent server, the code of an agent, and a suitcase. The agent and the suitcase are sent to the agent server for execution.

Finally, an *agent* is a user-defined piece of code parameterized by a suitcase and a briefing. All the data needs of the agents should be satisfied by what it finds in either the suitcase or the briefing parameters. At each site, the agent inspects the briefing and the suitcase to decide what to do. After performing some tasks, it typically executes a hop instruction to move to the next site.

If an agent has a user interface, it takes a snapshot of the interface, stores it in the suitcase during the hop, and rebuilds the interface from the snapshot at the destination.

2.3 Agent Migration

As we said in the previous section, an agent is a procedure parameterized with a suitcase and a briefing; the suitcase travels with the agent from site to site, while a fresh briefing is provided at each site. We assume that the agent code is self-contained (that is, it has no free variables).

Agents move from site to site by executing a hop instruction:

```
(* definition of the recursive procedure agent *)
  let rec agent =
    proc(suitcase, briefing)
      (* work at the current site *)
      (* decide where to go next *)
      hop(nextSite, agent, suitcase);
      (* run agent at nextSite with suitcase *)
    end;
```

In Obliq, agents, suitcases, briefings, and hop instructions are not primitive notions. They can be fully understood in terms of the network semantics of Section 2.1.

Agents are just procedures of two parameters. Suitcases and briefings are arbitrary pieces of data, such as objects. Each agent is responsible for the contents of its suitcase, and each agent server is responsible for the contents of the briefing. Agent servers are simple compute servers whose main task is to run agents and supply them with appropriate briefings (and maybe check the agent's credentials).

The hop instruction can be programmed in Obliq as follows:

```
    let hop =
      proc(agentServer, agent, suitcase)
        agentServer(
(1)       proc(briefing)
            fork(
(2)           proc()
(3)             agent(copy(suitcase), briefing);
              end);
            ok
          end);
      end;
```

Suppose a call `hop(agentServer, agent, suitcase)` is executed at a source site. Here, `agentServer` is (a network reference to) a remote compute server at a target site.

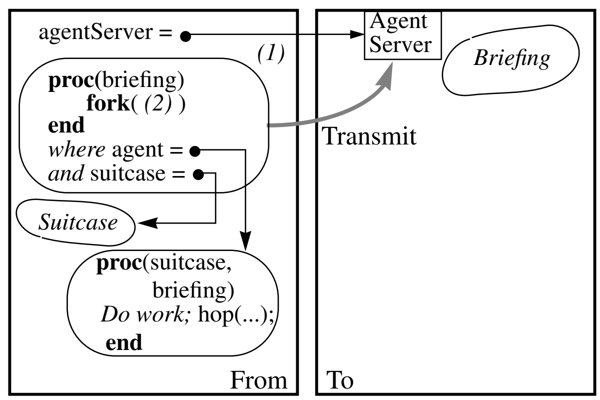

Figure 5: The hop instruction - Part I

The call `agentServer(...)` has the effect of shipping the procedure *(1)* to the remote agent server for execution. At the target site, the agent server executes the closure for procedure *(1)* by supplying it with a local briefing.

Next, at the target site, the execution of the body of *(1)* causes procedure *(2)* to be executed by a forked thread. Immediately after the fork instruction, procedure *(1)* returns a dummy value (`ok`), thereby completing the call to `hop` that originated at the source site.

The source site is now disengaged, while the agent computation carries on at the target site. The thread of computation at the target site is driven by the agent server. At the target site, the forked procedure *(2)* first executes **copy**`(suitcase)`. The suitcase, at this point of the computation, is usually a network pointer to the former suitcase that the agent had at the source site. The copy instruction (an Obliq primitive) makes a complete local copy of the suitcase, as described earlier. Therefore, the result of **copy**`(suitcase)` is a suitcase whose state is local to the target site, suitable for local use by the agent.

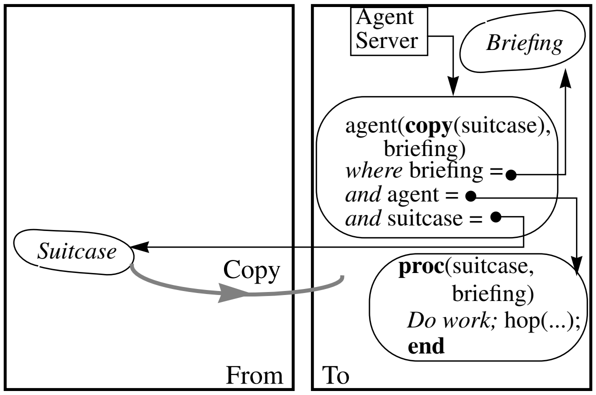

Figure 6: The hop instruction - Part II

After the copying of the suitcase, the agent migration is complete. The source site could now terminate or crash without affecting the migrated agent.

Finally *(3)*, the agent is invoked with the local suitcase and the local briefing as parameters. The program text of the agent was copied over as part of the closure of procedure *(1)*. Since the agent has no free variables, it can execute completely locally, based on the suitcase and the briefing.

In the special case when the suitcase contains the entire application state, we have a migratory application.

3 APPLICATION MIGRATION

We used the agent migration paradigm described in the previous section to implement migratory applications in Visual Obliq.

3.1 Visual Obliq

Visual Obliq is an environment for rapidly constructing user interface applications by direct manipulation [18]. It consists of:

- An interactive application builder that allows the user interface to be drawn and programmed. The builder generates code in Obliq.
- Runtime support, consisting of libraries and network services.

In previous work [3] we showed how the Visual Obliq environment supported the construction of distributed, multi-user applications (II, in Figure 7), in addition to traditional, non-distributed applications (I).

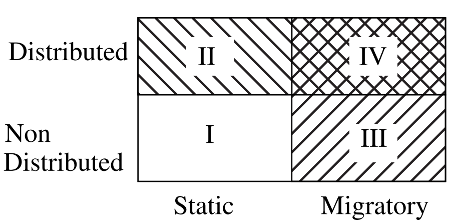

Figure 7: The space of networked applications

Here we describe how the environment was extended to support the creation of migratory, non-distributed applications (III). This was done in a manner transparent to the user, allowing any non-distributed application (in I) to be migrated by a single command. Migratory multi-user applications (IV) are significantly more complicated to implement, since connectivity needs to be maintained as the migration happens. We have yet to tackle this class of applications.

The support for distribution in II (described in [3]) has little in common with the support for migration. Hence we do not describe it here. However *Visobliq*, the GUI builder used to draw and program the interface has remained the same.

3.1.1 Visobliq.

Figure 8 shows Visobliq in action. The window on the left (in the background) is called the *design window*, and the window on the right (in the foreground) is known as the *attribute sheet*. The design window has a palette of widgets at the top, and a drawing area below, where widgets may be pieced together to form application windows. The application windows thus designed are called *forms*. The figure shows a single form being designed, containing the following widgets: a video-player, a button, a browser, and a file-browser. Widget geometry and the hierarchical nesting of widgets within a form can be manipulated interactively. All other resources are specified via the attribute sheet.

Figure 8: The Visobliq application builder

Double-clicking on a widget causes the resources of the widget to be loaded into the attribute sheet, for modification by the programmer. This includes attributes that determine the appearance and interactive behavior of the widget, as well as any code that is attached to the widget. When the resources have been modified, the programmer presses the ‘Apply’ button to make the changes take effect.

Pressing the ‘Run’ button causes the application to execute within an internal interpreter for testing and debugging. The ‘Code’ menu option provides a facility to output code in Obliq, for stand-alone execution within a Visual Obliq interpreter. We talk more about the interpreter and its special features to support migration in Section 3.3.

3.1.2 Programming a single-user application.

Each form defines a class of window objects, and can be multiply instantiated at run-time. Every instance of a form receives a unique index, and can be referenced through a global array that bears the form's name. For example, if the form being designed in Figure 8 were called MainWin, there would be an array called `MainWin[...]`, containing references to instances of MainWin created at run-time. Widgets are implemented as objects nested inside the form instance. Suppose the button labeled 'Capture' were named CaptureBtn, the programmer would refer to the button within instance `n` of MainWin as `MainWin[n].CaptureBtn`.

While building a single-user application in Visobliq, the programmer is asked to write four types of code in Obliq:

i. Callback code, which is attached to a widget
ii. Form support code, which is associated with a form.
iii. Global code. Any other code needed by the application can be placed here.
iv. Initialization code, which is executed when the program starts up and creates the initial form instances. After this the execution is fully input driven.

The above programming framework is general enough for the construction of most single-user UI applications.

3.2 Implementing Migration

The programmer makes the application migrate to a new site by executing the *migration command* within a callback. Specifically, one of the following commands is executed:

- `MigrateTo(Host)`
- `MigrateToServer(ServerName, Host)`

The first command migrates the application to a default agent server called '`VOMigrate`', on the machine named `Host`. `VOMigrate` continues the application from where it left off, and does not provide any briefing. This is sufficient for basic application migration. The second command causes the application to migrate to a customized agent server called `ServerName`, on the machine named `Host`. In both cases the agent server is run by the user who receives the application after it migrates.

3.2.1 The Migration Command.

The semantics of the migration command is that it returns `true` if the application is migrated successfully, and `false` upon failure. If it succeeds, the local instance of the application terminates the moment the callback finishes. The user interface is destroyed and the entire application state gets garbage-collected. In the event of failure, the application continues to execute locally as if nothing happened.

The **migration command** executes the following steps:

i. It first contacts the agent server at the destination to ensure that the migration can happen. Upon failure it returns immediately with a `false` value.

Otherwise...

ii. It checkpoints the state of the user-interface into the Obliq objects that make up the widget hierarchy.

This step is necessary because widgets in Visual Obliq are high-level "interface objects" in Obliq, which realize their presentation using lower-level interactors in the local UI toolkit. Currently, the only toolkit that is supported is Trestle [12], but if Obliq were ported to a different environment, the local toolkit would be used. Hence, Visual Obliq widgets do not maintain all of their state explicitly. In particular they do not maintain an up-to-date copy of attributes that can be changed interactively by the user (e.g. the geometry). These attributes are retrieved from the underlying toolkit whenever needed; either when the programmer's code requires them, or when the user interface state is being checkpointed.

iii. The user interface is destroyed, breaking links to the UIMS.
iv. Links to the local runtime are explicitly removed.
v. Visobliq prepares a suitcase, and executes the hop instruction discussed in Section 2.3. Recall that a suitcase is a data structure that gets copied to the destination. In this case, the suitcase contains a reference to each of the form-instance arrays in the program.

If the hop instruction executes successfully, `true` is returned. Upon failure (if the network operation raises an exception), the command rebuilds the user interface from the saved state, in the same way that the agent server at the destination would have, and returns `false`.

The hop instruction causes the agent server to perform a network copy of suitcase. Since the suitcase contains references to all form-instance arrays, this involves copying every piece of data that is reachable from a form-instance. It is easy to see that this will copy over every piece of the application state that is relevant to future execution. If a piece of data is not accessible from any form-instance, it will never be used, and so it is not copied.

At the source site, due to step iii, all links between the interpreter's UI threads and the application are destroyed. Once the existing callbacks exit, the application state becomes inaccessible to any thread in the system. The Obliq interpreter has automatic garbage-collection. Hence shortly after migration, the application state gets garbage collected.

Step iv ensures that the application state has no references to the Visual Obliq runtime when it is copied. This was done to prevent the runtime from being copied as well. At the new host, the local runtime is patched in, causing the local environment to take effect.

3.2.2 The Agent Server.

The agent server is an extended Visual Obliq interpreter. In addition to an internal UIMS thread, the agent server has a 'migration' thread to assist incoming agents.

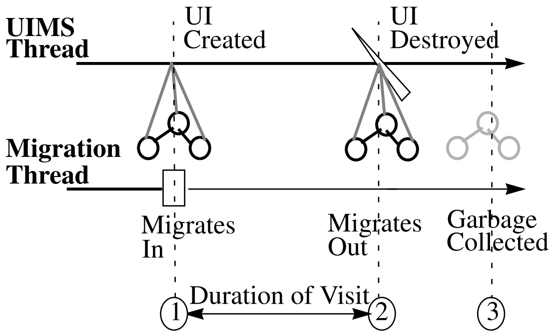

Figure 9: Operation of an Agent Server

When an application migrates in (at step 1 in Figure 9), the agent performs the following operations:

i. It performs a network copy of suitcase, causing the entire application state to be copied over.

ii. References to the local Visual Obliq runtime are added.

iii. For each form-instance in the application, it rebuilds its user interface based on its saved state. Callbacks are re-attached. This sets up links between the application and the local UIMS thread.

When an application migrates out (step 2), links from the UIMS are broken, and soon its state is garbage-collected (step 3). In this manner, the agent server allows applications to migrate in and out of the host repeatedly, without running out of memory. Multiple applications can co-exist within the interpreter, because they will not have links to each other.

User-defined agent servers are created by extending the default agent server to provide application-specific briefing and access control. To be useful, the agent server needs to have a user interface of its own to help the user monitor and regulate the activities of migratory applications. For example, the user might stipulate: "I will entertain only applications of type X"; "I will be back at time Y"; "If you get an agent from so-and-so, provide file Z as input". This presupposes an underlying mechanism for authentication and encryption. There is work in progress to provide secure communication and authentication at the Network Objects layer [5] – the transport layer for Visual Obliq.

In practice, there are likely to be other locale-specific resources, such as file-handles and network connections, that need to be preserved during migration. The replication of such resources cannot be automated since it is highly application and situation dependent. For instance, it is not clear how open files should be treated. One option would be to have the system reopen all open files upon reaching the destination, but often the two sites may not share a common file-system. Hence, we let the application programmer deal with the checkpointing and reinstantiation of such resources. The programmer is given the option of adding code to two system-defined routines: `PreMigrate()` and `PostMigrate()`, which are invoked before and after migration respectively.

3.3 The Visual Obliq runtime

The 'Visual Obliq interpreter' is simply the Obliq interpreter with a set of support libraries (known as the *runtime*) preloaded. The original purpose of the runtime was to provide access to the local UI library and implement abstractions needed by distributed applications.

Recently the runtime was redesigned and extended to meet the needs of migratory applications.

Firstly, since the runtime is closely tied to the local environment, it was decided that it would not be copied when the application migrates. Hence, all access to the runtime is through handles which are local to the interpreter in which the application is currently resident. The handles are removed before migration, and get patched in when the application arrives at a new host. Hence, all operations that involve local system resources such as the network, processor, file-system and the UI toolkit, are customized to the local environment.

In addition, the runtime provides the following facilities:

3.3.1 Migration Support. It implements the migration commands described earlier. The runtime at the source accesses the agent server using a remote-object access mechanism known as 'Network Objects' [5]. Then it checkpoints the local interface. At the target site, the agent server copies the application state over and uses the local runtime to rebuild the interface.

The two operations on the interface are implemented thus:

a) Checkpointing the user interface. This is done by walking the Visual Obliq widget hierarchy for each form-instance in the application, and copying relevant state information from the UI toolkit into the Obliq widget. Any attribute that cannot be modified by the user (and can only be modified under program control) need not be checkpointed, since the widget will already have the latest value.

b) Rebuilding the user interface. The same mechanism used to create the original user interface is used to rebuild it at new sites. The routine walks the Visual Obliq widget hierarchy for each form-instance and creates for each widget therein, a corresponding interface using interactors in the local toolkit. In doing so it may adhere to the checkpointed geometrical attributes or decide to override them, e.g. if the application migrates to a portable computer with a substantially smaller screen, dimensions might shrink. This provides the flexibility needed to cope with the differences between individual machines, while preserving the appearance of the interface as far as possible.

In our present implementation, we have another intervening layer, FormsVBT [2]. FormsVBT allows

Visual Obliq widgets to be described in terms of symbolic expressions representing the hierarchical arrangement of (smaller) UI components. The runtime generates the symbolic expression corresponding to each Visual Obliq widget by replacing tokens in a template with the attributes of the widget. Users can customize the appearance of the widgets displayed by their agent server by manipulating the template.

Once the user interface has been rebuilt, the runtime re-attaches callbacks so that interaction can resume.

3.3.2 Safety. The runtime is responsible for safety, and protects the user from attacks and privacy violations by the applications that migrate in. It does this by disabling all unsafe commands (namely commands that could be used to damage the user's environment and/or violate privacy), and instead provides safe alternatives that are subject to user-specified checks before execution.

In Obliq, all unsafe operations are readily identified by the fact that they require the use of "access" handles to system resources. For instance a `processor` handle is needed by routines that create new processes and execute system calls. Similarly there are handles to provide various levels of access to the file-system. The Visual Obliq runtime hides all system handles after having defined a "safe" version of each routine that uses a handle. The safe-routines have the handles bound inside them. An alien program can access a safe-routine but not the handles within it. These routines are considered safe because they compare their op-code and argument list with patterns in a user-specified configuration file (called `.vorestrict`), to decide which operations are to be allowed and which are to be blocked.

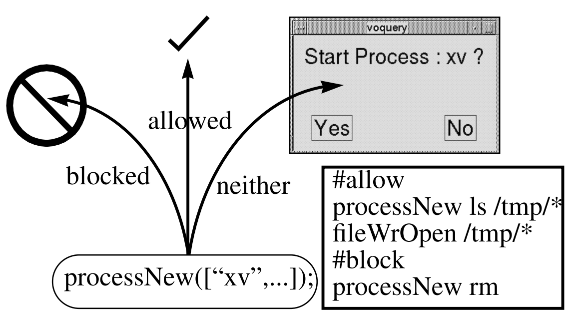

Figure 10: Safe Routines in Visual Obliq

Operations that are allowed by the configuration file are executed in the regular manner. When a blocked operation is encountered, the runtime notifies the user that the program is attempting something illegal and aborts the application. When an unsafe operation falls in neither category (which is the default case when no preferences have been specified) the runtime rewrites the operation in a human intelligible form, and pops up a notice to ask the user if it should be allowed to go through.

Unlike in Safe-Tcl [13], where a special "Safe" interpreter is required, we are able to implement safety entirely at the user level. Most users would use a default `.vorestrict` file provided by the system administrator. The ability to customize the file and relax the restrictions is useful within workgroups, where there is a high level of trust.

3.4 Variants

A variant of migration is *cloning*. An application is cloned by network copying it to the new site without destroying it at the original site. One possible application of cloning is in debugging. When a bug is encountered in an application, the user can send a clone of the application to the person responsible for debugging it, instead of a mere 'bug report'. Another application would be a divide-and-conquer agent mechanism, wherein an agent *splits* into multiple agents that interact independently with various users, and later *merge* or resynchronize.

Obliq provides a *pickle* operation, which is very similar to the network copy. Instead of replicating the data-graph, it writes it to a buffer. The contents of the buffer can be saved to a file or transmitted over another transport e.g. e-mail. At a later point in time, it can be converted back into the original data-graph, by the complementary *unpickle* operation. This allows Visual Obliq programs to checkpoint their state to file when necessary. If agents are expected to be persistent and endure machine crashes, they will need the ability to periodically save their state to stable storage, and resume from a saved configuration when the machine restarts.

4 A COMPLETE EXAMPLE

We present a small survey agent in its entirety, to demonstrate how easy it is to create a migratory application in Visual Obliq. The agent has an agenda consisting of a list of hosts to visit.

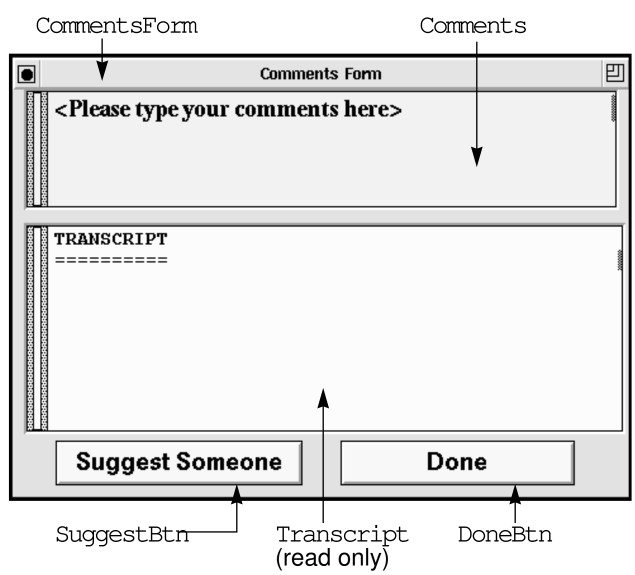

Figure 11: `CommentsForm`, a top-level form

At each host it presents the user with two top-level forms: `CommentsForm`, shown in Figure 11, and `SurveyForm`, shown in Figure 12[†]. SurveyForm has two questions to be answered by the user, and CommentsForm has an editor widget at the top, where comments may be typed in.

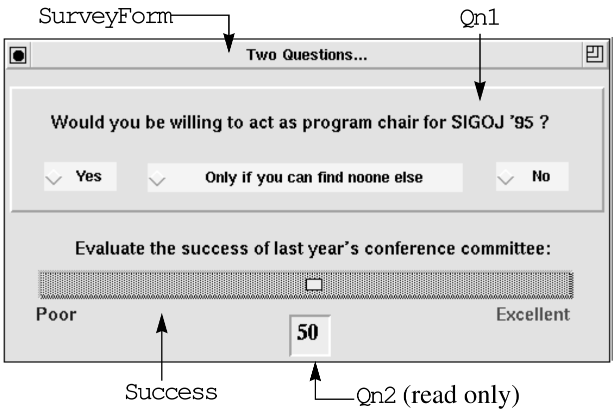

Figure 12: SurveyForm, another top-level form

When a user finishes with the questionnaire she clicks on the button labeled "Done" to send the agent on to the next user. The read-only "Transcript" window maintains a log of the input given by each user. When the agent has visited all hosts in its agenda, it will return to the host where it started.The initial list of users and hosts to visit is supplied by the person who starts the application. Subsequently, other users may add to the agenda. Users and hosts are added to the agenda via the Suggest form (Figure 13), which is popped up by clicking on the "Suggest Someone" button (in CommentsForm).

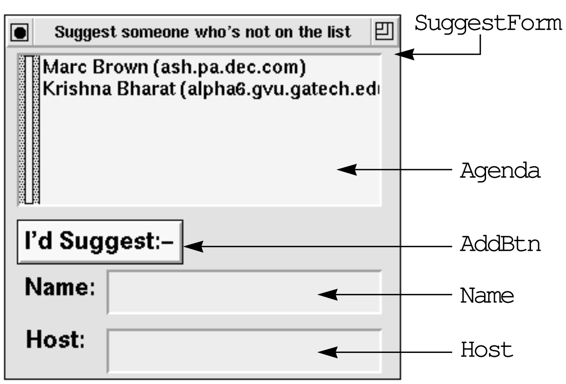

Figure 13: SuggestForm, a popup in CommentsForm

By default, the application is configured to create one instance of each top-level form when it starts up. In this case CommentForm[0] and SurveyForm[0] will be created. So no user specified *initialization code* is necessary.

The *global code* (shown below) contains a counter, NumVisited, to keep track of the number of hosts visited, and the arrays, people[...] and hosts[...], to keep track of the agenda. The host name of the originating site is saved in OrginalHost.

[†] The annotations represent widget names in Visobliq

```
var NumVisited = 0, people = [], hosts = [];
let OriginalHost = volibLocal.getHostName();
```

The following *callbacks* are added:

Clicking on SuggestBtn causes SuggestForm to pop up. At design time, SuggestForm is anchored to CommentsForm. This causes SuggestForm to become a field within CommentsForm. In Visual Obliq, SELF is used within a callback to refer to the current form. Hence the callback for SuggestBtn is as follows:

```
SELF.SuggestForm.show();
```

When AddBtn is clicked, the contents of the typein fields, Name and Host, are appended to the arrays, people and hosts respectively, and also to the browser named Agenda.

```
let name = SELF.Name.getText();
let host = SELF.Host.getText();
people := people @ [name];
hosts := hosts @ [host];
SELF.Agenda.append(name & " (" & host & ")");
```

The callback for the slider named Success copies the current slider value into the typein field named Qn2 :

```
let n = SELF.Success.getValue();
SELF.Qn2.putText(fmt_int(n));
```

When the button labeled "Done" is clicked, the user's comments and responses (to the questions in SurveyForm[0]) are appended to the editor Transcript. The editor, Comments, contains the user's comments. The answer to the first question is the name of the currently selected choice inside the frame named Qn1. The names of the three choices are Yes, Maybe and No (not shown). The answer to the second question is the text inside the typein field named Qn2. Then the destination host, dest, is computed and the migration command is invoked. If the migration succeeds the loop is exited; otherwise it tries to migrate to the next host. The code to do this is as follows:

```
let comments = SELF.Comments.getText();
SELF.Comments.putText(
        "<Please Type Your Comments Here>");
SELF.Transcript.appendText(people[NumVisited]
  & " said\n" & comments & "\n"
  & " Qn 1: " & SurveyForm[0].Qn1.getChoice()
  & " Qn 2: " & SurveyForm[0].Qn2.getText() );
loop
  if SELF.Agenda.numElements() is NumVisited
  then
    dest := OriginalHost;
  else
    dest := hosts[NumVisited];
    NumVisited := NumVisited + 1;
end;
  if MigrateTo(dest) then exit end
end;
```

The application programmer perceives the migration command as a primitive operation. In reality the operation `MigrateTo(dest)` involves:

- Contacting the `VOMigrate` agent server on `dest`.
- Checkpointing and destroying the user interface.
- Performing a network copy of `suitcase`, an array containing references to `CommentsForm[...]` and `SurveyForm[...]`. This causes `CommentsForm[0]`, `SurveyForm[0]` and all the global code to be copied over as well.
- Rebuilding the interface at `dest`; reattaching callbacks.

5 ISSUES RAISED BY MIGRATION

The ability to migrate applications can substantially change the way we view programs, and the way we interact with them.

- **Interaction Techniques.** The distinction between executing programs and data becomes blurry with the ability to migrate and checkpoint applications. Applications can now be treated as first-class objects on the desktop, and subject to direct manipulation in the same manner as files and folders. For instance, the iconic representation of an application could be dragged-and-dropped (or cut-and-pasted) from one region to another, causing it to move between machines and disks.
- **Privacy.** The ability to operate on an application from afar raises privacy and access-control issues. Does the person who started the application have "yank" and "kill" privileges? Most people would not like others to know what applications they have on their desktop; yet users would like to keep track of the agents they have sent out. It should be possible to restrict access to users within a group.
- **Secrecy.** Then there is the issue of privacy in the reverse direction. The user who receives an agent may violate the privacy of the sender by examining its hidden agenda. For instance a car dealership may wish to send an agent to a client, with a list of cars and a strategy to negotiate the price. It should not be possible for the client to learn what the strategy is, by examining the agent code or modifying the interpreter. It seems impossible to prevent this from happening in software. It could possibly be achieved by having a trusted third-party implement the interpreter in hardware, with encryption of the agent code.
- **Protocol.** In any agent-human interaction there is the issue of protocol. How long does an agent wait for a user to respond before deciding to move on? Does it leave a note? How does it know if the user is busy or away? It could look at the idle time. How does a user prevent unwanted agents from bothering him, while keeping the door open for acceptable agents? Do agents have a classification?
- **Heterogeneity.** If applications are to migrate between diverse architectures, the widget set needs to be chosen carefully so that it can be rendered efficiently in all environments. When rebuilding an interface, the checkpointed state should be interpreted in the light of available resources and local preferences. Fonts and labels need to change to match the local language.

6 RELATED WORK

Application migration most closely resembles the work on process migration in Operating Systems [14, 20], although the aim of process migration is to do load-balancing and improve parallelism. Process migration is usually implemented at a low level, making no assumptions about the application structure, or even about the programming language. The machine architecture and environment are assumed to be very similar, if not identical, at the two ends of the migration. Process migration is unable to cope directly with applications that keep part of their state externally, e.g. when much of the user interface state information is stored within a window system server (as in the X window system).

Our priorities are very different. Our focus is on heterogeneity (interoperabilty with diverse machine architectures), customization to the local environment (the user interface should be built using the local UI toolkit), and the flexibility that comes with implementing migration at the programming language level. In our system, the application programmer can implement new migration strategies by re-programming the migration mechanism. For instance migration could be limited to selected parts of an application by appropriately modifying the suitcase. Split and merge techniques could be used to deploy agents in parallel.

One of the strengths of our design is that migration is completely captured by the semantics of the programming language. This makes it easy to comprehend the program and troubleshoot it if it does not behave as expected. Also, heterogeneity is not a problem, since correct implementations of the language are guaranteed to interoperate.

Migratory applications may also be viewed as mobile, interactive agents. The term "agent" has been used with many different meanings. There are agents in AI, in databases, and in application software; classification and search agents (robots and knowbots) in information retrieval [10, 11, 15]; agents within adaptive applications and learning-by-demonstration systems [9]; and assistants in design automation and help systems [4]. Typically, these mobile agents are not interactive; for example, search agents operate silently on behalf of a client which interacts with the user. Conversely, interactive agents are usually not mobile across machines; they are usually "symbiotic" and exist within some application context or workspace; for example, helpers in learning-by-demonstration systems such as Eager [8], and "Balloon Help" on the Macintosh desktop [1].

Agents that support collaborative work on the other hand require to be mobile or at least distributed, and also need a user interface to interact with users. A major advantage of our design is that *any* single-user application in Visual Obliq can be turned into a mobile, interactive agent by invoking the migration command. When not in use, an agent can write its state to disk and be restarted when needed.

Obliq resembles Java [17], with its object oriented, multi-threaded features, but also has integrated support for distributed objects which Java lacks. HotJava [16] and Safe-Tcl [13] have taken steps to ensure safe execution of external code. Safe-Tcl supports virtually no end-user customization; HotJava allows users to restrict the execution of incoming programs, based on the host they came from (using access-lists). A difference between the execution of external programs in these systems and the migratory applications in Visual Obliq is that in the former case the applications always begin executing from a default (start) state when they arrive at an interpreter, instead of continuing from where they left off as in our case. Hence it would not be possible to forward an arbitrary program to a new user after interacting with it, as one could with a piece of annotated electronic mail.

7 CONCLUSIONS

The ability to migrate applications has several applications. In a world of ubiquitous computing, users may like their applications follow them – from home to work and to a colleague's machine. A migratory, interactive application can act as an agent, moving from one user's machine to another, interacting with each user, and carrying out an agenda. Such agents may be short-lived, if they are deployed by a user or a group for a specific task. They could be long-lived as well and perform a role in a work-group as a human colleague would. All of this presupposes a mechanism to migrate arbitrary applications between machines.

We have presented a distributed language semantics that supports application migration, and an architecture for migratory applications. The architecture has been incorporated into the Visual Obliq application programming environment [3]. We have yet to explore the full potential of this paradigm in collaborative work, but we have successfully migrated a number of small to medium size applications. In these cases, the migration operation took between 5 and 45 seconds over a local area network, depending on the program size and network traffic.

8 ACKNOWLEDGEMENTS

We thank Marc H. Brown for founding the Visual Obliq project and being a great source of help and encouragement.

9 REFERENCES

[1] Apple, "Human Interface Guidelines", Addison-Wesley, 1994.

[2] Avrahami, G., Brooks, K.P., and Brown, M.H., "A Two-View Approach to Constructing User Interfaces", *Computer Graphics*, 23(3):137-146, 1989.

[3] Bharat, K., and Brown, M.H., "Building Distributed Multi-User Applications By Direct Manipulation", *Proc. ACM Symposium on User Interfaces Software and Technology*, Marina Del Rey, 1994, pp. 71-82.

[4] Bharat, K. and Sukaviriya, P., "Animating User Interfaces with Animation Servers", *Proc. of UIST'93*. pp. 69-79.

[5] Birrell, A.D., G. Nelson, S. Owicki, and E. Wobber, "Network objects". *Proc. 14th Symposium on Operating Systems Principles*. 1993.

[6] Borenstein, N. and M.T. Rose, "MIME Extensions for Mail-Enabled Applications: application/Safe-Tcl and multipart/enabled-mail", *Draft*, Bellcore, Dover Beach Consulting, September, 1993.

[7] Cardelli, L., "A Language with Distributed Scope", *Computing Systems,* 8(1), 27-59. MIT Press. 1995.

[8] Cypher, A., "EAGER: Programming Repetitive Tasks by Example", *Proc. of CHI '91*, 1991, pp. 33-39.

[9] Cypher, A. [Ed], "Watch What I Do - Programming by Demonstration", MIT Press, 1993.

[10] Emtage, A. and Deutsch, P., "Archie: An Electronic Directory Service for the Internet", *Proc. USENIX Winter 1992 Conference*, 1992, pp. 93-110.

[11] Goldberg, D., Nichols, D., Oki, B. and Terry, D., "Using Collaborative Filtering to Weave an Information Tapestry", *Communications of the ACM*, **35**(12), pp. 61-70, 1992.

[12] Manasse, M.S. and G. Nelson, "Trestle reference manual". *Research Report #68*. Digital Equipment Corporation, Systems Research Center. 1991.

[13] Ousterhout, John K., "Scripts and Agents: The New Software High Ground", *Invited Talk at the Winter 1995 USENIX Conference*, New Orleans, LA, January 19, 1995.

[14] Powell, M. and Miller, B., "Process Migration in DEMOS/MP", *Proc. of 9th ACM Symposium on Operating System Principles*, 1983, pp. 110-119.

[15] Sheth, B., and Maes, P., "Evolving Agents for Personalized Information Filtering", *Proc. of IEEE Conference on AI for Applications*. 1993.

[16] Sun Microsystems, "HotJava Browser: A White Paper", *Sun Microsystems White Paper*, 1994.

[17] Sun Microsystems, "The Java Language: A White Paper", *Sun Microsystems White Paper*, 1994.

[18] Shneiderman, B., "Direct Manipulation: A Step Beyond Programming Languages", *Computer*, **16**(8), 1983, pp. 57-68

[19] White, J.E., "Telescript technology: the foundation for the electronic marketplace", *White Paper*, General Magic, Inc. 1994.

[20] Zayas, E., "Attacking the Process Migration Bottleneck", *Proc. of 11th ACM Symposium on Operating Systems Principles*, 1987, pp. 13-24.