Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications

T.C. Nicholas Graham and Tore Urnes *
York University
4700 Keele St., North York
Canada M3J 1P3
graham@cs.yorku.ca urnes@cs.yorku.ca

Abstract

Multi-user applications support multiple users performing a related task in a distributed context. This paper describes *Weasel*, a system for implementing multi-user applications. *Weasel* is based on the *relational view model*, in which user interfaces are specified as relations between program data structures and views on a display. These relations are specified in *RVL*, a high-level, declarative language. Under this model, an application program and a set of *RVL* specifications are used to generate a multi-user application in which all issues of network communication, concurrency, synchronization, and view customization are handled automatically. These programs have a scalable distribution property, where adding new participants to a session does not greatly degrade over-all system performance. *Weasel* has been implemented, and was used to generate all examples in this paper.

CR Categories: D.2.2 (Software Tools and Techniques for User Interfaces); D.1.1 (Applicative Programming); D.3.4 (Compilers and Runtime Environments)

General Terms: Human Factors, Languages

Additional Keywords: Multi-User Applications, Automatic Distributed Implementation. Semi-Replicated Architecture

*Extended version of the paper appearing in Proceedings of Computer Supported Cooperative Work, CSCW'92, November 1992.

1 Introduction

Multi-user applications support multiple users performing some related task in a distributed context. Examples of multi-user applications include groupware systems, which provide computer support to group activities [9]. Multi-user applications are hard to implement due to a competing set of requirements [6, 9, 14, 20, 21]:

Evolutionary design: user interfaces supporting cooperative work cannot be designed *a priori*, but must evolve based on experimentation;

Customizability: Sometimes all users should see the same thing on their display; users should also be able to customize these views without modifying those of other participants;

Mixed Control: Sometimes one or more users should control the session; sometimes the application should take charge; sometimes both should execute concurrently;

Distributed Implementation: Applications must run efficiently in a distributed context; they should be *scalable* so that new participants joining a session do not significantly slow down over all performance.

In the *Weasel* system, we have developed tool and language support for creating multi-user applications that moves a significant distance toward meeting these goals. In particular, *Weasel* allows the automatic distributed implementation of multi-user applications: the programmer must provide a central application program, written in a traditional imperative programming language, and a set of view specifications, written in the declarative language *RVL*. From these specifications, the *Weasel* system creates a distributed implementation, where issues of network communication, concurrency, synchronization, and customization are handled automatically. This automation simplifies the programming of multi-user applications, making them easier to prototype and evolve. Weasel supports a wide range of direct manipulation user interfaces based on buttons, menus and dynamic creation and layout of display objects. Weasel has been implemented to run on a network of Unix workstations, and was used to produce all examples in this paper.

One of the main advantages of *Weasel* over earlier systems for producing multi-user applications (e.g. [3, 21]) is that systems generated by *Weasel* have a *scalable distribution* property – that is, that adding new participants

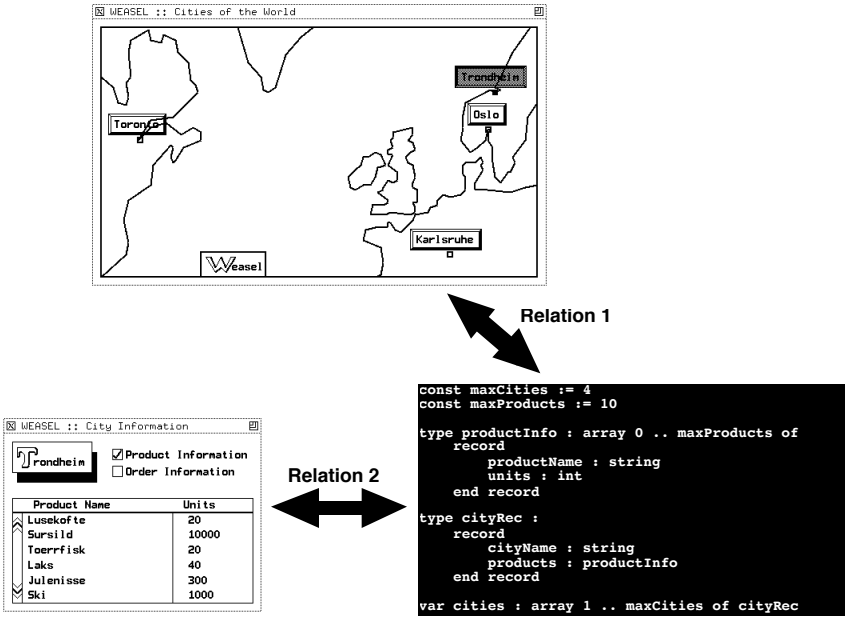

Figure 1: An inventory program as a pair of relational views. Users may click on a city, and view or modify product and order information for that city.

into a session does not significantly slow down the performance of the system as a whole. This scalability comes from the use of a novel semi-replicated distribution scheme. Section 3.1 shows timing results that demonstrate that a participant in a 10-person session waits only 50% longer for updates than a participant in a single-person session. Applications generated by *Weasel* are fast enough to use, but not yet fast enough for a production environment. Many optimizations are planned, but in the meantime *Weasel* is an interesting tool for rapid prototyping.

The paper is organized in two major sections. The next section introduces the relational view model (upon which *Weasel* is based), and the *relational view language* that is used to specify views. Section 3 then describes the mapping of relational views onto a network of workstations, and empirically evaluates the performance of systems generated by *Weasel*.

2 The Relational View Model

The motivation behind the relational view model of user interface construction is that user interfaces should be specified completely separately from application programs. As is widely cited [12], the main advantage of this *dialogue independence* is that user interfaces can evolve over time without impacting an operational application program. This strict separation brings the side-benefit that user interfaces can be easily distributed over a network, forming the basis of the *Weasel* implementation.

Under the relational view model, application programs are written in a traditional imperative language (in our case, Turing [15], a modular, Pascal-like language). The user interface consists of a set of *views*, graphical displays of data, which the user can manipulate directly using various input devices such as a mouse and keyboard. Every view is bound to the application program via a *relation*: whenever the program data changes, the picture is automatically updated to reflect the change, and whenever the user modifies the view through direct manipulation, the program data is automatically updated to reflect the change. These relations are specified in a special language called *RVL*, the Relational View Language, as described in section 2.1.

The paper contains various examples of relational views. Figure 1 shows an inventory program, where users at various sites can access order and inventory information by clicking a city on a map. The two views are specified by two different relations, each with the same data structure in the program.

Each user receives the same views, but *customized*, so that if one user clicks on a different city, the other users' views do not change.

Figure 2 shows a card-file name and address data base. Here, clicking on a letter moves to that position in the data base. Those letters for which there is no card are greyed-out. The data in the cards themselves can be edited by direct manipulation.

Figure 7 shows a tic-tac-toe game for two players. A strict turn sequence is observed, and both players view the current version of the board. The tic-tac-toe board is a *WYSIWIS* (What You See Is What I See) view [28], where a modification to the board by one player causes an identical modification to the other player's board.

Differing from the popular active values approach [27], relational views are not bound to individual variables, but to entire data structures, as delimited by Turing's module construct. Section 2.2 shows how this approach provides an implicit synchronization between the application and the user interface, so that updates only take place at sensible times.

Certain parts of a viewed data structure belong truly to the application, while other parts are used only to control the display. For example, in the inventory program, the inventory levels are part of the application data structure, whereas the current city that the user has selected is part of the display data state (figure 6). In a multi-user context, these display variables are automatically replicated so that each user receives a customized view. Section 2.3 shows how this customization is achieved.

The following three sections describe the relational view language, how relations are bound to data structures in the program, and how relations are customized in a multi-user context.

2.1 The Relational View Language

Relational views are specified in the *Relational View Language (RVL)*, a version of the *GVL* output language [4], extended to handle input. *RVL* specifications are functions from data structures in the application to display views. These functions can be annotated with Garnet-style interactors [18] to allow input. *RVL* is a powerful functional language based on a small set of primitive functions such as lines, arrows, boxes, circles and text, and providing abstraction through function definitions. The language supports limited geometric constraints, and automatic tree and graph layout. Since programmers do not have to worry about state, control flow, or layout, *RVL* has a specificational flavour.

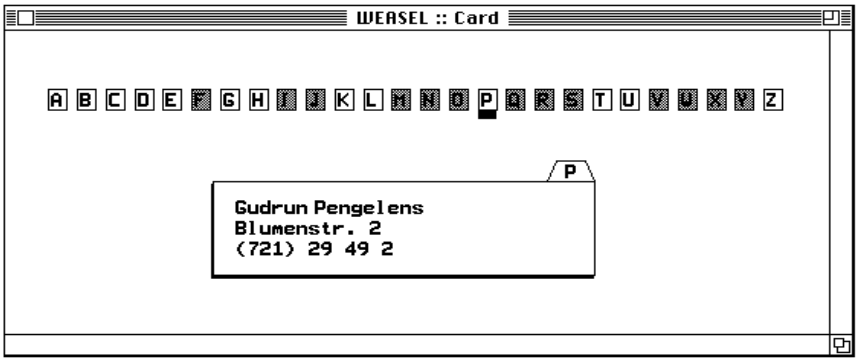

Figure 2: A name and address card file as a relational view: clicking a letter moves to that position in the file; the text of cards may be modified by direct manipulation, updating the underlying data structure.

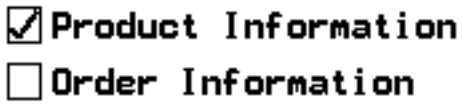

Figure 3: A Toggle Box – Part of the User Interface of figure 1

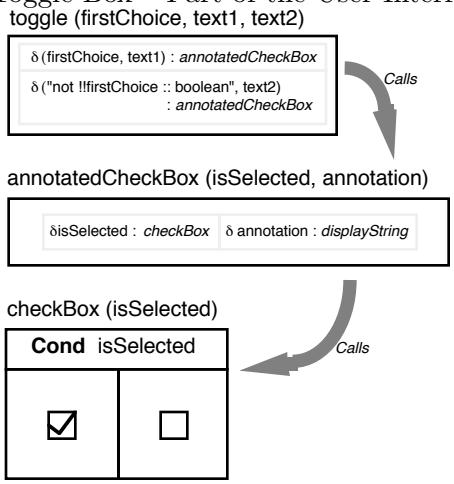

Figure 4: Specification of a Toggle Box

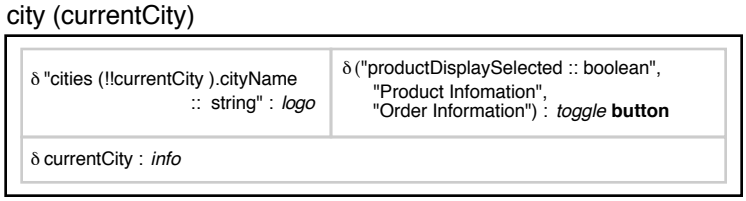

Figure 5: Application of the Toggle Box

Figure 4 shows an example *RVL* specification for a part of the inventory program of figure 1. In this example we use a visual syntax for *RVL*, as defined in [4]. Work is currently underway to develop a visual editor for a variant of the RVL language [26]; in the meantime, a textual version of RVL is used.

This specification is used to display a toggle box that selects between two projections of information: either *Product Information*, giving inventory information for that city, or *Order Information*, showing outstanding orders between two cities. The current selection is indicated with a checkmark. Clicking the mouse anywhere on the checkboxes or text changes from one view to the other. The toggle box itself is shown in figure 3.

The togglebox is specified as a set of three functions that map the current state of the application to a picture on the display, where the correct projection is checked. The checkbox itself is specified with the function *checkBox*, which takes a single parameter *isSelected*. This parameter is a boolean value from the application that determines whether this box should be checked or not. The function is programmed using a *cond* construct: if *isSelected* is true, the picture on the left is displayed, and if false, the picture on the right is displayed.

To place a text label beside a checkbox, the function *annotatedCheckBox* is used. This function takes two parameters, *isSelected* to determine whether this box should be checked or not, and *annotation*, which is the text label. The checkbox is drawn by applying the *checkBox* function to the parameter *isSelected*: this is accomplished with the syntax:

$$\delta \text{ isSelected} : \textit{checkBox}$$

which can be read as: "display *isSelected* as a *checkBox*". The text label is

displayed with the built-in function *displayString*. The grey boxes surrounding function calls indicate where the resulting picture is to be placed; these boxes are sized to fit, so in this case, the text label appears immediately to the right of the checkbox.

The full toggle box is then specified with the *toggle* function. The parameters to *toggle* indicate whether the first or second choice is to be used, and specify the two text annotations. The two lines are displayed one above the other, using the *annotatedCheckBox* function. In the upper line, the *firstChoice* parameter is used to indicate whether the box is to be checked or not. In the lower line, *firstChoice* is negated, so that the lower line is checked whenever the upper line is not. This shows how specifications can make use of expressions in the underlying programming language (in this case Turing): the expression

"**not** !!firstChoice :: boolean"

is to be evaluated as a Turing expression. The "*:: boolean*" part is the type of the expression; the "*!!firstChoice*" notation allows the embedding of an *RVL* parameter in a Turing expression.

2.1.1 Turning Views into Relations

The *toggle* function is a general function independent of any data structure in the application program. This generality provides the power of *RVL*: graphical techniques can be encapsulated in functions that are completely independent of any context. To use the functions, they must be applied to some data structure. Figure 5 shows how this is accomplished. Here, a specification for the entire relation of figure 1 is shown. The function *city* takes one parameter, specifying which city the user has currently selected. The function *logo* is used to display the city name in a fancy style, while information about the city is generated with the *info* function (not defined in this paper). The product/order information toggle box is displayed with the following application of the *toggle* function:

δ ("productDisplaySelected :: boolean,
"Order Information" ,
"Product Information") : *toggle* **button**

Here, *productDisplaySelected* is an application variable indicating whether the user has selected the product or order display. The function call is annotated with the *button* directive: this states that the result of the function,

that is the graphical representation of the toggle box, should be defined as a button. The button is implicitly bound to the first parameter of the function call, the application variable *productDisplaySelected*. Whenever the button is clicked, the sense of *productDisplaySelected* is changed: i.e., if the variable has the value *true*, it becomes *false*, and vice versa. Whenever the value of *productDisplaySelected* changes because of a button click, the view is automatically regenerated, causing the check mark to move to the other box.

Thus, *toggle* by itself is only a function from the data state of the application to the display state. By adding the *button* interactor to an application of *toggle*, it becomes a relation: direct manipulation of the screen image of the toggle box can effect changes in the data state, just as changes in the data state can effect changes in the display view.

There are a series of such annotations to allow input, inspired by the interactor concept of *Garnet* [18, 19]. The *interactive* interactor allows direct editing of text or numeric data on the display. The *count* interactor is another kind of button that increments (or decrements) an integer counter every time it is clicked. The *signature* interactor is another kind of button; when clicked, it writes an integer uniquely identifying this participant into an integer variable.

2.2 Binding Views to an Application

Under the relational view model, application programs are not just passive entities awaiting user input, but may perform computation in their own right. For example, in the inventory program of figure 1, the program might be initiating orders and monitoring production levels at the same time as providing views to users. Such an organization is called *mixed-control* between the application and the user interface [12]. Mixed control can lead to difficult synchronization problems: the application should not update a value at exactly the same time as the regeneration of a view depending on the value; user-input should not cause a change in a value at exactly the same time as it is being used by the application program.

This synchronization is achieved automatically, by considering views to be related to entire data structures, not just to individual variables. A data structures can be encapsulated in a module. A module defines an abstract data type, that is, a data structure and a set of operations on the data structure. This means that the only code that has direct access to the data structure must be within the module. The body of the module is

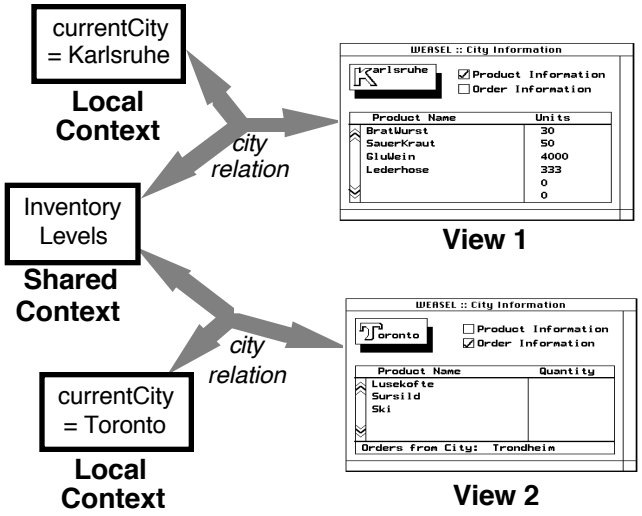

Figure 6: The same relation can be used to generate two different views by parameterizing each relation by a local context.

then treated as a critical region, so that only one view or the application program may modify the module state at a time. The application programmer may then assume two guarantees which make it appear as if he/she is programming in a purely sequential environment:

- Whenever a module operation is executing, the data structure is immutable; that is, user actions will be buffered until the module operation is complete;
- No view based on a module will be updated while a module operation is executing; that is, the programmer does not have to worry about temporarily bringing the module into a bad state that could cause the generation of an incorrect view.

The one new possibility with which a programmer must contend is that between calls to a module, the state of that module may change. Therefore, the programmer may not assume that a query operation provided by a module will deliver the same result twice in a row. In the examples we have created up to now, this restriction has not proved to cause difficulties.

2.3 Customization of Views

Customization is required in multi-user applications for two purposes. First, as in the inventory program of figure 1, different users may see the same

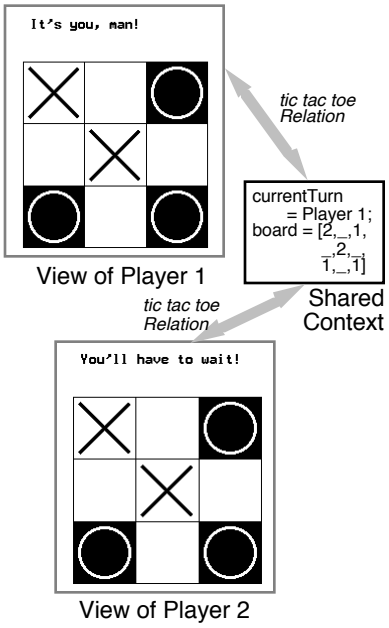

Figure 7: A Tic-tac-toe game as a relational view

view, but wish to specialize them locally. In the inventory example, all users' views are maintained by the same *city* relation (figure 6), but should be able to view different cities. We call this form *transparent customization*, since it should happen automatically. Secondly, relations may be bound to the central data structure differently according to who the user is. In the tic-tac-toe game, this allows two users to see the same board, but to receive different messages indicating whether it is their turn or not (figure 7). This form is called *signature* customization, meaning that the relation is implicitly parameterized by the signature (or name) of the particular client. Signature customization provides the basis for programming *collaboration awareness* [7], where participants are made aware of the activities of other members of a group.

2.3.1 Transparent Customization

Transparent customization is provided automatically through the mechanism of binding views to application programs.

One of the primary goals of the relational view model is to separate the specification of the user interface completely from the application program itself. This separation becomes blurred when variables are introduced into the application data structure that are only there to control the display. For example, in the inventory program of figure 1, the inventory levels belong

truly to the application, while the currently selected city is a variable that is introduced to help manage the display.

In order to regain the desired separation, a special *interface* section is introduced into any module that is to be displayed. In this section all display variables must be declared, separately from the application variables. Display variables may not be viewed or modified in the application program itself; they are only available for use in *RVL* specifications. To the *RVL* specification, there is no difference between display and application variables: they are both available for viewing and modification. This split makes user interfaces easier to modify, since display variables can be added or removed with no danger of impacting the application itself in any way.

The key to transparent customization is that these display variables are automatically replicated for each display (figure 6). In the inventory program, each user receives his/her own version of the current city, whereas the central inventory information is shared among all views. Following the terminology of Ellis et al. [9], this local state is called the *local context*, whereas the central data structure is called the *shared context*.

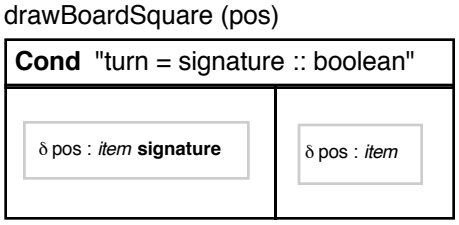

Figure 8: The squares on a tic-tac-toe board are sensitive to input only if it is this player's turn.

2.3.2 Signature Customization

Figure 7 shows a tic-tac-toe game as a relational view. Here, players take turns clicking on squares to indicate their moves. The underlying data structure is a 3×3 array representing the board state, and a variable representing whose turn it is. In this game, each player's view is customized in three ways, depending on whose turn it is: firstly, each player receives a message telling him/her either to make a move or to wait for the other player; secondly, the

view of the active player is sensitive to input, whereas that of the waiting player is not; thirdly, when player one clicks on a square, an "*O*" is to be displayed, whereas player two's actions generate an "*X*". All this customization is to be achieved even though the same specification is used for both players.

One simple *RVL* mechanism accomplishes all three of these tasks, as shown in figure 8. This function draws one square of the board, displaying either an "*X*", an "*O*", or a blank square as appropriate. The parameter *pos* indicates which element of the underlying data structure is to be used to draw the position.

To make a square sensitive to input, it is tagged with the *signature* interactor:

$$\delta \text{ pos} : \textit{item} \textbf{ signature}$$

This displays the position *pos* as an *item* (i.e., a square with an "*X*", "*O*", or blank), and states that when the square is clicked, the *signature* of this user is to be written into *pos*. Each participant (i.e., each player) automatically receives a unique integer signature, thus providing a means of recording which player clicked which square, even though the same *RVL* specification is used for all players.

This signature is also available for use in expressions. To test whether it is this player's turn or not, the expression:

cond "turn = signature :: boolean"

is used. Here, *turn* is an application variable indicating whose turn it is, and *signature* is a predefined parameter whose value is the unique signature of this view. This *cond* construct therefore chooses whether or not to apply the *signature* interactor to this square, depending on whether it is this player's turn. A similar *cond* expression is used to notify the player whether he/she should play or wait.

According to Patterson et al. [21], this ability to support signature customization is one of the defining differences between multi-user applications and groupware.

3 Distributed Implementation of Relational Views

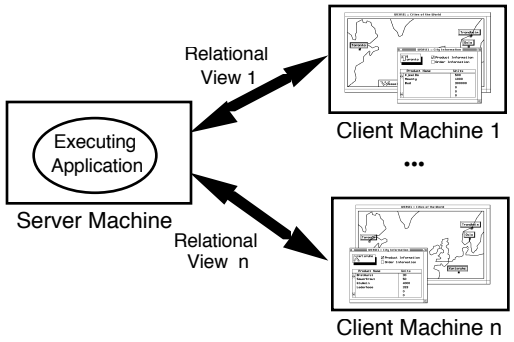

Figure 9: Distributed Organization of Relational Views

In a network of workstations, *Weasel* programs are organized following a client/server model (figure 9). The application program runs on a central workstation called the *server*. Each user works on a *client* workstation, which communicates with the server to access and modify application data.

Two goals form the key to this distributed implementation:

- Put as much useful computation as possible on the client machines;
- Keep the network traffic to a minimum.

The bottleneck of a client/server organization is that adding new clients increases the load on the server and the network. Eventually, the server or network becomes overloaded, and performance degrades unacceptably for all clients. By putting more computation on the client machines, scalability is aided, since each added client takes away from the total work to be done. We call this scheme *semi-replication*, since the main application remains on the server machine, while all code and data related to the user interface is replicated to the client machines. It is also critical to keep network traffic low, since network communication is slow; too much reliance on the network can lead to a system that is scalable, but too slow to be useable.

Figure 10 shows how the computation is split between client and server. The application program itself executes on the server, while the entirety of the view computation occurs on the client. The heart of the client computation is the *RVL* interpreter. Given an *RVL* specification, this maps the

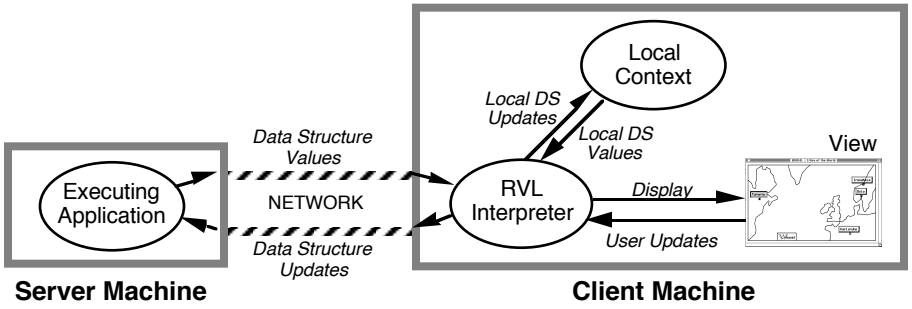

Figure 10: Semi-Replicated Distribution of Relational Views

current version of the application data state to a display view. The interface between the client and server is on the level of expression evaluation: whenever a Turing expression is reached in the *RVL* specification, a message is sent to the server, asking for the value of the expression. The expression value is returned, and interpretation of the *RVL* specification can continue.

Typical *RVL* specifications contain few expressions, so this interface requires little network communication. The cost of adding new clients is also small: since the entire user interface computation occurs on the client, the only cost that a new client introduces is an increased traffic of expressions to be evaluated. Since this number of expressions is small, large numbers of clients execute only minimally slower than smaller ones (section 3.1).

As an optimization, all *display* variables (section 2.3.1) are represented locally on the clients (figure 10). We call this data state the *local context*. This optimization allows many views to be computed without reference to the server whatsoever. As an additional optimization, the *RVL* interpreter performs expression caching so that the same expression does not get evaluated over the network twice in one view generation. We are examining extending expression caching to the local context, so that re-evaluation would not occur between view generations either.

The main disadvantage of *Weasel's* interface between client and server is that communication is synchronous: that is, whenever a client requests an expression, it must wait for a response before continuing. There are various possible optimizations to turn much of this synchronous communication into

an asynchronous form, where computation can continue once a message has been sent. We are now examining reimplementing RVL using a form of parallel graph reduction [22] to allow computation on the client machines to continue while network requests are pending.

One of the pleasant advantages of semi-replicated distribution is that the system is robust to failure. The server never waits for clients, so a client crash cannot cause the server to hang. Clients depend on the server only for expression values – the state of the interface itself is represented locally. Therefore, if the server crashes, the clients can wait until the server returns to functional operation, and continue with their interface state preserved.

The current *Weasel* implementation has some rough edges, and therefore cannot yet be considered to be production quality. Some aspects of *RVL* need improvement. Ongoing modifications include the introduction of true second-order functions, and the introduction of interactive display views as true first-class values in the language. The *Weasel* implementation is described in detail in [10, 31].

3.1 Evaluation of Implementation

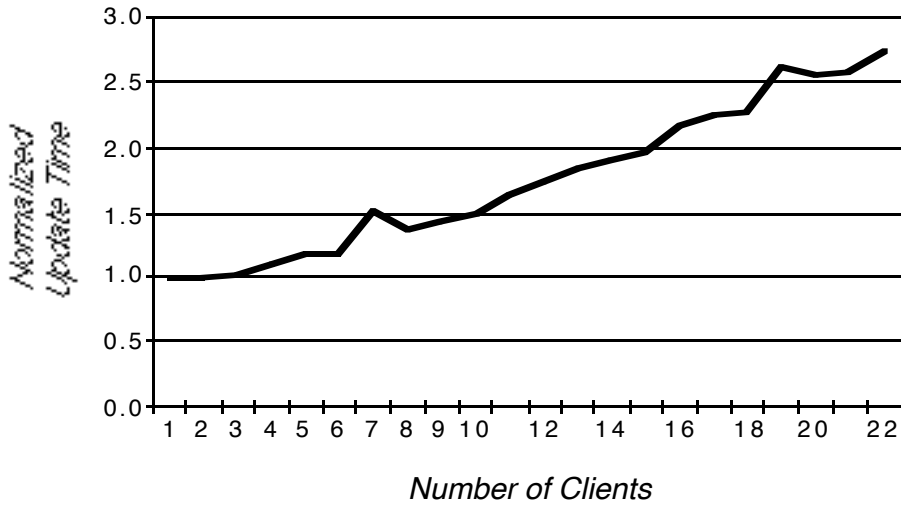

Figure 11: Normalized Cost of Updating One Client View as a Function of Number of Clients

In order to determine whether our means of distribution is effective or not, we performed some simple timings on a network of Sun workstations at York University. We used a SparcStation 10/30 workstation with 64MB memory as the Weasel server, and a collection of 22 SparcStation 10/20 stations as clients. Our testing was intended to determine the cost of bringing new users into a group session: specifically, it was hoped that bringing in new users would only minimally slow down the performance of the users already taking part in the session.

As an example program, we used a modified version of the inventory program from figure 1. All optimizations were removed from the program in order to give the most pessimistic results possible. In particular, all data was represented on the server rather than some being represented locally, and every view on every machine was calculated from scratch every time an update was made. These pessimizations served to maximize the exchange of data required to generate a new view.

We first used the server and just one client, measuring the amount of time required to update the display following a modification. We then successively added new clients, each one on a different machine. Each new client received the same view as the first one, so that an update of any one view triggered a recomputation of the views on every machine. In the final case, with 22 machines and two views per machine, this meant a total of 44 views being updated simultaneously.

At each stage, we measured the time that the first client had to wait during view generation, thus providing a measure of the degradation of performance witnessed by this client as other participants join the session.[1] The timings were performed in real time, and 10 measurements were taken and averaged for each new client. These timing results were normalized[2] so that an update in a one client system has a cost of 1, the results of which are shown in figure 11.

These figures show that the time per client to update views does rise as new clients are introduced, but that the increase is modest: in a 10 client session, clients take approximately 1.5 times as long to make updates as in a 1 client session; in a 20 client session, this number increases to approximately 2.5 times longer.

As a comparison, we tested the scalability of a centralized version of the

[1] The other clients all experienced approximately the same times, so that updates were practically simultaneous on all machines.

[2] The average time on a 1 client system was 0.7 seconds with a standard deviation of 0.03 seconds over 10 trials.

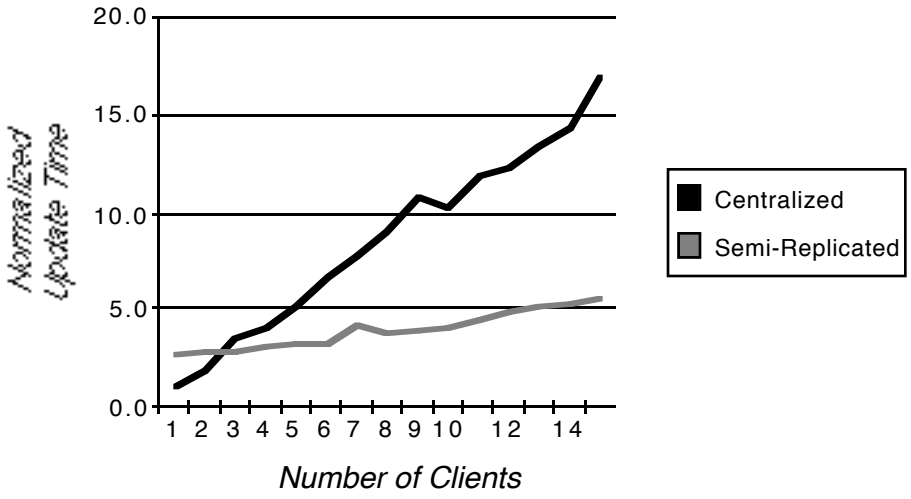

Figure 12: Cost of updating one client view in a centralized view computation model; the Weasel semi-replicated architecture is shown for comparison

program. In this version, all views are computed on the server, and X is used to display the views remotely on other machines; this is the approach used in the RENDEZVOUS system [21]. Figure 12 shows that when executed on the same server as in the distributed case, this version has an approximately linear cost increase as new clients are introduced, where the normalized cost of ten clients (10.3) is ten times the cost of one client (1.0). The semi-replicated version is also shown for comparison. With the particular configuration of machines used, semi-replication outperforms the centralized version for any number greater than two clients.

From these figures we can conclude that the *Weasel* form of distribution demonstrates quite reasonable scalability, especially compared to the centralized form of computation used in some other systems. In terms of absolute speed, the current *Weasel* implementation is fast enough to be usable, but not fast enough to be production quality. The optimizations listed in this section show promise for achieving sufficient speedup without sacrificing scalability.

4 Related Work

In this section we relate our work to other tools for developing multi-user applications. We start by comparing the *relational view model* to other attempts to link applications and user interfaces. Next, we focus on the *Weasel* implementation and how its architecture relates to those of other systems for implementing multi-user applications.

4.1 Connecting Application and User Interface

The relational view model builds from earlier work in the *conceptual view model* of output [11]. The output component of *RVL* is based on the *GVL* language [4], extended with interactors motivated by the Garnet system [19].

Several other systems [14, 19, 32] use constraints to achieve the same effect as *Weasel*'s relational views. In these systems, certain objects are predefined as being display objects. Display objects can be related to data in the application using explicit logical constraints. The constraints must always be maintained, so modifications in either the application or the display data automatically trigger updates in the other. Constraints are a much more general system than relational views: any constraint may be written over any data value of any object. Relational views make a clear separation between display maintainence (as accomplished by the *RVL* specifications), and the application program. It is this clear separation that makes it possible to create an effective distributed implementation.

The Trip-2 system [29] is also a realization of the relational view model. In this system, relations are specified in Prolog rather than *RVL*. Prolog is far more general than *RVL*, but the authors report severe performance difficulties.

The RENDEZVOUS system [14, 20, 21] also provides high-level support for creating groupware and multi-user applications, and was perhaps the single biggest influence on our work. This system provides declarative support for output, implicit concurrency through Garnet-style constraints, and distribution based on the X Window System [25].

RENDEZVOUS is based on the *Abstraction-Link-View* (ALV) architecture model [13]. In this model, the application data state (the *Abstraction*), and the display data state (the *View*) are programmed explicitly in an object-oriented style. Constraints (the *Link*) are used to keep the two consistent. ALV is similar to our relational view model, in that the constraints specify a relation between view and application, and that this rela-

tion is automatically maintained by the system. The relational view model is somewhat higher-level than ALV, in that the view state is automatically inferred from an *RVL* specification, whereas in ALV it must be programmed. ALV, on the other hand, gives the programmer more control over the relation between application and display, potentially leading to more efficient incremental display updating.

4.2 Architectures for Multi-User Applications

There are two basic architecture alternatives to consider when implementing interactive multi-user applications. The first alternative is the *centralized* architecture which is basically a client-server architecture where a single instance of the multi-user application is shared by all users. The second alternative is called a (fully) *replicated* architecture. Under this architecture a copy of the multi-user application is replicated to each user, typically executing locally on the user's workstation.

Shared window system researchers have investigated benefits and drawbacks of the two basic architectures [5, 17]. Analyses can also be found in for example [7] and [8, pp 456–457]. Briefly, the advantage of using a centralized architecture is its simplicity in terms of implementation. This is due to the fact that only one copy of the multi-user application exists. The major drawback of a centralized architecture is, as we demonstrated in section 3.1, the rapidly increasing response times experienced by all users as the number of users increases. This is because a centralized architecture is unable to employ a larger number of processors as the number of users increases.

The fully replicated architecture, on the other hand, offers a response time which is fairly independent of the number of users. The reason being that all responses are generated on the (local) workstations on which the users are working. However, fully replicated architectures are notoriously difficult to implement correctly. The critical issue is to maintain consistency across all the application replica. Consider the race condition inherent in situations where concurrent input is allowed and WYSIWIS views must be maintained. If user A and user B make almost simultaneous inputs, then A's application copy will receive A's input before B's input while B's copy will see the reverse order. The problem is, of course, that interactive systems must react instantaneously to input, so traditional (e.g. distributed database) solutions to this problem will typically be too slow. In addition to synchronization problems of the kind just indicated, replicated architectures

further complicate matters by executing semantic operations multiple times, once for each application copy. This can lead to great problems, for example when doing file I/O in a replicated architecture setting.

Full replication also leads to difficulties when bringing late-comers into a group session [16, 3]. Since there is no globally available state, fully-replicated systems may have to keep a full history of actions performed in order to allow a late-comer to derive the state currently held by other participants.

The Weasel system combines the centralized and replicated appraoches into an architecture we have called *semi-replicated*. The motivation is to have the benefits of the two basic architectures while avoiding the drawbacks. As shown in section 3, we have replicated the view generation process to each of the client workstations, i.e. responses are generated locally as in the fully replicated architecture. However, as in the centralized case, we still have a single copy of the application and hence neither need to consider sophisticated synchronization schemes nor multiple time execution of semantic operations. Semi-replication solves the late-comer problem, since all shared information is available from the server machine. In Weasel, latecomers are handled completely automatically. It should be noted that even though we have significantly improved response times relative to those of a centralized architecture, a fully replicated architecture would probably offer even faster responses.

The Multi-user Suite [7] group, independently from our work, developed an architecture similar to the semi-replicated Weasel architecture. The idea behind the Multi-user Suite framework for implementing multi-user applications is to use editing of shared data structures as a model for multi-user interfaces. The views that can be generated in Multi-user Suite are basically text-only views, and users are expected to interact with applications by executing long transactions which are explicitly committed. Consequently, view updates are performed less frequently than in applications generated by Weasel where users interact by direct manipulation. This, in turn, means that delays resulting from fetching global context data is less of a problem in Multi-user Suite.

The RENDEZVOUS system [21] described above uses a centralized architecture. A single application process and a set of view generation processes, one for each user, execute on a single workstation. The centralized version of Weasel described in section 3.1 has the same architecture. This means that multi-user applications generated by RENDEZVOUS suffer from

scalability problems[3].

Groupkit is toolkit [23] for developing groupware. Groupkit provides a simple and elegant specification language suitable for rapid prototyping. Applications are based on a replicated architecture, and hence have good performance; however, the potential exists for synchronization problems to occur. Groupkit provides no general solution for handling latecomers, requiring sometimes complex hand-coding to solve the problem.

A system called Group Interaction Environment (GroupIE) [24] is an object-oriented[4] toolkit offering development and run-time support for distributed multi-user (group) applications. An object-oriented, layered distribution support and management system takes care of distribution issues. It offers services like object migration (e.g. relocation of in-demand interaction objects at local workstations), remote method calls, and naming and location support. The GroupIE architecture is more flexible than the Weasel architecture since it can change dynamically. It is unclear how synchronization issues are handled in the GroupIE system.

5 Conclusions

The computer support of cooperative work involves the programming of multi-user applications which aid a group of people in working together on some task. Such programs are difficult to create because of the conflicting demands of rapid prototyping, concurrency and distribution.

This paper has presented *Weasel*, a system for programming multi-user applications. Through declarative view specifications in the *Relational View Language*, normal, imperative application programs can be turned automatically into programs that run in a distributed environment. The paper showed how issues of concurrency, synchronization, and customization of views are handled automatically in *Weasel*. Timing results demonstrated that the implementations generated by *Weasel* have a scalable distribution property, where adding new users to a session does not severely impact the performance of the system as a whole.

Much future work remains with *Weasel*, in terms of optimizing the implementation, and increasing the flexibility and functionality of the *RVL*

[3]The RENDEZVOUS group along with for example [2, 30] advocate the use of an architecture similar to that of Weasel as a good underlying architecture for interactive multi-user applications. It is unclear whether these authors have experimented with a semi-replicated architecture in a multi-user setting.

[4]The GroupIE system is implemented in Smalltalk-80.

specification language. One interesting line we are persuing now is how to automatically distribute parts of the application program itself to the clients, so that communication is further reduced.

Acknowledgements

Weasel was implemented by the authors, James R. Cordy, and Stefan Hügel. Thanks are due to Prasun Dewan, Saul Greenberg and Thomas Rüdebusch for making their systems available to us. This work was partially funded by NSERC, NTNF, the ESPRIT Basic Research Action 3147 (the Phoenix project), and NTH, Norway.

References

[1] Baecker, R.M. *Readings in Groupware and Computer-Supported Cooperative Work, Assisting Human-Human Collaboration.* Morgan Kaufmann Publishers, ISBN 1-55860-241-0, 1993.

[2] Beaudouin-Lafon, M. and Karsenty, A. Transparency and Awareness in Real-Time Groupware Systems. In *Proceedings of the Fifth Annual Symposium on User Interface Software and Technology, (Monterey, California, Nov. 15–18)*, pages 171–180. acm press, 1992.

[3] Chung, G., Jeffay, K., and Abdel-Wahab, H. Accomodating latecomers in shared window systems. *IEEE Computer, Project Overviews*, 26(1):72–74, January 1993.

[4] Cordy, J.R. and Graham, T.C.N. GVL: Visual specification of graphical output. *Journal of Visual Languages and Computing*, 1992.

[5] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and Tomlinson, R. MMConf: An Infrastructure for Building Shared Multimedia Applications. In F. Halasz, editor, *Proceedings of the Third Conference on Computer-Supported Cooperative Work (Los Angeles, Ca., Oct. 7–10)* (also in [1]), pages 329–342. ACM Press, 1990.

[6] Dewan, P. and Choudhary, R. Primitives for programming multi-user interfaces. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 69–78. ACM, acm press, November 1991.

[7] Dewan, P. and Choudhary, R. A high-level and flexible framework for implementing multiuser user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[8] Dix, A., Finlay, J., Abowd, G., and Beale, R. *Human-Computer Interaction.* Prentice Hall, ISBN 0-13-458266-7, 1993.

[9] Ellis, C.A., Gibbs, S.J., and Rein, G.L. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[10] Graham, T.C.N. Conceptual views of data structures as a programming aid. Technical Report 88-225, Department of Computing and Information Science, Queen's University at Kingston, August 1988.

[11] Graham, T.C.N. and Cordy, J.R. Conceptual views of data structures as a model of output in programming languages. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.

[12] Hartson, H.R. and Hix, D. Human-computer interface development: Concepts and systems. *ACM Computing Surveys*, 21(1):5–92, March 1989.

[13] Hill, R.D. The abstraction-link-view paradigm: Using constraints to connect user interfaces to applications. In *ACM SIGCHI 1992*, pages 335–342, April 1992.

[14] Hill, R.D. Languages for construction of multi-user multi-media synchronous (MUMMS) applications. In B. A. Myers, editor, *Languages for Developing User Interfaces*. Jones and Bartlett, 1992.

[15] Holt, R.C. and Cordy, J.R. The Turing programming language. *Communications of the ACM*, 31(12):1410–1423, December 1988.

[16] Lauwers, J.C. and Lantz, K.A. Collaboration Awareness in Support of Collaboration Transparency: Requirements for the next Generation of Shared Window Systems. In *Proceedings of CHI'90* (also in [1]), pages 303–311. ACM Press, 1990.

[17] Lauwers, J.C., Lantz, K.A., and Romanow, A.L. Replicated Architectures for Shared Window Systems: A Critique. In *Proceedings of the Conference on Office Information Systems, Cambridge, MA* (also in [1]), pages 249–260. ACM Press, April 1990.

[18] Myers, B.A. Ideas from Garnet for future user interface programming languages. In B. A. Myers, editor, *Languages for Developing User Interfaces*. Jones and Bartlett, 1992.

[19] Myers, B.A., Giuse, D.A., Dannenberg, R.B., Vander Zanden, B., Kosbie, D.S., Pervin, E., Mickish, A., and Marchal, P. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85, November 1990.

[20] Patterson, J.F. Comparing the demands of single and multi-user applications. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 87–94. ACM, acm press, November 1991.

[21] Patterson, J.F., Hill, R.D., Rohall, S., and Meeks, W.S. Rendezvous: An architecture for synchronous multi-user applications. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 317–328. ACM, October 1990.

[22] Peyton Jones, S.L., Clack, C., and Salkild, J. High-performance parallel graph reduction. In *Proceedings of PARLE 89*, pages 193–206, 1989.

[23] Roseman, M., Yitbarek, S., and Greenberg, S. Groupkit reference manual: A guide to its architecture, interprocess communication, and programs. Included in the public domain Groupkit distribution, available by anonymous ftp from `ftp.cpsc.ucalgary.ca` under `pub/grouplab/software`, December 1993.

[24] Rüdebusch, T.D. Development and Runtime Support for Collaborative Applications. In H.-J. Bullinger, editor, *Proceedings of the Fourth International Conference on Human-Computer Interaction, Stuttgart, Germany*, Human Aspects of Computing, pages 1128–1132, Amsterdam, 1991. Elsevier Science Publishers.

[25] Scheifler, R.W. and Gettys, J. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[26] Song, G. Mixing visual and textual programming in functional languages. Master's thesis, York University, North York, Canada, August 1994. (expected).

[27] Stefik, M.J., Brobow, D.G., and Kahn, K.M. Integrating access-oriented programming into a multi-paradigm environment. *IEEE Software*, pages 10–18, January 1986.

[28] Stefik, M.J., Foster, G., Bobrow, D.G., Kahn, K.M., Lanning, S., and Suchman, L. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[29] Takahashi, S., Matsuoka, S., Yonezawa, A., and Kamada, T. A general framework for bi-directional translation between abstract and pictoral data. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, November 1991.

[30] Taylor, R.N. and Johnson, G.F. Separation of Concerns in the Chiron-1 User Interface Development and Management System. In *Proceedings of INTERCHI '93*, Human Factors in Computing Systems, pages 367–374. ACM Press (and Addison Wesley, ISBN 0-201-58884-6), 1993.

[31] T. Urnes. A relational model for programming concurrent and distributed user interfaces. Master's thesis, Norwegian Institute of Technology, University of Trondheim, 1992.

[32] Vander Zanden, B.T., Myers, B.A., Giuse, D., and Szekely, P. The importance of pointer variables in constraint models. In *Proceedings of the Fourth Annual Symposium on User Interface Software and Technology*, pages 155–164. ACM, acm press, November 1991.