Fiia: User-Centered Development of Adaptive Groupware Systems

Christopher Wolfe[1], T.C. Nicholas Graham[1], W. Greg Phillips[2], Banani Roy[1]

[1]School of Computing
Queen's University
Kingston, Canada K7L 3N6

{wolfe | graham | broy}@cs.queensu.ca

[2]Electrical and Computer Engineering
Royal Military College of Canada
Kingston, Canada K7K 7B4

greg.phillips@rmc.ca

ABSTRACT

Adaptive groupware systems support changes in users' locations, devices, roles and collaborative structure. Developing such systems is difficult due to the complex distributed systems programming involved. In this paper, we introduce *Fiia*, a novel architectural style for groupware. *Fiia* is user-centered, in that it allows easy specification of groupware structured around users' settings, devices and applications, and where adaptations are specified at a high level similar to scenarios. The *Fiia.Net* toolkit automatically maps *Fiia* architectures to a wide range of possible distributed systems, under control of an annotation language. Together, these allow developers to work at a high level, while retaining control over distribution choices.

Categories and Subject Descriptors

H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—Computer-supported cooperative work.

General Terms

Human factors, Languages

Keywords

Groupware development toolkit, groupware architecture

1. INTRODUCTION

Recent years have seen exciting advances in the design of groupware systems. One emerging trend is groupware that adapts over time to changes in participants' physical settings, hardware, roles and collaboration structure. We term this kind of system *adaptive groupware*.

Examples of adaptive groupware include *RAMSES*, a tool for organizing archaeological digs that allows users to move from their office PC to a PDA at the dig site [1]; *Mohoc*, a tool allowing health care workers to coordinate their activities with changing location and degrees of connectivity [23]; *OnStar*, a commercial product where drivers in a car are automatically connected to emergency operators in case of an accident [4]; *Age Invaders*, where "young" people play a game on a physical platform while their "aged" relatives join in from a web client [19]; and *Software Design Board*, which allows people to migrate between synchronous/asynchronous interaction, colocated/distributed location, and PC/electronic whiteboard devices [29].

In these examples, people interact with each other, but from very different settings (game platform vs web browser vs automobile), using very different user interfaces, and fulfilling different roles. Furthermore, these applications provide examples of adaptation to changes in users' settings, clients and roles over time.

Developing adaptive groupware involves enormous technical challenges. Implementing adaptations may require moving data and code between computers, dynamically connecting devices with radically different capabilities, and resolving partial failure at runtime, all while choosing and configuring consistency maintenance, concurrency control, caching and network transport strategies appropriate to the newly adapted application. Adaptive groupware is still, for the most part, developed using low-level networking and hand-coded distribution algorithms. This lack of high-level tools for runtime adaptation is impeding groupware developers from fully exploring the design possibilities of the wide range of interactive devices now available.

To address this problem, we introduce *Fiia*, a notation for describing the structure of groupware systems, and *Fiia.Net*, a development tool that helps render these designs into code. *Fiia* helps bridge the gap between design and implementation, allowing groupware developers to model their systems using a notation evocative of user-centered scenarios. The *Fiia.Net* toolkit provides code constructs directly implementing the features of these models, making it an easy step to translate *Fiia* diagrams into code. This removes much of the conceptual gap between architectural model and running system. We refer to *Fiia* as providing *user-centered* design of groupware systems, as it encourages developers to think of users in the context of their roles, physical settings and available devices. *Fiia* has the following main properties:

- *Fiia* designs are user-centred, allowing collaborative applications to be modeled from the point of view of the people who will be using them.
- *Fiia* is *application-level*, abstracting low-level details of distributed systems such as data replication, networking protocols and failure handling.
- *Fiia* provides high-level support for *runtime adaptation*: developers specify changes in participants' collaboration struc-

EICS'09, July 15–17, 2009, Pittsburgh, Pennsylvania, USA.

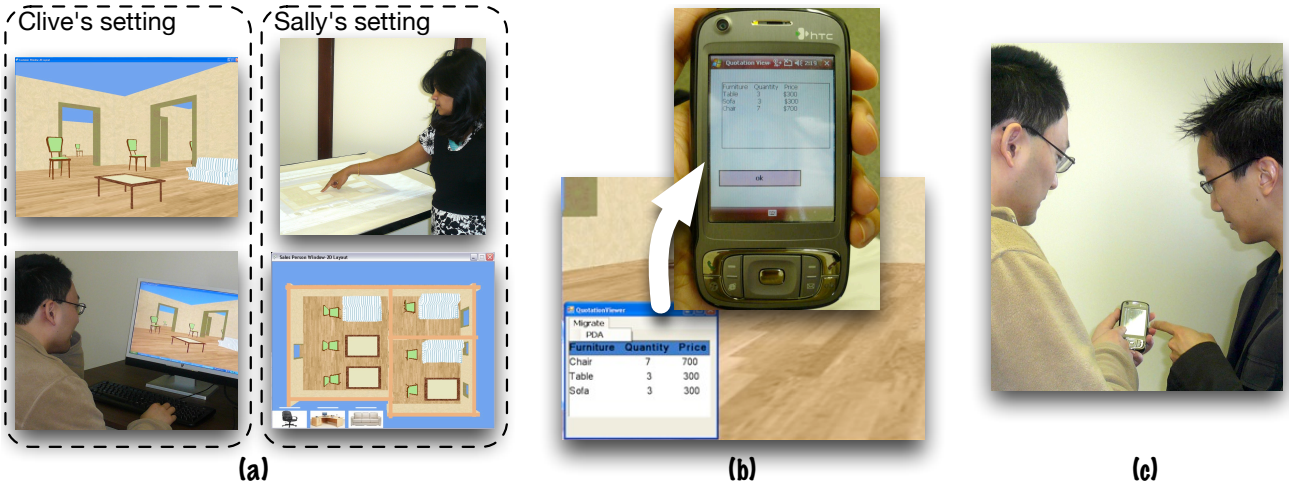

Figure 1: Scenes in the scenario: (a) Sally and Clive collaborate, (b) quotation viewer moved to PDA, and (c) in Bill's office.

ture, devices and location at a conceptual level, and the toolkit automatically enacts the changes on the distributed system.

The paper is structured as follows. We first introduce a scenario motivating the need for better development support for adaptive groupware systems. We discuss related approaches, and then introduce the *Fiia* notation, illustrating how it simplifies the development of adaptive groupware applications. Finally, we discuss the implementation of the *Fiia.Net* toolkit, and conclude with an evaluation of its performance.

2. MOTIVATING SCENARIO

Figure 1 shows an adaptive groupware system supporting a collaborative furniture purchasing task. The application has been implemented using the *Fiia.Net* toolkit.

2.1 Scene 1: Multi-Role, Multi-Platform Furniture Layout

"Clive" is moving to a new office, and is interested in purchasing furniture. He contacts "Sally", a sales agent for a furniture company. Sally starts a furniture layout program on her tabletop surface. The program shows a top-down view of Clive's office. Using gestures, Sally can drag furniture pieces onto the floor plan and orient them to fit. Meanwhile, Clive, who is sitting at a PC, sees a first-person view of the furniture layout, updated in real time in response to Sally's actions. He can use his program to "walk" through the virtual space representing his office. Clive and Sally communicate via a voice over IP link.

The first scene in our scenario illustrates considerable heterogeneity in the two participants' settings: *Roles:* Sally is guiding the furniture placement, while Clive views the results; *Hardware:* Sally is using an electronic tabletop, a horizontal, touch-sensitive surface with which she interacts using gestures, while Clive is using a traditional PC with mouse and keyboard; *Software:* Sally is using a top-down client that allows editing of the layout, while Clive is using a 3D, first-person virtual world view of the layout.

2.2 Scene 2: Adding a Quotation Viewer

The second scene introduces a simple form of runtime adaptation: *Sally wants to show Clive a quotation for the furniture picked so far. She initiates a quotation viewer, which appears on Clive's display. As new furniture is added, the quotation viewer updates automatically. Clive prefers not to have to flip windows, so moves the quotation viewer from his PC display to his PDA.*

Figure 1(b) shows how progress with the collaborative task has led to changes in the tools Clive and Sally are using to collaborate. There are two forms of runtime change: *Application:* A new application is started and used to share data; *Device:* An application is moved from one device to another.

2.3 Scene 3: Moving Location

The final stage of the scenario shows two further forms of runtime change: *Clive wants to show the price list to Bill, his boss, to discuss pricing. He ends the session with Sally, picks up the PDA, and walks to Bill's office. Clive and Bill look at the price list together.*

First, there is change in participant *location*, as Clive moves to Bill's office. Second, there is a change in *participants*, as Clive moves from collaborating with Sally to collaborating with Bill. Finally, there is a change in *collaboration style*, as Clive moves from distributed interaction with Sally to colocated interaction with Bill.

This scenario motivates the many challenges to developing adaptive groupware systems. We have seen that heterogeneity in participants' settings, roles and devices impacts their collaboration. We have addressed this heterogeneity by giving different user interfaces (with very different capabilities) to each of the participants. We have seen that the progression of the collaboration requires significant runtime change. This involves change in tools (as the quotation viewer was added) and change in device (as the quotation viewer was moved from PC to PDA.) Finally, change in participants and location was solved by migrating to a mobile device which was used in a colocated setting.

These changes impact not only the user interface, but also the distributed system on which the groupware tool runs. As Clive leaves the collaborative session with Sally, it is important that any shared data be made available on on his own computer and PDA once the session is complete. It is important that the migration of the quotation viewer interface be smooth, without any loss of data. Finally, it is important that partial failure of the distributed system be automatically resolved, leaving the application in a consistent state. Current groupware architectures and toolkits provide little support for these forms of adaptation, leaving developers to resolve them using low-level tools such as raw sockets and manual configuration of the distributed system.

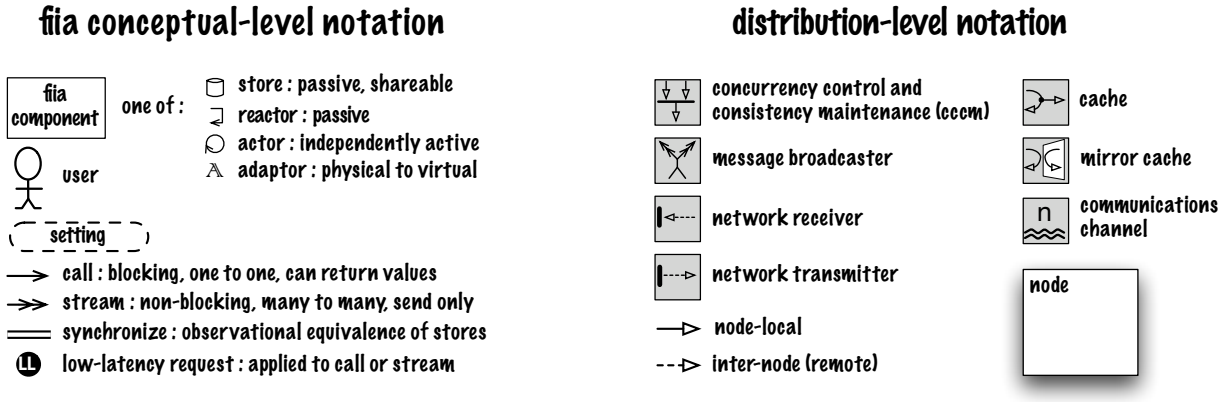

Figure 2: The *Fiia* notation. Developers use the conceptual notation. The *Fiia.Net* toolkit generates implementations using the distribution notation.

Following a discussion of related work, we introduce the *Fiia* notation and its associated toolkit. *Fiia* helps developers by allowing direct implementation of scenarios involving adaptation, such as those used to illustrate the furniture layout application presented here.

3. RELATED WORK

Several *conceptual* architectural styles have been proposed for reducing the complexity of groupware implementation by providing developers with a high-level view of their system. The Clover Model [20] and PAC* [5] describe the design of a groupware application based on layers of sharing, communication, and coordination. Some styles, like ALV [17] and Clock [15], are tied to toolkits, and so can be used to directly implement groupware. The use of conceptual architectural styles encourages iterative development, as the result of architectural changes can be tested without reimplementing the distributed system. Neither ALV nor Clock provides support for runtime adaptation.

Groupware tools reduce development complexity, but traditionally support only simple adaptations. For example, GroupKit [25] allows users to create, join, and leave collaborative sessions with shared applications. More complex adaptations, such as presented in our scenario, need to be implemented manually.

Traditional middleware provides surprisingly little support for building adaptive groupware applications. Many of these infrastructures do not support runtime adaptation at all [9, 27]. Those that do are often limited to load-balancing [8, 24], or plug-replacing specific policies [2]. There are a few middleware infrastructures that support general runtime configuration, but do so purely at the distribution level [7, 21]. These approaches require developers to work at a low-level, rather using a conceptual architectural style.

Newer techniques have been developed that help with dynamic adaptation of specific aspects of applications. *Plastic* user interfaces can adjust their appearance based on changes in the runtime environment [6], or information from user behavior [26]. Similarly, MATCH [22] evaluates and chooses a configuration suitable for users' interaction tasks. Chung and Dewan [10] describe a system which dynamically allocates replicated servers across the computers in a groupware system. Genie [3] adapts the implementation of a distributed system by generating configurations, and automatically migrating between then. One.world [16] supports the kind of adaptation described in our scenario, requiring programmers to work at the level of the distributed system, explicitly determining when to replicate and migrate data. Recent work in modeling performance of groupware applications promises to guide tradeoffs between different distribution configurations [18].

While these techniques help specify or implement limited adaptations, they provide little support for the drastic conceptual changes found in adaptive groupware scenarios. In the following sections, we will show how *Fiia* helps address this need.

4. MODELING ADAPTIVE GROUPWARE WITH FIIA

Fiia helps developers model adaptive groupware systems. Developers map scenarios (such as those from our furniture layout example) onto *Fiia* diagrams, and then use the *Fiia.Net* development toolkit to map those diagrams to running code. The key benefits of *Fiia* are its high-level support for adaptation and its easy mapping to executable code. In this section, we introduce the *Fiia* notation (figure 2) through examples drawn from the furniture layout scenario. In section 4.4, we show how these diagrams can be mapped into code within the *Fiia.Net* toolkit. *Fiia*'s core features are:

- Designs are structured around *settings*, each capturing a participant in context of his or her hardware, software, and ongoing collaborations.
- The contents of settings can be *synchronized*, allowing them to be shared by multiple participants.
- Settings can easily be modified at runtime. Changes can encompass application, collaboration structure, devices used and participant locations.
- Settings capture the components and devices with which participants interact, but abstract details of their implementation as a distributed system.

4.1 Scene 1: Fiia and Heterogeneous Settings

Fiia's core construct is the *setting*, a representation of a user together with his or her resources. Resources include computers, display and input devices, and applications currently in-use. Users can collaborate even when they have radically different settings. Figure 3 shows the settings for Sally and Clive in scene 1 of our furniture layout example. Clive can interact with the digital world via a *mouse and keyboard* and a *monitor*. Following the terminology of ASUR, these interaction devices are termed *adapters* [11].

Clive uses a *3D Layout Viewer* to display the contents of the furniture layout from a first-person perspective. This viewer takes

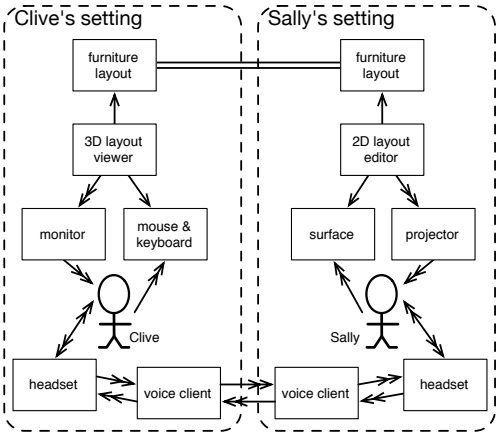

Figure 3: *Fiia* diagram for scene 1

its data from a *Furniture Layout* component, which describes the current set of furniture, and its position and orientation. Together, these describe Clive's setting, which comprises his interaction devices and the software components that he is using to carry out the collaboration.

Sally's setting differs from Clive's. She uses a tabletop *Surface* and *Display* to interact with the furniture layout. She uses a different client application, a *2D Layout Editor* that allows her to view and modify the layout from a top-down perspective. Unlike the mouse and keyboard that Clive uses, Sally uses gestures on the tabletop surface to place and orient furniture. Sally's client also obtains data from a *Furniture Layout* component.

Sally and Clive communicate via a voice over IP link. A voice client streams data between their *Headset* adapters.

From this example, we can see that Sally and Clive's settings differ in terms of their interaction devices and the client software they use to interact with the furniture layout application. The point of commonality between their settings is the *Furniture Layout* component. For Sally and Clive to interact, the *Furniture Layout* components in each of their settings are *synchronized*, as indicated by the parallel bars symbol (═). Synchronizing two components guarantees that they are *observationally equivalent*, that is, that they appear to their clients to behave in the same way. When the system is in a quiescent state, synchronized components provide the same responses to queries and produce equivalent event streams. Components may communicate via calls (⟶), and asynchronous data streaming (↠) connectors. Calls represent traditional method calls, while streaming is used to represent events (such as input events) and continuous data (such as voice over IP traffic.)

The *Fiia* notation presents users in their settings similarly to how scenarios describe participants' contexts. *Fiia* diagrams capture the interaction modalities the participant has at her disposal, and the software and data with which she is interacting. A single diagram represents a "scene", or a snapshot in time. As the collaboration changes over time, new *Fiia* diagrams are used to describe the resulting new scenes.

Fiia abstracts all issues of distributed implementation. Figure 3 does not specify how components are to be allocated to computational nodes, what networking protocols are to be used, or what caching or replication strategies are to be employed. The synchronization between the two *Furniture Layout* components specifies only that each setting has an instance of this component available, and that the instances are observationally equivalent. As we will see, the *Fiia.Net* toolkit is free to implement this using two replicated instances of *Furniture Layout*, a single instance shared by the two clients, or some hybrid scheme. This abstraction of distributed systems issues allows designers to concentrate on the structure of collaboration (at the level of scenarios) rather than becoming mired in low-level implementation issues.

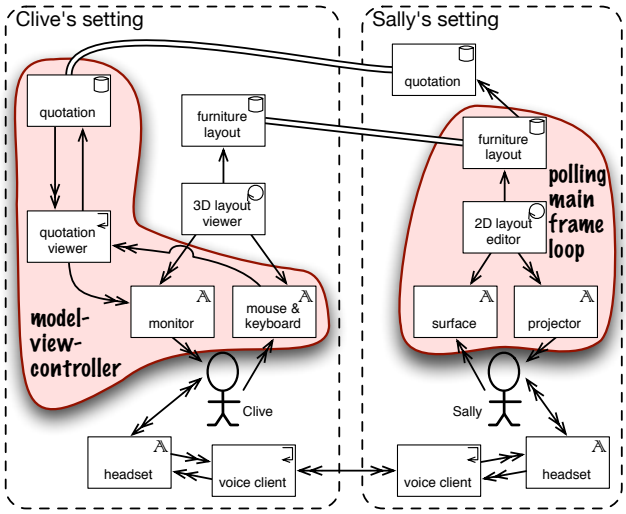

Figure 4: In scene 2, Clive's user interface is extended to include a quotation viewer

4.2 Scene 2: Fiia and Adaptation

Figure 4 shows the effect of adding a quotation viewer to the collaboration. A new *Quotation* component is created containing a price list for the furniture currently in the layout. In *Fiia*, this form of runtime adaptation is specified simply by adding components to the two participants' settings, attaching the components to adapters, and synchronizing the *Quotation* components.

Figure 4 also gives a sense of the expressiveness of the *Fiia* notation. The *2D Layout Editor* is implemented using a main-frame-loop architecture, polling the tabletop input device, and updating its *Furniture Layout* state. Meanwhile, the *Quotation Viewer* follows a traditional Model-View- Controller design, where the *Furniture Layout* reacts to events from the input device and the *Furniture Layout* store.

Fiia's powerful support for adapation is illustrated by Clive's move of the quotation viewer to his PDA (figure 5). In the *Fiia*

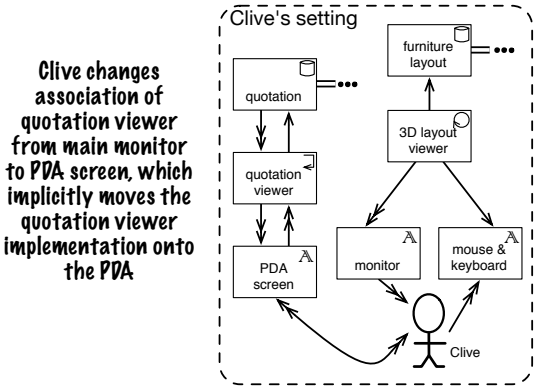

Figure 5: The quotation viewer is moved to the PDA

model, the *Quotation Viewer* component is attached to the PDA's adapter (screen and keypad). This automatically invokes a migration of the quotation viewer interface to the PDA. Issues of migrating components and their state between devices are left to the *Fiia.Net* toolkit to resolve. This simple change (accomplished by four lines of code in *Fiia.Net* – two disconnect and two connect calls) is all that is required to initiate the migration of the user interface to the PDA. All of the distributed systems issues of moving the component over the network and restarting it in the same state on the PDA are performed automatically by the toolkit.

4.3 Expressiveness of Fiia notation

Despite its simplicity, the *Fiia* notation can express most conceptual architectural styles for synchronous groupware. Figure 4 is annotated to show how *Fiia* conveys rich information about the architectural style. The *2D Layout Editor* is based on the standard main-frame-loop architectural style, where it repeatedly polls for input, updates its state (in *Furniture Layout*), and renders to the projector. The same main-frame-loop architecture is used to implement the *3D Layout Viewer*. Conversely, the quotation viewer is implemented using the traditional model-view-controller architecture. The *Quotation Viewer* component reacts to events delivered either from the mouse and keyboard or (via synchronization with Sally's setting) from the *Quotation* component.

As this scenario has shown, *Fiia* is a simple notation allowing architectures to be expressed at a conceptual level, close to the structure of the scenarios. *Fiia* diagrams can be written at different levels of detail, ranging through the simple view of figure 3 to the more detailed view of figure 4. Despite the simplicity of the notation, it can be used to encode a wide range of architectural styles typically used to develop groupware.

4.4 Mapping Fiia Diagrams to Code

As we have seen, *Fiia* diagrams follow naturally from scenarios, where each diagram corresponds to a scene. Similarly, it is straightforward to map diagrams to code; the concepts from *Fiia* diagrams map one-to-one to C# code using the *Fiia.Net* toolkit. The toolkit's API allows developers to create runtime models implementing *Fiia* diagrams. The toolkit them automatically implements these models as distributed systems. Programmers carry out adaptations at the level of the *Fiia* diagram, while *Fiia.Net* automatically maintains the distributed implementation.

As an example, the following code creates the *Furniture Layout* component in Sally's setting, and synchronizes it to the equivalent component in Clive's setting:

```
Fiia.Setting = "Sally";

Store furnitureStore =
   Fiia.NewStore<
      FurnitureLayout>();

SyncConnector sync =
   Fiia.SyncConnect(
      "shd-furniture", furnitureStore);
```

The *FurnitureLayout* class is a standard C# class, providing operations for manipulating the positions and orientations of furniture objects. The following code creates the the *2D Layout Editor* and adds the call connector between it and the *Furniture Layout* store:

```
Actor editor =
   Fiia.NewActor<TwoDLayoutEditor>();

Fiia.CallConnect(
   editor.Property("Layout"),
   furnitureStore.Type("IFurniture"));
```

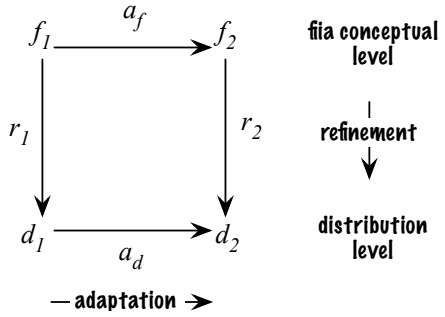

Figure 6: Fiia Framework

The *2D Layout Editor* is implemented by the C# *TwoDLayoutEditor* class. The call connection establishes the editor's Layout property as a reference in the *editor* component; this reference enables inter-component calls using standard C# syntax, such as

```
furnitureStore.AddFurniture(...);
```

We see from these examples that mapping from *Fiia* diagrams to code is straightforward, as each element of the *Fiia* diagram has a corresponding concept in the *Fiia.Net* API. The toolkit builds naturally on the features C# programmers are used to using, and has been integrated with Windows forms (for traditional graphical user interfaces) and XNA Studio (for 2D and 3D games and simulations.) *Fiia.Net* runs on a wide variety of platforms, including PCs and Windows Mobile PDAs.

5. FIIA PROGRAMS AS DISTRIBUTED SYSTEMS

As the previous section illustrated, *Fiia* diagrams abstract the details of distributed systems programming, allowing developers to concentrate on the functionality of their application rather than its implementation over a distributed network. Distribution issues in groupware applications are, however, challenging to address. Distributed groupware applications must provide excellent feedback and feedthrough times, must be secure, and must smoothly resolve partial failure of the underlying distributed system. Achieving these goals requires efficient handling of different distribution topologies, including centralized and replicated data distribution, consistency maintenance and concurrency control algorithms, data migration and caching. The *Fiia* approach helps with these problems by allowing developers to design their systems using the high-level *Fiia* notation, and then use the *Fiia.Net* toolkit to map the resulting code to an efficient distributed system.

Fiia.Net's mapping of the *Fiia* architecture to a distributed system is highly flexible. For example, two synchronized components might be mapped to a a single copy of the component accessed over the network by remote procedure call, or to two copies of the component whose replicated state is managed by any of a collection of consistency maintenance algorithms. As the *Fiia* conceptual architecture evolves at runtime, the distributed system is kept up to date. If parts of the distributed system fail, the *Fiia* architecture is automatically updated to reflect the new system state, allowing developers to react to failure without having to deal with the distributed system itself.

Developers may accept the distributed implementation that *Fiia.Net* provides by default, or may insert *annotations* into the architecture that guide the process of refining it to a distributed system.

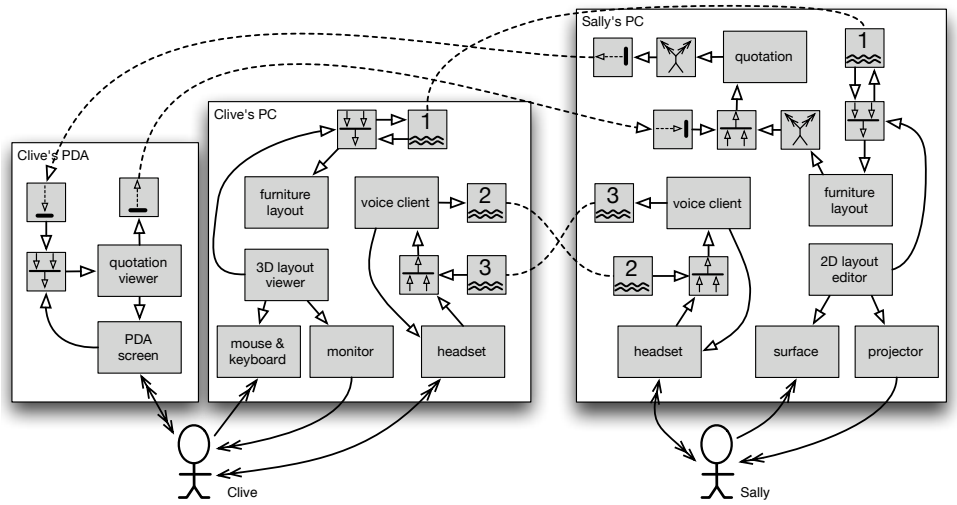

Figure 7: Distribution architecture for the *Fiia* diagram of figure 4

In this section, we show how *Fiia.Net* maps architectures to distributed systems. In the next section, we provide performance evaluation demonstrating the practicality of the *Fiia* approach.

5.1 Fiia.Net Framework

Figure 6 shows *Fiia.Net*'s conceptual organization. Programmers create a *Fiia* diagram (f_1) representing a scene. This diagram is automatically refined by *Fiia.Net* to a distribution architecture (d_1).

Runtime adaptations may occur at both the *Fiia* level (change application, device, location, etc.) or at the distribution level (network or component failure.) This leads to a new *Fiia* or distribution architecture (f_2 or d_2.) The toolkit then carries out necessary operations to reestablish consistency between the two levels. To our knowledge, *Fiia.Net* is unique in maintaining this two-level view of the system at runtime, and in automatically maintaining consistency between the two views. This allows developers to enact runtime change using high-level *Fiia* scenes. It also gives developers the means to deal with partial failure at a high level. Rather than dealing with repair of broken sockets, programmers view failure as the disappearance of components and connectors, making it easier to respond to a failure in its broader context.

5.2 Distribution Architecture

Figure 7 shows the distribution architecture for Scene 3 of the scenario. Distribution architectures are expressed in terms of infrastructure components, as enumerated in figure 2. These include implementations of the components specified in the *Fiia* design, as well as built-in components that handle issues such as caching, concurrency control and communication. In effect, these components make up a machine language for distribution architectures, to which the *Fiia.Net* refinery compiles *Fiia* designs.

The distribution architecture differs from the *Fiia* design in several ways, including:

- Components are allocated to computational nodes. For example, the *2D Layout Editor* is represented on Sally's PC, while the *3D Layout Viewer* is allocated to Clive's PC.

- It is determined how components communicate over the network. For example, the two instances of the *Furniture Layout* component use a multicast channel to communicate, while the *Quotation Viewer* communicates with a remote *Quotation* component via remote procedure call.

- Support for replica consistency is added in the form of special infrastructure components that provide a choice of concurrency control and consistency maintenance algorithms.

The *Fiia.Net* toolkit has a great deal of flexibility in how it chooses to produce a distribution architecture from a given *Fiia* design. Application programmers can choose to accept *Fiia.Net*'s choice of architecture, or, as we shall see, can influence these choices via high-level hints attached to the *Fiia* design itself.

We will now examine the specifics of this architecture in order to illustrate the range of issues that *Fiia* designs hide from the application programmer. We emphasize that figure 7 shows one of many possible distribution architectures for the *Fiia* design of figure 4.

The *2D Layout Editor* and the *3D Layout Viewer* both require access to data in the *Furniture Layout* component, which is replicated to both Sally and Clive's PCs. Each instance of the *Furniture Layout* can be updated by both the local and remote user interfaces (in response to Sally and Clive's inputs). Therefore, *Consistency Maintenance/Concurrency Control* (CCCM) components are required to ensure consistent execution of operations on the two replicas. CCCM's are shown visually as: . The CCCM components use an internal protocol based on message broadcasting to maintain the consistency of the replicas. The CCCMs communicate via a *channel* (), which provides multicast messaging.

The *Quotation Viewer* and *Furniture Layout* both require access to the *Quotation* component. In figure 7, the *Quotation Viewer* is represented on Sally's computer (a centralized implementation). The *Quotation Viewer* accesses the *Quotation* via remote procedure call. This is implemented via a pair of transmitter ()/receiver () infrastructure components.

In summary, the distribution architecture is a rich language allowing the expression of a wide range of distributed implementations. This example has shown some of this flexibility through the choice of centralized versus replicated data representation, remote procedure call versus multicast channel communication. Not shown here is the choice of implementations of the infrastructure components, which encapsulate, for example, choice of concurrency control algorithm or networking protocol.

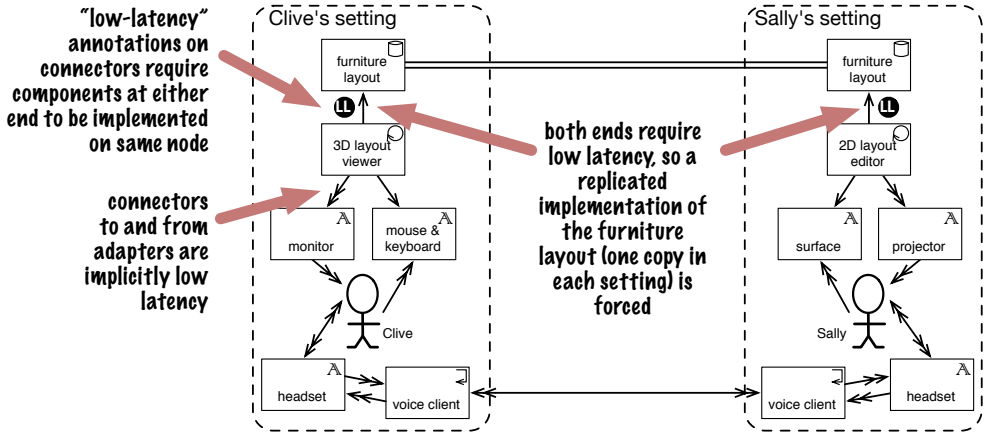

Figure 8: Annotated *Fiia* diagram for scene 1

5.3 Architectural Annotations

As discussed in the last section the *Fiia.Net* toolkit has considerable freedom in what distribution architecture is chosen for a given *Fiia* model. Developers can leave the choice of distribution architecture to the toolkit, or can use *annotations* to guide the toolkit's choices.

Annotations can be used to specify any of a wide range of implementation choices including: how components are anchored to nodes, whether to use replicated vs centralized data distribution, what channel topology to employ (multicast, centralized broadcast, multicast trees), caching strategy, CCCM algorithm, and networking protocol (lossless, lossy).

Annotations are attached to *Fiia* components. For example, in figure 8, the connector between the *3D Layout Viewer* and the *Furniture Layout* components is given a "low latency" (LL) annotation. This specifies that communication between the two components must be as fast as possible. When the refinery views this hint, it chooses (if possible) to anchor the two components on the same node. Similarly, Sally's *2D Layout Editor* is anchored to the same node as her *Furniture Layout* component. When combined, the effect of these two hints is to force the replication of the *Furniture Layout* component so that both the editor and viewer have immediate access to it. The consequences of this replication were shown in figure 7.

Annotations constrain the *Fiia.Net* runtime's choices. Whereever the runtime has discretion (e.g., on what node to locate a component, what concurrency control algorithm to use, or whether to replicate data), an annotation can be used to guide the runtime system in its choice. Annotations are an important part of the *Fiia* development process. If provided with no direction, the *Fiia.Net* toolkit is capable of deriving an implementation, but not necessarily a highly performant one. Annotations provide a very high-level way of guiding the refinement to distribution architecture. Annotations allow the developer to specify desirable performance qualities without having to specify the distributed systems details of how to achieve them.

5.4 Distribution Aspects of Adaptation

Simple runtime changes in a *Fiia* design may have significant effects at the distribution level. The real power of *Fiia*'s adapation is that developers are able to express changes easily, while *Fiia.Net* takes care of the potentially complex ramifications of these changes on the distribution-level.

As we saw in figure 5, migrating the *Quotation Viewer* to a PDA platform was easily specifed by rewiring the component's input/output connectors. This simple rewiring causes the following steps at the distribution level, roughly following the *Memento* design pattern [14]:

- The component's state is serialized
- The state is transmitted to the new platform, and deserialized, effectively relocating the component to the new platform
- The old component's connectors are redirected to the new component, and the old component is destroyed
- If any of these operations fail, the component is considered lost, and all connections to it are cleaned up.

Fiia's strength is that it allows access to such complex functionality through simple mechanisms.

Figure 9 reinforces this. In our scenario, Clive disconnects from Sally's session. All that is required to disconnect is to remove the synchronization between Clive's and Sally's *Quotation* stores. Both Clive and Sally are then free to continue to using the application in their own settings, but their actions are no longer communicated to the other. As was shown in figure 7, the synchronization was implemented using a single copy of the data on Sally's computer. For the disconnection to be implemented correctly, the data

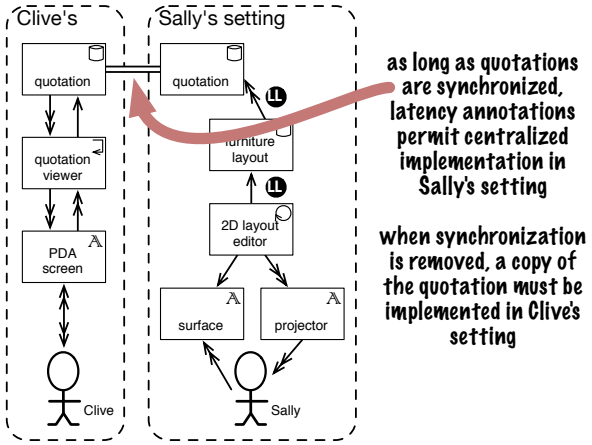

Figure 9: Effect of disconnecting from the session

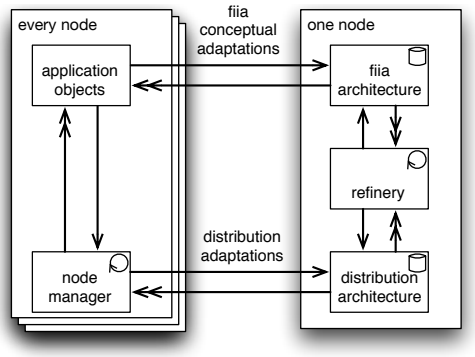

Figure 10: Implementation of the Fiia.Net runtime system

must be copied to Clive's computer, allowing him to retain access to it following the disconnection. Once again, this behaviour is easy to specify at the conceptual level (requiring a one-line command in *Fiia.Net* to remove the synchronization), saving the programmer from considerable complexity at the distributed systems level.

Figure 10 shows the runtime architecture of the *Fiia.Net* system. Every node has a *Node Manager* component responsible for configuring local objects as directed by the *Refinery*. One "master" node has the special status of being responsible for storing the conceptual and distribution architectures (the *Architecture* component), and manging the consistency between them. The *Architect* carries out conceptual-level changes to the architecture (e.g. create a new workspace and adding components), and notifies the *Refinery*. The *Refinery* is a rule-based system responsible for mapping the *Fiia* architecture to a distributed implementation, taking into account any specified annotations.

The heart of the *Fiia.Net* runtime is its rule-based refinery. The refinery treats maintenance of the conceptual and distribution models as a bi-directional graph rewriting problem. 34 graph rewrite rules, written in Story Diagram notation [12], embody the choice points in the implementation of conceptual architectures. At a given point during the refinement, some number of rules may be applicable. The choices of which rules are applied determines the ultimate distribution model. These choices include:

- How to allocate a component to a computational node
- How to implement a synchronization connector (centralized with proxies; replicated, or hybrid)
- How to implement event stream communication (peer-to-peer; routed via server, or group-casting)
- How to implement inter-node calls (RPC; firewall-friendly RPC via supernode; with or without cache).

At runtime, a *trace* data structure represents the choices that were made in refining the current conceptual model to the current distribution model. When changes occur at the distribution level (e.g., due to partial failure), this trace is unwound to the point that it correctly describes the current distribution (i.e., the failed components have been removed), and then re-refined to a valid distribution. When changes occur at the conceptual level, the conceptual architecture is re-refined, re-using the previous refinement where possible. This provides stability in refinement, so that small changes in the conceptual level do not lead to unnecessarily large changes at the distribution level.

6. EVALUATION

We have applied *Fiia.Net* to the development of a range of adaptive groupware applications. These include the tabletop-based furniture layout application described in this paper, the *Raptor* tool for collaborative video game sketching [28], a mobile, distributed presentation tool, a simple video conferencing system, and a framework for experimenting with concurrency control in games [13]. We have found that with appropriate training, designers are able to effectively specify scenarios using the *Fiia* notation, and to then map the scenarios to code. The *Fiia.Net* toolkit is usable within the context of our research group, and following further polishing and documentation will be made available to the broader community.

Fiia.Net allows developers to specify adaptation at a very high level. In order to illustrate that the approach is practical, we have measured the performance of the furniture layout application. In these experiments, *Fiia.Net* gives excellent runtime performance, and the time required to carry out runtime adaptations is insignificant.

We measured the runtime performance of one user manipulating furniture on the tabletop while another user views the furniture on a PC (scene 1 of the scenario.) Measurements were taken with 20 pieces of furniture in the layout. Three metrics were used to assess performance: *frame rate* (the number of frames per second that the 3D Layout Viewer is able to generate); *feedback time* (the time it takes a user to see the results of moving a piece of furniture on the 2D Layout Editor), and *feedthrough time* (the time it takes a furniture move performed with the 2D Layout Editor to appear in the 3D Layout Viewer.) We did not directly measure feedback time, but instead exploited the property of the frame loop architecture that feedback time can be estimated from frame rate; i.e., an input will be processed in the next iteration of the main frame loop, allowing feedback time to be bounded by $1000/frame\_rate$.

We measured two conditions: (1) over a 100 Mbps local area network, with the 2D Layout Editor on an Intel Core 2 at 2.4 GHz, 2 GB RAM and an NVIDIA 6600 GPU, and the 3D Layout Viewer on an AMD TL-50 1.6 GHz dual core processor, 2 GB RAM, with ATI X1300 GPU; and (2) over a low-bandwidth wide area network with the 3D Layout Viewer on an AMD TL-60 2.0 GHz dual core processor, 3 GB RAM, with NVIDIA 7000M GPU. We measured frame rate of the 3D Layout Viewer, feedthrough time from the 2D Layout Editor to the 3D Layout Viewer, and calculated feedback time on the 2D Layout Editor.

We used three versions of the application. As a control, we modified the application to replace *Fiia* with *.Net Remoting*, the remote procedure call facility built into Microsoft's .Net framework. We used two *Fiia* versions of the application, one where the Furniture Layout component is centralized on the tabletop PC, and one where it is replicated to both PCs.

	LOCAL AREA			WIDE AREA		
	Frame Rate *(fps)*	Feedbk Time *(ms)*	Feedthru Time *(ms)*	Frame Rate *(fps)*	Feedbk Time *(ms)*	Feedthru Time *(ms)*
.Net Remoting	57	21	18	7	17	136
Fiia Centralized	58	17	17	8	17	125
Fiia Replicated	58	17	18	52	17	25

Figure 11: *Fiia* runtime performance. Times rounded to nearest ms.

These results are summarized in figure 11. Not surprisingly, *Fiia*'s replicated case gives the best performance, as the 3D Layout Viewer has a local copy of all data necessary to execute its main frame loop. This allows it to run close to XNA Studio's maximum frame rate of 60 fps. This version also provides the best feedthrough times.

On the local area network, the performance of the .Net Remoting case is similar to the *Fiia* centralized case, indicating that *Fiia*'s runtime system has minimal overhead; both give performance that from the user's perspective is instantaneous. Over the wide area, their performance was virtually the same, and both unusably slow. This was because the network communication required to render the scene took in excess of 100 ms, allowing the frame loop to execute at only 7-8 fps.

In all cases, feedback time on the 2D Layout Editor was less than 20 ms, instantaneous from the point of view of a user.

These results show that *Fiia.Net*'s runtime performance compares well with Microsoft's integrated remote procedure calls, while granting much more flexibility. Tuning the system using annotations gave far superior performance to standard .Net Remoting.

Next, we measured the time it took *Fiia.Net* to carry out two adaptations drawn from scene 2 of the scenario. Over a local area network, and using the computers listed in condition (1) above, the average time to perform each adaptation five times was:

- *Creating the quotation viewer:* 897 ms
- *Moving the quotation viewer from PC to PDA:* 344 ms.

In both cases, the adaptations were carried out with a delay of less than one second. This shows that *Fiia.Net* performs adaptations quickly enough to easily support groupware systems of this size.

7. CONCLUSION

This paper has introduced the *Fiia* notation for modeling adaptive groupware, and its associated *Fiia.Net* toolkit. We have seen that *Fiia* allows developers to model their systems at a level similar to scenarios, providing snapshots of the system state at different points in its execution. *Fiia*'s conceptual, user-centered modeling allows developers to easily specify complex adaptations without being bogged down by the details of the distributed system. An annotation concept allows developers to specify, again at a high level, desired distribution properties such as choice of networking, caching or consistency maintenance strategy. Finally, we have reported that *Fiia.Net*'s performance is more than acceptable for the development of groupware applications.

There are considerable opportunities for future work. While it is straightforward to encode *Fiia* architectures using C# commands in *Fiia.Net*, we plan to experiment with the development of a graphical editor that will allow the diagrams to be executed directly. The toolkit opens up the possibility for experimentation with distribution algorithms, allowing us to easily compare the effectiveness of different strategies simply be modifying annotation. Finally, more work is required to assess the effectiveness of the toolkit with a broader user community.

8. ACKNOWLEDGMENTS

We gratefully acknowledge the funding of the Natural Science and Engineering Research Council of Canada and the NECTAR CSCW research network.

9. REFERENCES

[1] M. Ancona, G. Dodero, and V. Gianuzzi. RAMSES: A mobile computing system for field archaeology. In *Handheld and Ubiquitous Computing*, pages 222–233. Springer-Verlag, 1999.

[2] G.E. Anderson, T.C.N. Graham, and T.N. Wright. Dragonfly: Linking conceptual and implementation architectures of multiuser interactive systems. In *Proc. ICSE 2000*, pages 252–261, 2000.

[3] N. Bencomo, G. Blair, and P. Grace. Models, reflective mechanisms and family-based systems to support dynamic configuration. In *MODDM '06*, pages 1–6. ACM Press, 2006.

[4] E.A. Bretz. The car: Just a web browser with tires. *Spectrum*, 38(1):92–94, January 2001.

[5] G. Calvary, J. Coutaz, and L. Nigay. From single-user architectural design to PAC*: A generic software architecture model for CSCW. In *Proc. CHI '97*, pages 242–249. ACM Press, 1997.

[6] G. Calvary, J. Coutaz, and D. Thevenin. A unifying reference framework for the development of plastic user interfaces. In *Proc. EHCI '01*, pages 173–192, 2001.

[7] L. Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation, System Research Center, Palo Alto, CA, March 1994.

[8] R. Chandra, A. Gupta, and J.L. Hennessy. Data locality and load balancing in COOL. In *Proc. PPOPP*, pages 249–259, 1993.

[9] R.S. Chin and S.T. Chanson. Distributed object-based programming systems. *ACM Comput. Surv.*, 23(1):91–124, 1991.

[10] G. Chung and P. Dewan. Towards dynamic collaboration architectures. In *Proc. CSCW '04*, pages 1–10. ACM Press, 2004.

[11] E. Dubois, L. Nigay, and J. Troccaz. Consistency in augmented reality systems. In *Proc. EHCI '01*, LNCS 2254, pages 117–130. Springer-Verlag, 2001.

[12] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. In *Proc. TAGT'98*, pages 296–309. Springer-Verlag, 2000.

[13] R.D.S. Fletcher, T.C.N. Graham, and C. Wolfe. Plug-replaceable consistency maintenance for multiplayer games. In *Proc. NetGames*, pages 34–37, 2006.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[15] T.C.N. Graham and T. Urnes. Linguistic support for the evolutionary design of software architectures. In *Proc. ICSE 18*, pages 418–427. IEEE Computer Society Press, 1996.

[16] R. Grimm. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[17] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, and W. Wilner. The *Rendezvous* language and architecture for constructing multi-user applications. *ACM TOCHI*, 1(2):81–125, June 1994.

[18] S. Junuzovic, G. Chung, and P. Dewan. Formally analyzing two-user centralized and replicated architectures. In *Proc. ECSCW '05*, pages 83–102. Springer-Verlag, 2005.

[19] E.T. Khoo, S.P. Lee, A.D. Cheok, S. Kodagoda, Y. Zhou, and

G.S. Toh. Age Invaders: Social and physical inter-generational family entertainment. In *Proc. CHI '06*, pages 243–247. ACM Press, 2006.

[20] Y. Laurillau and L. Nigay. Clover architecture for groupware. In *Proc. CSCW '02*, pages 236–245. ACM Press, 2002.

[21] R. Litiu and A. Prakash. Developing adaptive groupware applications using a mobile component framework. In *Proc. CSCW 2000*, pages 107–116. ACM Press, 2000.

[22] T. McBryan and P.D. Gray. A model-based approach to supporting configuration in ubiquitous systems. In *Proc. DSV-IS '08*, pages 167–180, 2008.

[23] D. Pinelle and C. Gutwin. Loose coupling and healthcare organizations: adoption issues for groupware deployments. *Computer Supported Cooperative Work*, 15(5–6):537–572, 2006.

[24] C.G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. SPAA '97*, pages 311–320. ACM Press, 1997.

[25] M. Roseman and S. Greenberg. Building real time groupware with GroupKit, a groupware toolkit. *TOCHI*, 3(1):66–106, March 1996.

[26] J.-S. Sottet, V. Ganneau, G. Calvary, J. Coutaz, A. Demeure, J.-M. Favre, and R. Demumieux. Model-driven adaptation for plastic user interfaces. In *Proc. INTERACT '07*, pages 397–410, 2007.

[27] C. Wolfe. Conceptual programming models of distributed systems. Technical Report 2006-525, School of Computing, Queen's University, 2006.

[28] C. Wolfe, J.D. Smith, T.C.N. Graham, and W.G. Phillips. A model-based approach to engineering collaborative augmented reality. In E. Dubois, P. Gray, and L. Nigay, editors, *Engineering of Mixed Reality*. Springer Verlag, 2009.

[29] J. Wu and T.C.N. Graham. The Software Design Board: A tool supporting workstyle transitions in collaborative software design. In *Proc. EHCI/DSVIS '04*, pages 363–382. LNCS, 2004.