The Workspace Model:
Dynamic Distribution of Interactive Systems

by

WILLIAM GREG PHILLIPS

A thesis submitted to the School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

July, 2006

Abstract

This thesis presents an architectural model for synchronous groupware called the Workspace Model, which provides a clean separation of conceptual structure from distributed implementation. The model includes a formally-defined, distribution transparent, conceptual level architectural model with appropriate abstractions for the development of groupware; a formally-defined implementation level architectural model that exposes the distributed system issues abstracted at the conceptual level; a formal relation between the two levels that allows a range of implementations to be automatically computed for any conceptual level architecture; and an explicit representation of runtime change.

We argue that this combination of properties allows the model to satisfy needs arising from three communities: end-users, application programmers, and toolkit developers. End users require that their groupware systems support fluid collaboration, perform efficiently, and behave in a predictable manner. Application programmers need an environment that is appropriately expressive and that doesn't require them to commit prematurely to a distributed implementation. Toolkit implementors need a model that is formally represented, that provides a range of distributed implementations, that supports efficient incremental refinement of conceptual architectures to distributed implementations.

The Model has been designed such that all syntactically correct conceptual level archictures can be refined to fully refined architectures by a finite computation. A proof sketch of this key property is offered.

The Workspace Model has been implemented in a toolkit and runtime system called fiia, which demonstrates that the model is both practically implementable and usable for the creation of groupware programs. The fiia programming model and the implementation of fiia itself are presented.

Acknowledgements

I have heard it said that a doctoral dissertation is the ultimate in self-indulgent solo research. If so, then I feel tremendously cheated, because this work has been very much a part of a rich and ongoing collaborative enterprise.

My most sincere appreciation goes to my doctoral supervisor, Nick Graham, leader of the EQUIS lab at Queen's, whose initial germ of an idea eventually grew into the work reported in this document. Nick was always there when I needed him, offering useful advice, criticism, cajolery, and steadfast encouragement, each at exactly the right times.

Much that is good about this work grew out of early discussions with other members of the EQUIS lab, including Tore Urnes, Gary Anderson, and Crazy Tim Wright. Later interactions with David Smith, Rob Fletcher, Banani Roy and especially Chris Wolfe helped me regain my enthusiasm, just when I needed it for the final push.

Members of the RMC Symbiotic Computing Lab provided feedback on various earlier versions of this work. This includes my colleagues Bob St. John, Lobna Chérif, Sandy Berg, Jeff Paul, and Mike LeSauvage, as well as my students Grant Griswold, Bruce Conlin, Len Terpstra, Michel Hutchson, Ian Krepps, Scott McLean and David Leblanc.

Present and former members of IFIP Working Group 2.7/13.4 provided both encouragement and sharply pointed questions. While they have all been helpful in one way or another, I would particularly like to thank Len Bass, Morten Borup Harning, Gilbert Cockton, Joëlle Coutaz, Phil Gray, Bonnie John, Rick Kazman, Laurence Nigay, Fabio Paternò, Philippe Palanque, Kevin Schneider and Leon Watts.

This work has been completed entirely on a part-time basis. I'd like to go on record as saying that this is *not* the best way to earn a doctorate. However, if you have to do it, you'll absolutely need supportive bosses in your day job. Fortunately I have had them, in the persons of Aziz Chikhani, Doug Dempster, Yui-Tong Chan, Bernard Mongeau, Jacques Hamel, Côme Rozon, and Derrick Bouchard.

Scott Knight provided the existence proof that a part-time doctorate is possible;

thanks for holding up the light at the end of the tunnel. And thanks also to my co-part-timers Alain Beaulieu, Sylvain Leblanc and Ron Smith for providing just the right balance of support and derision. You'll get there too guys, I know you will.

My examining committee of Saul Greenberg, Tom Dean, Jim Cordy and Pat Martin provided many excellent comments; the final version of this dissertation much the better for their suggestions.

Finally, I'd like to thank my wife Karen and children Tatiana, Shona and Tristan for always being there to remind me of why I was doing this, and for putting up with me so well, even at my most distracted. My kids have never really known a father who wasn't working on his doctorate... I wonder what we'll do next!

Statement of Originality

I, William Greg Phillips, certify that all results in this thesis are original, unless otherwise noted. Specifically, those results due to other authors which have appeared in the literature have been cited as necessary.

Earlier versions of some parts of the work reported in this thesis have previously appeared as [101, 102, 103, 104].

Table of Contents

Table of Contents — Continued

Table of Contents — Continued

Table of Contents — Continued

Table of Contents — Continued

List of Tables

List of Figures

List of Figures — Continued

List of Figures — Continued

Chapter 1

Introduction

This thesis presents an architectural model for synchronous groupware that provides a clean separation of conceptual structure from distributed implementation. Called the Workspace Model, it includes an evolution calculus that allows the formal description of architectural change at runtime, whether this be user-driven conceptual change (such as adding a new user to a collaboration) or system-driven implementation change (such as the failure of a network link). The model defines a formal relation between the conceptual and the implementation level, allowing automatic generation and incremental computation of the set of implementation architectures corresponding to any given conceptual architecture.

Groupware is challenging to implement in part because it is difficult to design and program fluid usable collaboration support while simultaneously coping with low-level distributed system issues such as the maintenance of shared state and response to partial failure [61]. The aim of our model is to address this issue by separating the essential component and interaction semantics of a groupware system from the system's distributed implementation at runtime. This separation allows functionality and design time quality attributes like clarity, maintainability, and reusability to be considered separately from runtime quality attributes like performance, availability, and security.

We have designed and formally specified the Workspace Model, and have implemented it within our novel fiia groupware development toolkit. Our preliminary experience with the model suggests that its separation of concerns makes it a useful software engineering approach when considered from the perspectives of end users, application programmers, and groupware toolkit developers.

This research sits at the intersection of the studies of synchronous groupware and of software architecture. In the remainder of this chapter we present brief introductions to these two fields. We also define the problem that our work is intended

to address, and identify key attributes of an effective solution. This is followed by brief overview of the Workspace Model and a list of its core contributions. We close with a road map to the balance of this document.

1.1 Synchronous groupware

As predicted by J.C.R. Licklider in the 1960s [80], the emergence of high-bandwidth networks and inexpensive computers has lead to the widespread use of computers for communications and collaborative tasks. This has required the study of how to engineer interactive software that supports multiple users. Such software is often called *groupware* [6], or *synchronous groupware* in the special case of software supporting realtime collaboration. In the balance of this paper, "groupware" is used to mean synchronous groupware.

There are many kinds of groupware applications, each with different development and ergonomic requirements. Some support pure communications tasks; for example, Internet telephones [73, 7] and chat programs [137] allow people to converse using voice or text, while media spaces [33] contribute to mutual awareness in work groups by allowing people to see views of others' offices. Tools like WebEx [142], GoToMeeting [53], and WebArrow [141] provide support for distributed meetings of small to medium sized groups, including voice communication, screen and application sharing, file exchange, and other services.

Other applications support the collaborative production of work artifacts, such as school assignments [89], bank loan applications [72], software designs [65] or geophysical simulations [45]. These applications typically support both discussion and the realtime manipulation of shared artifacts.

Away from a work context, games that allow people to collaborate and compete across computer networks have also become enormously popular. Services such as GameSpy [51] and Xbox Live [146] allow millions of people to connect and play together. Massively multiplayer games such as World of Warcraft [144] allow thousands of people to meet, socialize and adventure together in the same persistent virtual world.

1.1.1 Vision scenario

To make our discussion more concrete, we outline the following scenario, which represents a vision of the kind of groupware we would like to provide. We follow the scenario with a brief analysis of why this kind of groupware is so difficult to build.[1]

Our scenario begins with Dave at his workstation computer in his office in Canada, fleshing out an initial design for a new software system using a computer aided software engineering (CASE) tool. Ian, who is a member of the same project team, walks into Dave's office to see what progress Dave has made. Since Dave's office is somewhat cramped, Dave opens a new CASE tool interface on a large, touch-screen display across the hall; this interface is different from the one on his workstation and is more appropriate for use with a touch-screen. Dave and Ian move across the hall, bringing along the tablet computer that Ian had in his bag.

The design Dave and Ian are working on is fairly complex. Despite the touch-screen's large size it has a relatively low resolution and doesn't show all the detail that they would like to see. Part of the screen is occupied by an object palette provided by the CASE tool's interface; Ian lends Dave his tablet computer and they free up some screen space by moving the palette onto the tablet. They use the tablet and the touch-screen together to modify and extend their design as their discussion progresses.

After an hour's work, Dave and Ian decide to get opinions about their initial approach from their colleagues Karen and Tatiana. Karen is visiting a customer site in Japan and her laptop has only a high-latency, medium-bandwidth connection to the corporate network; Tatiana is in the same building as Dave and Ian and her workstation has a low-latency, high-bandwidth connection.

Dave sends Karen and Tatiana invitations to join him and Ian remotely, and both of them accept. Their copies of the CASE tool start up, displaying Dave and Ian's design; at the same time a voice channel is opened between the four of them. The team browses through the various diagrams in the design together, each proposing

[1] We return to this scenario in chapters 3 and 6 to illustrate the Workspace Model.

modifications directly on the design as they proceed, sometimes in parallel. At one point Karen needs to check something on an overview diagram while Dave, Ian and Tatiana are discussing a more detailed one, so she temporarily decouples her view from the group's. After finding the information she needs, she rejoins the group view and makes a few quick changes to the detailed diagram; the group discusses her suggestions and decides to incorporate them. When the four of them are satisfied, Ian, Tatiana and Karen depart to work on other tasks; Dave returns to his office and continues elaborating the design.

1.1.2 Groupware is hard to build well

There are almost no existing systems that support the fluidly dynamic collaboration depicted in this scenario. Assuming that the large screen is driven by its own computer, moving the CASE tool from Dave's computer to the large screen would require saving the current design, closing the design on Dave's computer to prevent conflicting changes, and reopening the design in a new instance of the CASE tool on the large display. Moving the tool palette from the large display to Ian's tablet would simply not be possible. Initiating the remote collaborative session with Karen and Tatiana would require setting up a "conference" using an application-sharing tool like WebArrow [141], sending out email invitations, and then collaborating using the services provided by the WebArrow tool. And, since WebArrow, like most other application sharing systems, supports only sharing whole applications or screens, Karen's quick check of another diagram without interrupting her collaborator's work would not be possible. The overhead of collaborative software use is clearly excessive: small wonder that groupware has enjoyed limited success outside niche applications like instant messaging, meeting support, and multiplayer gaming.

But the fluid collaboration in our scenario is exactly the kind of collaboration we would expect if our four participants were all in the same room, using physical tools like pens, paper and whiteboards! Why is it so hard to create software systems that support dynamic, natural collaboration?

Greenberg argues persuasively that the problem is a lack of appropriate high-level abstractions and tool support, which forces programmers to deal with a host of issues not immediately related to the semantics of their applications [61]:

> Yet when we look at groupware, programming tools are back in the dark ages. Groupware development in non-research settings requires a highly-trained programmer adept at writing low-level code. The programmer's task often includes implementing a network protocol atop of sockets, dealing with multimedia capture and marshalling (e.g., audio and video), writing various compression/decompression modules for information transmission, worrying about distributed systems issues such as concurrency control, developing a session management protocol so participants can create, join and leave conferences, and creating some kind of persistent data store so that information is retained between sessions. This list goes on and on.

To take one concrete example, consider the point in the scenario where Karen and Tatiana join Dave and Ian's collaboration. Clearly this requires the instantiation of some form of distributed system—but what should that distributed system be? Tatiana is on the local area network, so it might be reasonable to have her remotely access the same design representation that Dave and Ian are using. But Karen, in Japan and connected by a high-latency network link, might be better served by having a copy of the design transferred to her computer so that she can interact with it locally. Of course, this requires concurrency control and replica consistency maintenance to keep Karen's copy in sync with the other: how should this be effected?

Using current technologies, the developer of our hypothetical CASE tool would have to deal with all of these issues, in addition to the core issues of how the case tool itself should behave and how best to support its effective collaborative use. This includes the prediction, at design time, of what distributed system implementation would be most appropriate at runtime.

The result is that development effort that should be invested in improving the users' collaborative experience ends up wasted instead on low-level issues—and we end up with inflexible, hard to use groupware that doesn't support the kinds of collaborations illustrated in our scenario. The situation with groupware is analogous to the early development of graphical user interfaces (GUIs) [61]. Initially, GUIs

were simply too hard to program and not economical for most applications. With the arrival of well-designed window systems and GUI toolkits, programmers suddenly had a repertoire of appropriate abstractions to call on; now systems without GUIs are the rare exception.

Completely addressing the issues raised by Greenberg constitutes a research and development program for a lifetime. In this work we restrict ourself to those issues relating to the dynamic distributed implementation of groupware systems.

1.2 Software architecture

Our approach to dynamic distributed implementation is based on a model of the *software architecture* of groupware systems. Software architecture is about the elements in a system and their relation to one another, and is normally expressed in terms of the system's computational *components* and the *connections* between them [99, 118]. It is not concerned with details of data representation or with algorithms; while these are essential to the construction of real software they are typically specified in other representations. An architectural representation allows temporary abstraction of such details so that larger structural issues may be considered independent of lower level concerns.

Software architecture contributes throughout the development life cycle of software systems [8, 20, 118]. Architectures for synchronous groupware particularly help in the challenges of programming distributed systems with exigent performance and usability requirements.

An adequate description of a system's software architecture typically requires multiple *architectural views*, each of which is concerned with different aspects of the system [29]. For example, Kruchten's "4+1 architectural blueprint" includes four key views, plus scenarios to tie them together [76]. The views that he suggests are a logical view, representing key problem domain concepts; a process view, which documents concurrency and interprocess communication; a development view, representing the organization of software modules in the development environment; and a physical view, which maps deployed runtime components onto physical pro-

cessors connected by network links. Kruchten's taxonomy has been adopted and extended in the standardization of the Unified Modeling Language [16].

Where an architecture is presented from multiple viewpoints each view is intended to be complementary, in the same way that a building design may include complementary architectural representations for structural support, electrical and communications wiring, plumbing, and heating and ventilation. As with a building, a full understanding of the software system requires an understanding of all relevant views; however, each view supports different activities and is useful in and of itself.

1.2.1 Applying architecture to groupware

The concerns of interest in the development of groupware are much the same as the concerns for other interactive software, with two notable additions. First, the key issue that distinguishes groupware from other software is the requirement to represent information that is shared between users [97]. And second, because groupware is inherently multi-user, it is almost always implemented as a distributed system.

In a previously-published review of software architectures for synchronous groupware [101], which is summarized and updated in chapter 2, we noted that the literature on architecture for synchronous groupware addresses two distinct sets of architectural concerns. Using Kruchten's taxonomy, these two sets of concerns can be characterized as logical/development and process/physical. We call the logical/development view the *conceptual* view and the process/physical view the *implementation* view.

The conceptual view is concerned with the groupware system as a software artifact; that is, the system as ultimately represented in its source code. It may also represent the intended topologies of components at run time, from a logical perspective. In the conceptual view, the components tend to be associated with units of software development like classes or modules and the connectors tend to represent the use of one component by another in terms of source code or logical composition. Examples of this view include Dewan's "zipper" generalization [37] of the Arch reference model [130], the Abstraction-Link-View architecture of the Rendezvous

toolkit [66], the Presentation-Abstraction-Control* (PAC*) architecture [22], the implicit architecture of the GroupKit toolkit [112], the Clock architectural style [58] and the Clover architecture [77].

The concerns addressed in the conceptual view are largely related to the development and maintenance of the software, as opposed to its use. The aim of conceptual architectures is to propose an appropriate partitioning of the software system to allow for efficient development, easy modification, and effective code or component reuse. In particular, groupware architectures typically provide mechanisms that minimize the developer's need to deal explicitly with the distribution of and concurrent access to shared state [62].

By contrast, the implementation view is concerned with groupware as a distributed system at runtime. In this view, components are instantiated objects either associated with or realized as independent processes, and communicating via either direct reference or inter-process communications mechanisms. Processes are located on particular computational nodes (such as clients and servers) and communicate across network links. Shared state may be represented by a single component remotely accessed by all participants, or by multiple replicated copies, the consistency of which is maintained by a replica consistency maintenance algorithm [97]. All real groupware systems have an implementation architecture, which may be more or less well documented.

The implementation view allows analysis of such concerns as performance, fault-tolerance, and security, all of which are strongly influenced by runtime distributed system topology and the inherent characteristics of the runtime environment.

1.3 Research motivation

While our review of existing groupware architectures found a broad range of approaches at both the conceptual and implementation levels, we did identify two significant lacks. These stem from the fluid dynamic behaviours illustrated in our scenario above, and from the need to understand and reason about the correspondence between the conceptual and implementation views of a groupware system.

As illustrated in our scenario, human collaboration is inherently dynamic [40, 145]. People move between individual and collaborative work frequently and fluidly, and are often involved in multiple simultaneous collaborations. Collaborations may be brief or long running. The participants in any given collaboration may change over time. A single tool may be used to support many collaborations; a single collaboration may make use of many tools.

Further, if a collaboration is to make use of groupware, that groupware will necessarily be implemented on real computer systems connected by real networks. The computers may be of varying capabilities, ranging from high performance servers to small hand held devices like cell phones. Computers may join and leave a collaboration as their users do, or for other reasons both planned and unplanned. Network links will be of varying quality, and the quality of links will vary over time.

All this implies a need to reason about the dynamic properties of groupware systems, at the level of conceptual system structure as well as at the level of distributed implementation.

The time-varying nature of both the conceptual and implementation levels also suggests that groupware systems should be able to adapt, at runtime, to their changing environments [62]. If the conceptual view is expressed in a distribution-independent fashion, it is generally possible to realize a given conceptual architecture using any of a number of implementation architectures [101]. However, to do this we need to be able to identify precisely which implementation architectures represent correct realizations.

In practice we find that current architectural approaches to groupware lack this capability. Some conceptual architectures offer no direct support for identifying corresponding implementation architectures, leaving the translation between views entirely up to the designer [22]. Others, particularly those implemented in toolkits, offer only a limited range of implementations [66, 112]. Those that do offer a greater range of adaptability either lack the capacity to change their implementation at runtime [4, 132] or lack any mechanism for determining which implementations are semantically correct [81].

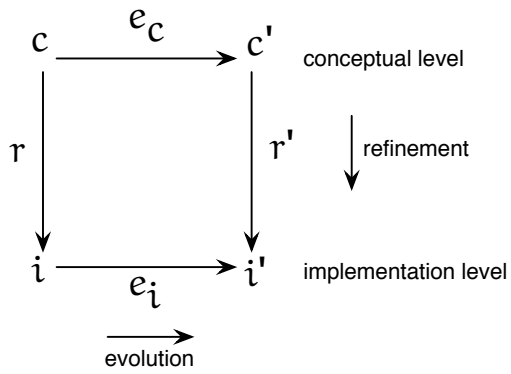


Figure 1.1: Key elements of the Workspace Model.

1.4 The Workspace Model

Our approach to addressing these concerns is the Workspace Model. In this section we first provide an overview of the model itself, then illustrate its use with a brief example. In chapter 3 we introduce the model in more detail using the CASE tool scenario presented above, and we return to the CASE tool scenario in chapter 6 to present Workspace Model approaches to scenario-based modelling and application design. The formal definition of the model is in chapters 4 and 5 and an evaluation of its properties is in chapter 8.

1.4.1 Overview

Figure 1.1 shows the key elements of the Workspace Model and how they relate to one another. These elements are the *conceptual level*, the *implementation level*, *evolution* at either level, and the *refinement relation* between levels.

In the Workspace Model, the architecture of a system is expressible at both the conceptual and the implementation levels. A conceptual architecture describes the structure of the elements making up an interactive system, without constraining how they are to be implemented as a distributed system. Conceptual architectures are illustrated by example in section 3.2 and formally defined in sections 4.1 and 4.2.

Implementation architectures are at a lower level of abstraction than conceptual architectures, exposing such details as the instantiation of components on nodes,

mechanisms for inter-process communication, and the implementation of concurrency control and replica consistency maintenance. Implementation architectures are presented in section 3.4 and formally defined in section 5.1.

The two levels are linked by a refinement relation R consisting of the reflexive transitive closure of a set of individual refinement rules. In general, R maps a given conceptual architecture to many implementation architectures. R therefore captures a space of possible implementations and $R(c, i)$ indicates that implementation architecture i is a valid refinement of conceptual architecture c. In figure 1.1, r is a particular refinement of c where $R(c, r(c))$ — that is, r refines c to some valid implementation $r(c) = i$. Section 3.3 provides an overview of refinement, which is formally defined in section 5.4.

Finally, evolution (e) expresses architectural change over time at the conceptual and implementation levels. In figure 1.1, $e_c(c)$, which is some series of conceptual evolution operations applied to conceptual architecture c, produces c', a modified conceptual architecture. Similarly, $e_i(i)$, a series of implementation level evolutions, produces i', a modified implementation level architecture. Given an initial c and i such that $R(c, i)$, and some conceptual level evolution e_c, it is the role of the runtime system to choose an implementation level evolution e_i such that $(\exists r')[r'(c') = i' \wedge R(c', i')]$. A similar condition, discussed later, applies to evolutions beginning from the implementation level. Evolution is discussed in section 3.5. The set of atomic evolutions make up the *evolution calculus*; the conceptual and implementation level evolution calculi are defined in sections 4.3 and 5.3, respectively.

1.4.2 A Workspace example

The actual use of the Workspace Model and its supporting notation is best illustrated with a simple example.[2] The notation used in this section's figures is summarized in appendix A and explained in more detail in chapters 3, 4 and 5; here we explain just enough to make the example understandable. The particular example used here is

[2] In fact, we contend that this is the simplest interesting shared-state groupware application, and as such is a groupware equivalent of the canonical "Hello, world!" program.

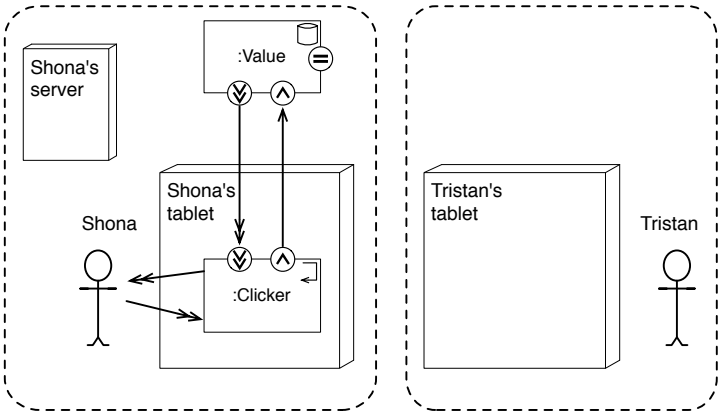


Figure 1.2: Shona arrives at the auditorium and sets up.

discussed in more detail in the contexts of application development and reasoning about fault tolerance in chapter 6.

In our example, Shona and Tristan are ushers at an auditorium. Their work involves taking tickets and maintaining a count of the people who enter. They are each provided with a "clicker"— a device that increments the count by one whenever they click its button. The total count of patrons entering the auditorium through either door is maintained in realtime and is visible on both clickers.

As head usher, Shona arrives first and begins to set up. Her clicker's user interface is implemented as software running on a small tablet computer. In addition to the tablet, she also has access to a server, running in the auditorium's administrative offices. Her clicker maintains its count of patrons in a separate software component called a "value"; Shona is not particularly concerned with where the value is physically implemented, as long as it functions correctly. This initial situation is illustrated by the conceptual level diagram in figure 1.2.

In the figure, the dotted lines represent the boundaries of Shona and Tristan's *workspaces*, which contain the physical objects, computers, and software of interest to each of them. In Shona's workspace we see Shona herself, along with two *nodes* representing her tablet and her server. Her clicker's user interface is shown as a *component* of type Clicker which is *anchored* to her tablet, which means that it must actually be instantiated there. The Clicker is connected to a Value component,

used to store the current patron count, which is not anchored to any node. This means that the runtime system is free to instantiate the Value on any appropriate node. The symbol in the upper right corner of the Clicker component shows that it is a *reactor*; that is, a component that is passive except in response to externally-provided inputs. The Value component is a *store*; like a reactor it is passive, however it has the additional feature that it may be shared between multiple users.

The double-headed arrows between Shona and her Clicker are called *subscription connectors* and represent asynchronous streams of events. Here, they indicate that Shona is able to provide input to her Clicker interface and can observe any changes in its state. The double-headed arrow from the Value to the Clicker similarly indicates that the Value can send asynchronous events to the Clicker. The single-headed arrow, called a *call connector*, allows Shona's Clicker to synchronously modify and query the state of the Value. The intent is that these connectors will be used in a classic model-view-controller communication pattern [74], with the Clicker modifying the Value as required, and the Value notifying the Clicker any time its state has changed.

The circles on the boundaries of the two components are *ports*, which represent potential attachment points for connectors. The symbols in the ports correspond to the kinds of connectors that may be attached, and (for call and subscription ports), the directions of the arrows (into or out of the components) indicate whether they are *source* or *target* ports. The unconnected port on Shona's Clicker is a *synchronization* port, whose use is illustrated below.[3]

When Tristan arrives at the auditorium he must set up his clicker interface and associate its count with Shona's count. This is a dynamic *evolution* of the system at the conceptual level. The Workspace Model provides a language, called the *evolution calculus*, for representing such dynamic change. The conceptual-level evolution calculus is defined in section 3.5. For the current instant we assume that appropriate evolutions have been requested and that the end result is as shown in figure 1.3.

In the figure we see that Tristan has both Clicker and Value components and that

[3] In informal diagrams we often leave ports out to reduce visual clutter; the existence of a port may be inferred by the attachment of a connector to a component.

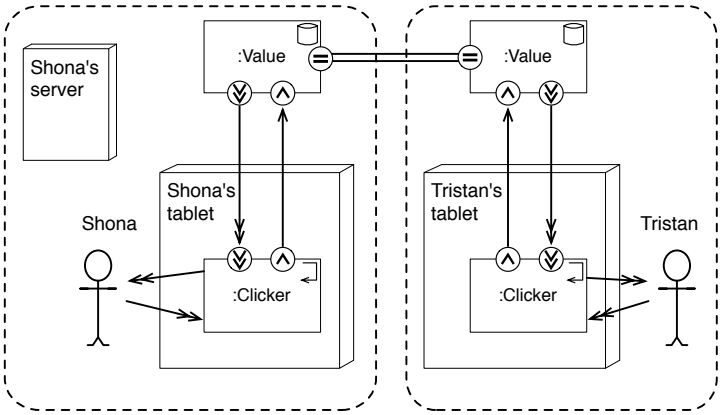


Figure 1.3: Tristan joins Shona at the auditorium.

---

these have been configured similarly to Shona's. Additionally, Tristan and Shona's Value components have been connected together by means of the double-line *synchronization connector*. The precise semantics of synchronization are defined in chapter 4; the intuition is that two synchronized components represent "the same thing", so Shona's Value and Tristan's Value are the same Value. This means, for example, that if Shona presses the button on her Clicker and this causes a change in the count maintained in her Value, Tristan's Value's count is identically updated. Also, when Shona's Value sends a message to her Clicker indicating that its state has changed, Tristan's Value will do likewise.

However, it is important to note that synchronization says nothing about how this "sameness" is to be implemented: at the implementation level there could be a single copy of the Value implemented on any available node, or there could be multiple replicas of the Value, kept consistent with one another by any of a number of replica consistency maintenance algorithms.

The Workspace Model's refinement relation defines the space of allowable implementations for any given conceptual level architecture. Refinement is expressed in terms of a set of graph transformation rules, each of which has a precondition and a postcondition. Where a rule's precondition matches in the current architecture the rule may be applied, generating a new, partially refined architecture which conforms to the postcondition. The transitive closure of non-deterministic rule ap-

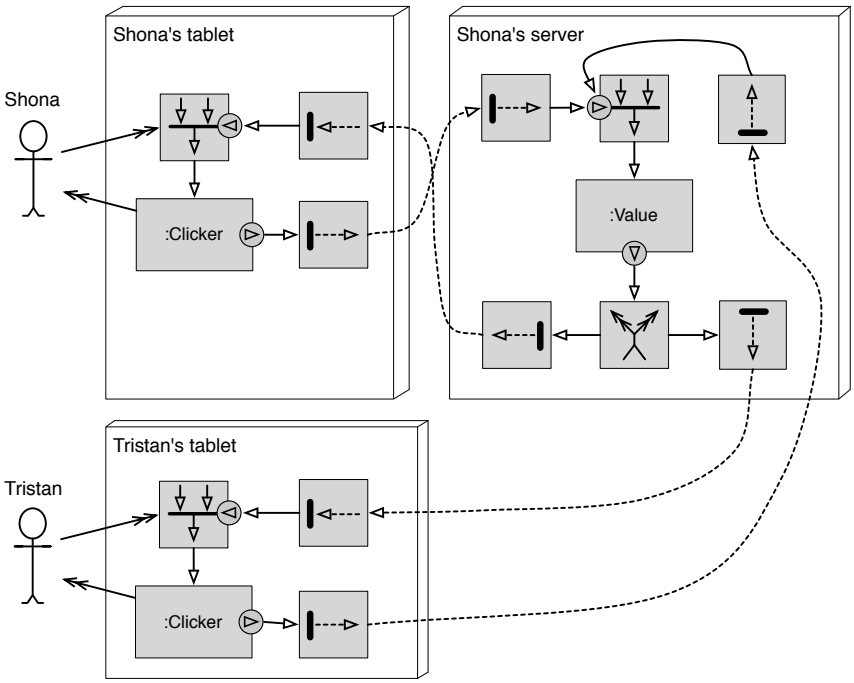


Figure 1.4: A server-based, centralized implementation of the architecture shown in figure 1.3.

plication on a given conceptual architecture generates all legal corresponding implementation architectures. The refinement relation is discussed in section 3.3 and defined in section 5.4. Application programmers need not understand, or indeed see, the refinement rules—the rules would typically be implemented at the level of a toolkit-provided runtime system supporting the Workspace Model.

One possible implementation of figure 1.3 is shown in figure 1.4. This is a *centralized* implementation, in which only one copy of the Value component has been instantiated, on Shona's server.

The arrows in the figure are implementation level connectors, indicated by the fact that they have closed arrowheads as opposed to the open arrowheads used at the conceptual level. Arrows with solid lines are *local connectors*, which have essentially the same semantics as procedure calls. The dotted arrows are *remote connectors* which are essentially the same as remote procedure calls.

The implementation level components corresponding to the conceptual Value and Clicker components are visible in the figure. In our notation, shading is used

to distinguish implementation level components from conceptual level components. The other components, identified by symbols, are *infrastructure components* furnished at the implementation level to provide appropriate implementations of the conceptual level semantics. The full suite of infrastructure components is defined in section 5.2. In this figure we see:

-  *concurrency control and consistency maintenance components* (CCCMs), which mediate concurrent access to application components;
-  *transmitters* and  *receivers*, which allow communication over network links between nodes; and
-  a *message broadcaster*, which provides the asynchronous message delivery semantics of the subscription connector.

Since this figure explicitly represents the locations of components on nodes, the links between them, and the places at which concurrency control is applied, it is suitable for analyzing the system in terms of fault tolerance (if Shona's server fails, the count will be lost), performance (for a classic model-view-controller style interaction, updating a clicker's value will require five network transmissions—a request and a response each to update and then read the Value, plus an asynchronous message to inform the Clicker that the Value's state has changed), and security (since the Value is implemented on Shona's server, she has positive control over access and modifications to it).

As mentioned above, figure 1.4 is just one of many implementation architectures corresponding to the conceptual architecture of figure 1.3. Some other possible implementations are shown in figures 6.6, 6.7 and 6.8 on pages 136, 137 and 138.

Developing software using a Workspace Model toolkit involves defining the application components: in this example, the Clicker and Value components. Component definitions are at the conceptual level of the model. A component definition includes the ports that the component provides, the calls and messages sent and received on those ports, and the internal algorithms and data structures used in the component's implementation. At runtime, components are assembled into conceptual level

architectures using evolution calculus operations. The application programmer's view of our fiia toolkit, which implements the full Workspace Model, is provided in section 6.4.

The toolkit's runtime system is responsible for applying the refinement rules in an incremental fashion as the system is configured, and for generating the implementation level architecture. The runtime is free to select any appropriate implementation of the given conceptual architecture, so it is never necessary for the application programmer to be concerned with this. However, for pragmatic reasons it is also possible for the application programmer to provide hints to the runtime regarding appropriate implementation architectures. The design of our fiia runtime system is presented in chapter 7.

1.5 Contributions of this thesis

The core contribution of this thesis is a formally-specified architectural model for synchronous groupware that provides a clean separation of conceptual structure from distributed implementation, that allows automated mapping between these two levels, and that provides explicit support for runtime architectural change at both levels. In particular:

- We have developed a conceptual architectural model that allows developers to concentrate on the function of their application, abstracting low-level distributed systems issues such as network transportation, allocation of components to nodes, and concurrency control. We have shown that this conceptual model allows the expression of the common architectural styles for groupware that have been reported in the literature. We and others have applied the model to the design of several applications ranging from a remote slide presentation tool to an exercise video game [56].
- We have developed an implementation architectural model that exposes the distributed systems issues abstracted in the conceptual model. We have shown that the implementation architectural model is sufficiently expressive to cap-

ture the common groupware distribution architectures reported in the literature. We have gained some practical experience with the model through its use to implement several small applications.

- To link the two models together, we have precisely specified a refinement relation showing how to map a single conceptual architecture to a set of implementation architectures. We have shown by example how different refinement paths can capture tradeoffs in the quality attributes of the system being constructed. We have shown that this form of refinement is suitable for inclusion in a toolkit: specifically, we have specified an implementation model for the refinement relation (via graph transformation rules), and used it to show that all refinement paths terminate in a fully refined implementation architecture, and that refinement steps are composable. By implementing this operational model via a rule engine, we have demonstrated that refinements can be efficiently and incrementally computed in real-time.
- Finally, we have developed an evolution calculus formally specifying runtime change in groupware architectures, and have shown how a runtime implementation can use this evolution calculus to permit changes to the architecture at both the conceptual and implementation level, while retaining their correspondence. We show how this allows runtime evolution of a groupware application, both in the form of programmer-driven changes and changes in the runtime infrastructure (*e.g.*, due to partial failure of the distributed system).

1.6 Presentation

The remainder of this document is structured as follows. Chapter 2 provides a review of the existing literature on groupware architecture. Chapter 3 begins with desiderata for our work, based on the literature review of chapter 2 and amplifying the material from sections 1.2.1 and 1.3. This is followed by a broad overview of the Workspace Model, highlighting its key features and their application in more detail than in section 1.4. A full formal specification of the Model itself is provided in chapters 4 and 5. Chapter 6 illustrates how the Workspace Model may be applied

to scenario-based modeling, design support, architectural analysis and application development. Chapter 7 presents the design of our Workspace runtime system, fiia, which was itself developed using Workspace Model concepts. Chapter 8 provides an evaluation of the Model against the design aims stated in chapter 3. Finally, chapter 9 offers a concluding discussion and suggestions for future work.

Chapters 3 and 6 constitute a self-contained overview of the Workspace Model and its applications and may be read alone.

The two appendices provide a quick-reference summary of the Workspace Model's graphical notation and definitions of certain key terms that are used in chapters 4 and 5.

Chapter 2

Literature Review

In section 1.3 we identified our research motivation as the desire to develop an architectural approach to groupware that supports dynamic evolution, both at the conceptual and implementation levels. In this chapter we survey the state of the art in groupware architectures, in terms of both conceptual architectures and implementation architectures. We follow this with a review of approaches for mapping between conceptual and implementation architectures, and with a discussion of the concept of architectural evolution. We close with a summary of the state of the art in groupware architectures, in comparison with the goals identified in section 1.3.

2.1 Conceptual architecture

Conceptual architectures represent the developer's view of a system's components and connectors, as well as their intended logical configuration at runtime. In this section we survey the common themes that we have identified in groupware architectural reference models and architectural styles [8]. The survey is organized in terms of core themes rather than as a series of tutorial descriptions of architectural styles themselves. A tutorial survey of most of the core groupware architectures is available separately [101].

The common themes we have identified in our review of conceptual architectures for groupware are:

- distribution transparency;
- separation of user interface from program state;
- the use of intermediate layers to coordinate between interface and state components;
- a notification mechanism for reporting state change;
- collaboration through shared state;

- tree structure; and
- collaboration through asynchronous messaging;

Each of these is discussed below, with specific examples from the groupware literature. We also discuss one feature that we rarely encountered in conceptual architectures for groupware: formality.

2.1.1 Distribution transparency

Most of the published conceptual architectures for groupware are *distribution transparent* [30]. This means that the groupware developer may design a groupware system at the level of logical interactions, ignoring (at least temporarily) such issues as process boundaries, interprocess communication, network limitations, and partial failure. Fully distribution transparent architectures include Suite [38], Weasel [57], Clock [60], ALV [66], Dragonfly [4], C2 [128] and the PAC family of architectural models [31, 94, 22, 32].

Some architectural styles for groupware are semi-transparent. For example, GroupKit's shared environments are fully distribution transparent; however GroupKit also includes an explicitly distributed remote procedure call construct [112]. The Java Shared Data Toolkit (JSDT) [19] is based on the Java Remote Method Invocation [5] and therefore requires the programmer to deal explicitly with partial failure in the form of mandatory exception handlers.

2.1.2 Separation of user interface from state

The structuring principle of separating the user interface from the application's state goes back to at least the 1960s [67]. It is now a commonplace in interactive system toolkits, including for example Java Swing [140] and wxWidgets [120]. Representing a classic separation of concerns approach to software design, it is intended to promote simplicity, modifiability and reuse.

In groupware architectures this separation is explicitly visible in the form of the *model* components in Clock [60] and Dragonfly [4], the *abstraction* components in

PAC* [22] and ALV [66], GroupKit's *environments* [112], Suite's *active values* [38], the JSDT's *shared byte arrays* [19] and the Clover architecture's *shared functional core* [77]. It is also visible, at least implicitly, in the C2 architecture's distinction between components' *top* and *bottom* sides [128] and in the *stem* portion of Dewan's generic model of groupware architectures [37] (see section 2.1.6). This last is a generalization of the Arch model for interactive systems [130], which explicitly includes a separate state component.

2.1.3 Intermediate layers

Many architectural approaches include intermediate layers between the interface components and the state components. These intermediate layers may act as mediators or adapters, transforming syntax and semantics to account for mismatches between the user interface and the application state [52]. They may also provide other services, including the control of extra-component communication [31] or the management of the dialogue between the user and the application [100].

The PAC* and Clover architectures provide intermediate layers at both the micro and the macro levels [22, 77]. At the micro level, each PAC component[1] includes a *controller* which mediates between its user interface oriented *presentation* and its application state oriented *abstraction*. At the macro level, PAC* borrows the five layer structure of the Arch architecture [130] and includes *logical interaction*, *dialogue* and *functional core adapter* layers in addition to its user interface *physical interaction* layer and its application state *functional core* layer. Clover extends this architecture by allowing an arbitrary number of layers.

Other architectural models that make explicit provision for intermediate layers include Weasel [57], Clock [60], Dragonfly [4] and C2 [128]. Dewan's generic model of groupware architecture attempts to model all possible groupware architectures in terms of an arbitrary number of layers [37].

[1] Clover components embed PAC components.

2.1.4 Notification mechanism

Complementing the separation of interface from state is the concept of a notification mechanism that advises interested components when the application's state changes. This idea was first formalized in Smalltalk-76 as the model-view-controller user interface paradigm [74] and is well-known as a design pattern under the name Observer [52]. A notification mechanism reduces the coupling between state and interface and allows the possibility of multiple interfaces responding independently and appropriately to state changes.

The most common notification mechanism in groupware architectures is based on *events*, which act as indications of state change. On receipt of an event, the receiving component is expected to perform whatever computation is appropriate to react to it, which may involve communication with the event sender. This technique is used in GroupKit [112], Suite [38], the JSDT [19], C2 [128] and Dragonfly [4]. It also forms the basis of the Notification Service Transfer Protocol (NSTP) [35, 98], an attempt to standardize a general notification service for heterogeneous synchronous groupware. The transmitted events may be simple indications of state change or may carry arbitrary information [118].

An alternate notification mechanism is based on *declarative constraints*, as found in Weasel [57], Clock [60] and ALV [66]. As used in these systems, constraints express a desired relationship between state and interface and are a higher level concept than events; once declared, they are maintained automatically by a constraint satisfaction algorithm in the underlying runtime system. While this is an appealing concept, fully general constraint satisfaction algorithms are difficult to implement efficiently, particularly for arbitrarily distributed systems [148].[2] They can also be difficult for developers to program effectively, occasionally resulting in surprising and difficult to debug behaviours [136].

[2] In special cases, including Clock's tree-structured distributed systems, they can be implemented as efficiently as hand-tuned notification mechanisms [131, 132].

2.1.5 Collaboration through shared state

The combination of separate state components and a notification mechanism allows for collaboration through shared state. In this approach, a single state component is shared by multiple system users. When any user modifies the state, a notification is provided to all. This allows each user's interface to react appropriately. Interaction through shared state is appropriate where groupware users are viewing or modifying some shared artifact. All the groupware architectures mentioned in section 2.1.2 support collaboration through shared state.

2.1.6 Tree structure

Many groupware architectures use a hierarchical tree structure as their principal topology. This structure may be used to identify shared versus private components, to indicate visual containment, to facilitate structured communication between components, or all three.

Groupware architectures deriving from the Arch reference model [130] typically use a tree structure to indicate the distinction between shared and private components. The shared components are found in the common "stem" of the tree, the private components in the individual "branches". Arch-derived groupware architectures include PAC* [22], Clover [77], and Dewan's generic model [37]. Other groupware architectures that are not directly derived from Arch but which use a tree structure to separate private and shared components are Clock [60], Dragonfly [4], C2 [128], and to a lesser extent, GroupKit [112].

Visual containment is a common feature of user interfaces. For example a typical text editor's user interface will contain a menu bar and an editing area; the editing area will contain the text area itself and a scroll bar. Visual containment may be naturally represented by hierarchy; this approach is taken in Clock [60], Dragonfly [4] and ALV [66].

Finally, hierarchy can be used to provide a structured communication mechanism between components. In the PAC family of architectural styles [31, 94, 22, 32] the

PAC components are arranged in a tree, with the control facets managing communication up and down the hierarchy. In Clock [60] and Dragonfly[4], requests which cannot be satisfied locally are automatically propagated up the architectural hierarchy and constraints (in Clock) or events (in Dragonfly) are propagated down. This allows components to be moved easily from one level of the hierarchy to another.

2.1.7 Collaboration through asynchronous messaging

The final theme that we have identified in groupware architectures is communication through asynchronous messaging. This is typically used for the propagation of input events, the distribution of state change notifications, and the delivery of time-based media such as audio and video.

Most architectural styles for groupware (and indeed, most interactive systems in general) include an event-based mechanism for dealing with user input. While at some level these events are actually processed by a polling event loop, from the programmer's perspective they arrive asynchronously. Groupware systems using asynchronous notification of user input include Weasel [57], Clock [60], ALV [66], Dragonfly [4], GroupKit [112] and the JSDT [19]. PAC-based systems may also be implemented using asynchronous input, though this is not a requirement of the style [32].

Systems that use event-based notification mechanisms for state change may do so either synchronously (that is, the response to the notification executes in the event-sender's thread) or asynchronously (in some other, possibly system-provided, thread). From a high-level perspective the original model-view-controller interaction pattern was intended to be asynchronous; however, the actual implementations in the Smalltalk user interface toolkit [74] and derivatives like Java Swing [140] are actually synchronous. Of the groupware architectures surveyed, C2 [128], GroupKit [112], the JSDT [19] and NSTP [98] provide true asynchronous change notification.

Finally, asynchronous messaging is frequently used to deliver time dependent media such as audio and video. This approach is frequently seen in commercial

groupware tools like Skype [7] and MSN Messenger [88]; it has also been used in Clock [59] and in many other communication-oriented groupware tools.

2.1.8 Formality

One feature that we found only rarely in descriptions of conceptual architectures for groupware was formality—most descriptions of groupware architectures are completely informal and may be more or less rigorous. The main exceptions to this rule are C2 and Clock.

C2 is specified in terms of four different levels of abstraction: internal functionality of the component, the interface(s) exported by the component to the rest of the system, interconnection of architectural elements in an architecture, and syntactic rules of the architectural style [87]. This formal specification assists in precise communication among developers using C2, supports automated enforcement of syntactic rules, and provides opportunities for other forms of automated design support and runtime implementation [86].

The Clock language has a formal definition based on temporal logic [54]. The multi-user version of the Clock system adds additional specification mechanisms, including communications protocol models [131]. Clock's formal definition allows for the creation of automated design tools [55] and for experimentation with flexible yet semantically correct distributed implementations of Clock systems [132].

In the next section we present the range of implementation architectures for groupware systems. In section 2.3 we return to the issue of formality, and discuss how formal representations of conceptual architectures can assist in the mapping from the conceptual level to the implementation level.

2.2 Implementation architectures

As discussed in section 2.1.1, most groupware conceptual architectures purposely hide the distributed system aspects of groupware systems from application programmers. This allows the designers of groupware applications to focus more directly on

problems in the application domain, while largely ignoring distributed implementation concerns. However, groupware systems are ultimately implemented as distributed systems, and the choice of distribution architecture will impact the system's performance, security, and fault tolerance, among other attributes.

In this section, we examine the range of implementation architectures for groupware systems that have been reported in the literature. The implementation architectures are illustrated using the Workspace notation introduced in section 1.4 and summarized in appendix A on page 216. A full formal definition of the notation is provided in chapters 4 and 5. Here, as in section 1.4 we provide enough informal explanation to make the meanings of the diagrams clear.

For simplicity, in the discussion that follows we assume that each participant has a user interface that manipulates shared state and (in most cases) private state, using a model-view-controller style interaction with asynchronous notifications. We show two participants in most diagrams. The architectures illustrated may be easily extended to more users and more complex application structures. For example, the shared state might in fact be a complex, active, shared application. Further, an individual application might use an arbitrary mix of the implementation architectures discussed.

The figures in this section are drawn using the Workspace Model's conceptual level notation, which may appear odd given that we are discussing distributed implementation. We have done this in order to keep the diagrams simple, avoiding the implementation-level requirement to explicitly indicate infrastructure components like CCCMs, message broadcasters, transmitters and receivers. We rely on the conceptual level's *anchor* relation to attach conceptual components to nodes; this allows us to depict the implementation architectures abstractly yet precisely. To reduce clutter, ports have been omitted from the diagrams in this section.

2.2.1 Centralized

In a fully centralized architecture, illustrated in figure 2.1(a), the application including all user interfaces is located on a single server and only display services are

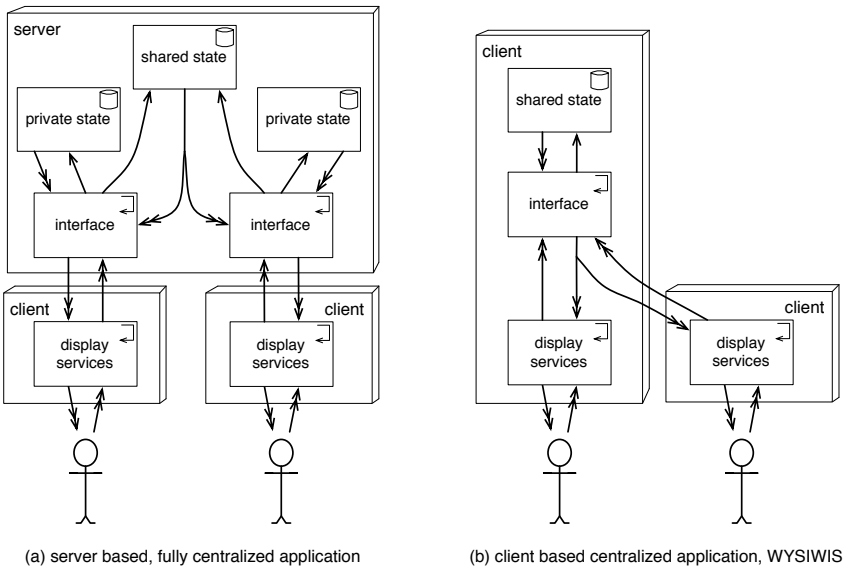


Figure 2.1: Centralized implementation architectures.

found at the users' sites. Communication from the users' sites to the application is via interface-level events; communication in the reverse direction is via rendering requests. The display services component shown in the diagram is used to trap user input and reroute it to the actual user interface, and to render the contents of the user interface on the user's screen. This normally requires either specialized low level software at the client sites and server sites, or a display-server based windowing system like the X Window System [149].

The main benefit of the fully centralized architecture is its simplicity. Designers adopting this approach can largely ignore distributed system issues since the only distribution is performed by the display services. Since there is only one instance of the application running on a single platform, internal efficiency of the application can be maximized and state consistency can be guaranteed relatively easily. The architecture also provides for accommodation of *latecomers* (users who join a groupware session after it has begun), since it is generally practical to provide them with access to the application's shared state or display [28].

The fully centralized architecture tends to be bandwidth intensive and sensitive to network latencies, since communication between the server and the user sites is

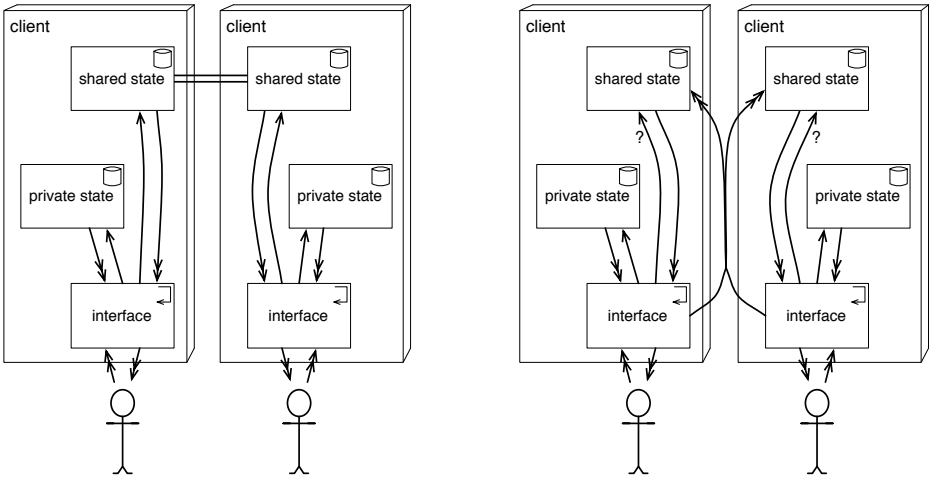
at the level of interface events in both directions. However, performance is often subjectively acceptable on high-speed local area networks [1, 66, 131]. The fully centralized architecture also suffers from poor scalability. If the state updates or interface view recalculation are computationally intensive, or if there is a large state storage requirement per user, the resources of the server can quickly become exhausted as the number of users in the groupware session grows [57]. This problem is compounded by the fact that changes in the shared state will normally require view recomputation to be performed for all users simultaneously [66].

The fully centralized architecture has been used in a number of research systems. The Rendezvous system was implemented using a centralized architecture since this avoided the requirement for a distributed constraint maintenance implementation. Rendezvous' designers considered adapting it to other distribution architectures [66], but implementations were never completed. The Clock runtime system adopts the fully centralized distribution architecture by default; however, Clock can also provide a semi-replicated architecture (section 2.2.3).

A variation of the fully centralized architecture, in which the application is found on one of the client machines and there is no private state, is shown in figure 2.1(b). This variation is used in window sharing systems like XTV [1] and distributed meeting applications like WebEx [142], and WebArrow [141] to enable the *collaboration transparent* sharing of single user applications. It is restricted to a strict What-You-See-Is-What-I-See (WYSIWIS) [122] presentation. This variation is more scalable than the architecture of figure 2.1(a) since only one view need be computed.

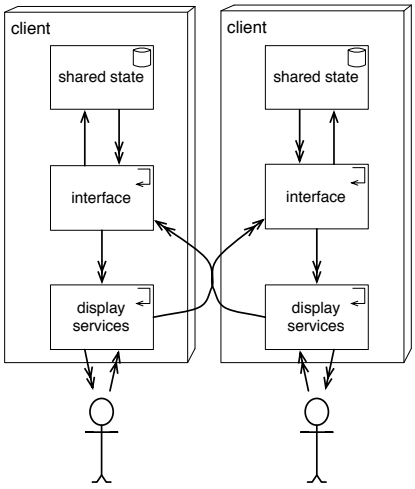
2.2.2 Replicated

A fully replicated architecture is the opposite of a fully centralized one: here all shared data and computation is replicated at all sites. At least three major variants of this architecture are possible. In one variant the consistency of shared state is maintained through an active replica maintenance policy. This is illustrated in figure 2.2(a) by the synchronization connector between the two shared state components. In the second variant, illustrated in figure 2.2(b), the consistency of shared



(a) replication with state synchronization

(b) replication with synchronized state modification



(c) replication with synchronized input

Figure 2.2: Replicated implementation architectures.

state is maintained implicitly by ensuring that all modifications to shared state are identical. (Shared state may be independently queried but not independently modified; this is indicated by the question mark on the call connector in figure 2.2(b).) Finally, in the third variant, illustrated in figure 2.2(c), shared state is maintained implicitly by ensuring that input to all user interfaces is identical; in this case all state is effectively shared since there is no mechanism for manipulating private state.

An obvious liability of replicated distribution architectures is the requirement that a separate copy of the application execute at each users' site. This means that replicated applications require more aggregate resources (processing power, memory, software licenses, *etc.*) than equivalent centralized applications. They may or may not require more aggregate communications bandwidth than centralized applications, depending on the messaging protocols used and the contents of the messages. In environments with a mix of machine types and operating systems, the requirement for identical applications at each site can become a significant constraint, although multi-platform systems like Sun's Java [5] mitigate this somewhat [10].

The main benefit of the replicated architecture is enhanced interface responsiveness. In the state synchronization variant, where an optimistic or operation transform (OT) concurrency control algorithm is used to maintain synchronized state, updates to shared state can be performed locally and are unaffected by network latency [49, 132]. The same is true of the synchronized state modification variant, provided that the ordering policy applied to state modification messages does not add significant overhead. A further benefit of replicated architectures is that they distribute the computationally expensive interface processing to the users' computers. This might be expected to lead to better scalability; however, in practice scalability depends strongly on the overhead incurred in synchronizing the state of the replicated instances. This may be significant, particularly for large numbers of participants.

In the case of the replicated architectures with synchronized state modification or input, some mechanism is required to effect the multi-source, multi-target subscription arrow shown in figures 2.2(b) and (c). If no particular ordering policy is

required, as in the case of an OT-based system, simple broadcast will suffice. Causal or total ordering may be implemented by a fully distributed algorithm like those provided in, *e.g.*, the Horus group communication system [133]. More commonly, total ordering is achieved by routing all user inputs through a central coordinator located on one of the user's machines or a separate server machine. This latter variant is sometimes referred to as a *centrally coordinated* architecture [101].

Research groupware systems using a fully replicated architecture with state synchronization include the CoWord editor [147], DreamTeam [114], Mushroom [71], GroupDesign [70], GINA [12] and the original version of COAST [117]. Commercial systems using this architecture include the multiuser text editor SubEthaEdit [125], the real-time strategy game NetStorm: Islands at War [63], and the various military command and control systems developed under the NATO-sponsored Multilateral Interoperability Programme [91].

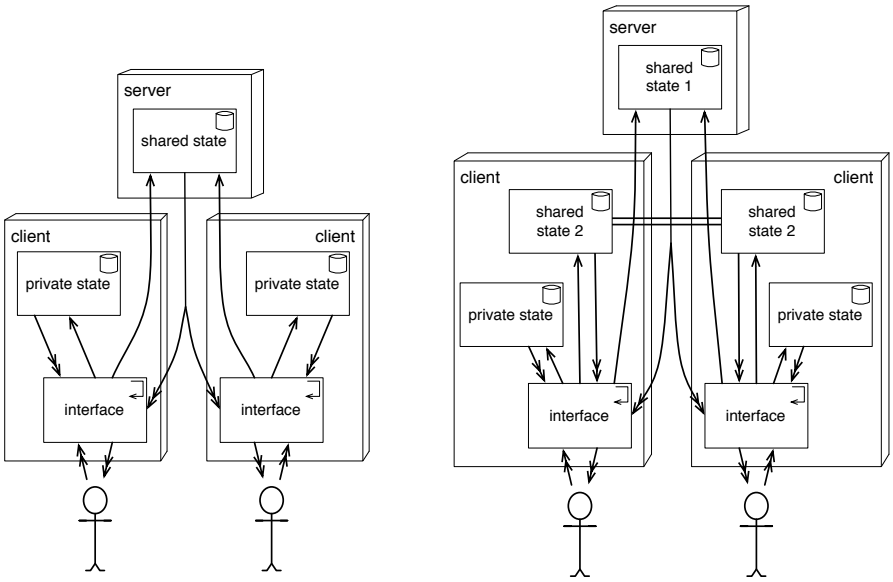
GroupKit is one of the few research systems to use the replication with synchronized state modification architecture; this appears in the form of its multicast remote procedure call construct [110, 112]. The technique is also used in numerous real time strategy games [63] including Age of Empires 2 [14].[3]

Research systems using the replicated architecture with synchronized input include the Java Collaboration Environment (JCE) [2], Java Applets Made Multi-user (JAMM) [10, 9], NCSA Habanero [24], the Prospero system [41], Ensemble [92], and the most recent version of COAST [116]. Commercial systems include the collaborative Word editing feature of Microsoft Groove [64].

2.2.3 Semi-replicated

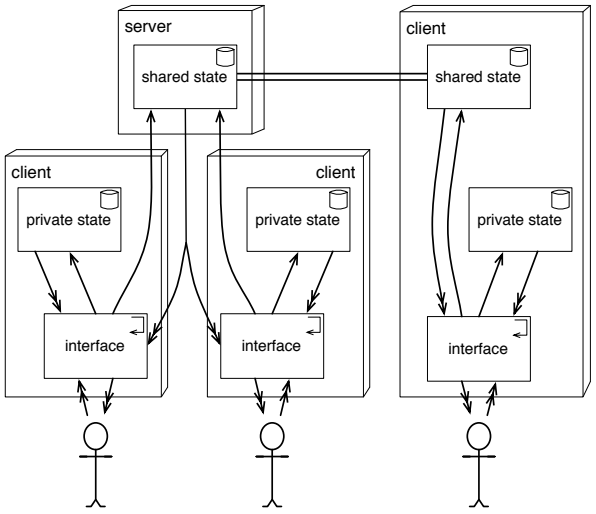
In a semi-replicated distribution architecture some aspects of computation and state are replicated while others are centralized. The policy for determining what is centralized and what is replicated may vary with the application or system. One common approach, illustrated in figure 2.3(a), is to to centralize shared state and repli-

[3] In the case of, *e.g.*, Age of Empires 2 the shared state component shown in figure 2.2(b) is actually a complex application with its own internal thread of control; however, the same general principles hold true.



(a) server-based centralized shared state

(b) combined centralized and replicated shared state



(c) hybrid centralized and replicated shared state

Figure 2.3: Semi replicated architectures.

cate private state and user interface. A minor variant of this approach (not shown) has the shared state centralized on one of the client sites rather than on a server.

Figure 2.3(a) can be seen as a variation of figure 2.1(a) in which the interface and private state have been moved to client machines, removing the requirement for the display services component. As might be expected, the semi-replicated architecture provides a mix of the benefits and liabilities of the centralized and replicated architectures. There is some evidence that with careful tuning the benefits can outweigh the liabilities [132].

The semi-replicated architecture is more flexible than the centralized architecture and accommodates latecomers better than the fully replicated architecture [68]. Semi-replicated applications generally scale better than either fully centralized or replicated ones, since computationally intensive user interface calculations are found at the user sites and communication paths are n-to-one rather than n-to-n [60]. They are also simpler to develop, since consistency maintenance can be managed centrally rather than via a distributed algorithm. If the protocol between the user sites and the server site is standardized, then a variety of user applications can access the shared data simultaneously [34].

The principal liability of the semi-replicated architecture is that responsiveness of the user interface may be impacted by network latencies between the client sites and the server. This effect can be mitigated by the introduction of caches at the client sites at the expense of additional computational and storage overhead [60]. In certain cases it can also be effective to combine the semi-replicated architecture with synchronized shared state and a high-performance consistency maintenance algorithm [131, 132]. This kind of architecture is illustrated in figure 2.3(b). Note that here the shared state has been divided into two parts, one of which is centralized and the other of which is replicated and synchronized.

For more than two users it is also possible to create a *hybrid* architecture in which shared state is centralized for some users and replicated for others. This is illustrated in figure 2.3(c). Hybrids incorporating the architectures of figures 2.1(b) and 2.2(c) are also possible [26]. Hybrid architectures may be attractive when nodes

vary significantly in their capabilities or in the quality of their network links [26].

The semi-replicated architecture is the basis of the Notification Service Transfer Protocol (NSTP) [34, 35, 98], which has been used as the basis of a number of synchronous groupware applications [90]. Variants of the semi-replicated architecture are also found in Suite [38], those GroupKit applications incorporating centralized shared environments [112], Clock [60], the Java Shared Data Toolkit [19], Jupiter [93], Promondia [50], the DOLPHIN system [124], Bentley's system for air traffic control [11], and Neil Stephenson's fictional Metaverse [123].

The semi-replicated architecture is also widely used in multiplayer video games, ranging from games that support on order of ten players such as Half-life [13], Halo 2 [21] and Neverwinter Nights [18], to massively-multiplayer games supporting thousands of concurrent players such as EverQuest [119], Lineage [78] and Star Wars: Galaxies [106].

This completes our review of implementation architectures for groupware. We turn now to the question of how to map from conceptual architectures to implementation architectures.

2.3 Mapping from conceptual to implementation

One approach to mapping from a conceptual architecture to an implementation architecture is to leave the choice entirely to the imagination of the developer. While this clearly provides the ultimate in design time flexibility, it is less powerful than approaches that offer the possibility of automated or semi-automated mappings, implementation assistance, and support for architectural evolution at both the conceptual and the implementation level.

In the current literature there are several alternative approaches to mapping conceptual architectures to distribution architectures. These are:

Principled mapping. Implementation architectures can be developed by designers, at design time, through the application of mapping guidelines. While still manual, this reduces the amount of creativity required on the part of the designer

and supports the reuse of best practices. This approach has been applied for example in the translation of PAC-Amodeus architectures to Java implementations [47].

Tool mapping. The mapping from conceptual architecture to implementation architecture can be performed by a tool. Tools such as Suite [39], GroupKit [112], Clock [132], Rendezvous [66], COAST [117] and the DRADEL toolkit for C2 [86] permit developers to work with relatively high-level, distribution-independent abstractions as described in section 2.1.1. The actual implementation architecture is then generated by the tool without requiring further intervention from the designer. This approach has the drawbacks that the selection of implementation architectures are normally quite limited (in some cases, just one) and the architecture chosen by the tool may not be optimal for either the application or its deployed environment.

Open implementation. Open implementation approaches combine the flexibility of principled mapping with the support of a toolkit. In an open implementation, the toolkit's mapping function may be customized through high-level parameters [39], via meta-languages that control the runtime system [42, 79, 127], or via meta-protocols allowing direct insertion of code into the toolkit itself [48, 95, 83]. For example, Clock allows the developer to provide architectural annotations regarding caching, concurrency control and replication that have an effect on the runtime's implementation behaviour [132]. Similarly, later versions of GroupKit provide a set of command-line switches that allow the choice, at application startup, of centralized or replicated implementations for shared environments [109].

Facet-based mapping. Facet-based mapping is a fine-grained version of open implementation developed for the Dragonfly architecture and its TeleComputingDeveloper toolkit [4]. Each conceptual level component is mapped onto a composite implementation component consisting of a component-specific facet provided by the developer plus system-provided facets that support for concurrency control, caching, and replication. The system provides several versions

of each kind of facet, which may be “plug replaced” at design time to provide different runtime implementations.

Tool mapping, open implementation and facet-based mapping all require some level of formality in the description of both the conceptual and implementation architectures. In some cases, as in Clock and C2, the architecture is formally specified independent of supporting tools. In other cases the only formal representation of the architecture is implicit in the implementation of the tool itself. This makes it difficult to formally reason about conceptual architectures and their implementations, and reduces opportunities for additional tool support.

One common feature of the approaches described above is that they allow for mapping from the conceptual to the implementation level at design time, or at the latest, at application or toolkit startup time. None of these approaches support the kind of fluid runtime behaviour discussed in sections 1.1.1 and 1.3; if we are to achieve truly dynamic groupware then another approach to mapping will be required. This mapping approach must be combined with support, at runtime, for both conceptual and implementation level evolution.

2.4 Evolution

Most groupware systems are designed to support at least a degree of conceptual level evolution. For example, in GroupKit, users may start new collaborative sessions with their associated applications and may join or leave existing collaborative sessions [112]. This obviously implies some degree of implementation evolution as well: new software components must be instantiated, new communication paths must be established, and so on.

However, very few groupware systems provide explicit mechanisms for representing or reasoning about evolution architecturally, at either the conceptual or the implementation levels. In this section we summarize the exceptions that we are aware of, and provide a brief discussion of the related literature on dynamic architectures for distributed systems.

The C2 architecture is based on components communicating via multi-way asynchronous message connectors that may span multiple address spaces [128]. Both components and connectors have “top” and “bottom” sides; a message sent into the bottom of a connector is received at the bottom side of all components that are attached to the connector’s top, and vice versa. C2 supports runtime change through an architecture modification language allowing the dynamic addition and removal of components and connectors, and dynamic “welding” and “unwelding” of components and connectors [96]. C2’s architectural model has an implicit one-to-one mapping between conceptual components and their implementations; however connectors may have complex implementations that change dynamically as components are attached and detached. Connector implementations are not representable in the C2 architectural model and no explicit mechanism is given for reasoning about implementation correctness.

The DACIA system [81, 82] provides a framework for building distributed systems, that can adapt to available resources and support user mobility. It does this through a mobile component model where components may move between process spaces and the connectors between components are automatically preserved during moves. DACIA’s connectors support only point-to-point asynchronous messaging. Its evolution support consists of a set of primitives allowing components to be created, destroyed, moved between hosts, and connected and disconnected from one another. While both papers reporting DACIA provide examples of multiple implementation configurations intended to represent “the same application”, DACIA provides no mechanism for reasoning about the correspondence between application semantics and distributed implementation. DACIA’s architectural representation is at the implementation level only.

While both C2 and DACIA support the creation of dynamic groupware, neither provide explicit representation of the transition between centralized and replicated approaches to the implementation of shared state. Chung and Dewan describe a system with exactly this property in [26]. Their system, which is aimed at the collaboration-transparent sharing of single-user applications, supports the archi-

tectures shown in figures 2.1(b) and 2.2(b), plus "hybrid" combinations of these architectures for three or more users. Their system allows for the creation and destruction of application replicas and display services components and for the re-routing of the connections between them. They provide a formal model of these changes and a distributed coordination language for implementing them. The creation of new synchronized replicas is achieved by logging a compressed version of the complete input event stream and replaying it on replica creation [25, 27].

Aside from the work on dynamic groupware discussed above, there is also a significant body of literature on formal architectural approaches to dynamic distributed systems. A comprehensive summary of this work is provided in [17]. As might be expected, all work in this field is at what we refer to as the implementation level of architectural description. A variety of formalisms have been proposed for discussing architectural dynamics, including graphs (*e.g.*, CommUnity [143]), process algebras (*e.g.*, Darwin [84]), and declarative logics (*e.g.*, ZCL [36]).

2.5 Summary

In this chapter we have surveyed the state of the art in architectures for groupware systems in terms of their conceptual representations, their distributed implementations, the mechanisms available for mapping between these two levels, and their support for runtime change. We have identified that currently available approaches lack certain key features when they are compared to the scenario and goals described in chapter 1.

In the next three chapters we continue our presentation of the Workspace Model. In chapter 3 we present our desiderata for an architecture for groupware, in terms of requirements arising from user needs, application programmer needs, and toolkit developer needs. This is followed by an overview of the Workspace Model, which has been designed to fill those requirements. Chapters 4 and 5 provide a specification of the Workspace Model itself. In chapter 6 we show how the Model can be applied to the design, analysis and implementation of groupware systems that meet the needs of users and application programmers. In chapter 7 we illustrate how the model

supports the development of toolkits by describing the implementation of our own Workspace toolkit, fiia. Then in chapter 8 we return to the requirements identified in chapter 3 and summarize how the Workspace Model addresses them.

Chapter 3

Desiderata and Overview of the Workspace Model

As shown in chapter 2, current architectural models for groupware lack key attributes necessary for the effective development and runtime implementation of scenarios like the one presented in section 1.1.1. In particular, no current model of groupware architecture provides all of the following features:

- a formally-defined, distribution transparent, conceptual level architectural model with appropriate abstractions for the development of groupware;
- a formally-defined implementation architectural model that exposes the distributed system issues abstracted at the conceptual level;
- a formal relation between the two levels, which allows a range of implementations to be computed for any conceptual level architecture; and
- an explicit representation of runtime change.

In the next section we lay out our rationale for these features in more detail, from the perspective of the groupware user, the groupware application developer, and the groupware toolkit implementor. This is followed by a more detailed overview of our own model of groupware architecture, the Workspace Model. The full formal definition of the Workspace Model is provided in chapters 4 and 5.

3.1 Desiderata for a model of groupware architecture

As described in section 1.3, our goal is to produce an architectural model for groupware that allows for fully dynamic behaviour at both the conceptual and implementation levels, with a precise link between the two. Ultimately, this model should support the efficient development of groupware systems possessing the kind of fluid behaviour described in the scenario in section 1.1.1.

In developing our architectural model, we need to ensure we satisfy the needs of three communities: end-users, application programmers, and toolkit developers. End users require that their groupware systems support fluid collaboration, perform efficiently, and behave in a predictable manner. Application programmers need an environment that is appropriately expressive and that doesn't require them to commit prematurely to a distributed implementation. Toolkit implementors need a model that is formally represented, that provides a range of distributed implementations, that supports efficient incremental refinement of conceptual architectures to distributed implementations.

In the sections that follow we discuss these requirements, which constitute a desiderata for a groupware architecture.

3.1.1 User requirements

Users want groupware systems that allow for fluid collaboration, that are acceptably efficient, and that don't provide unnecessary surprises in use. The implications of these requirements on our model are:

Evolution. The model must provide an explicit representation of dynamic change at both the conceptual and implementation levels. This must allow for user-driven reconfiguration in response to changing needs, as well as system driven reconfiguration in response to changes in the runtime environment, including partial failure.

Implementation efficiency. The model must allow for reasonably efficient implementations of groupware systems. To meet this, the model should not introduce unnecessary overheads and should include performance enhancing features such as caching, as well as support for a range of replica consistency maintenance approaches.

3.1.2 Application programmer requirements

As motivated in section 1.1.2, groupware application programmers need an appropriate set of high-level abstractions that simplify the construction of groupware systems and that avoid premature commitment to a particular distributed implementation. The implications of these requirements on our model are:

Conceptual expressiveness. The model must provide support for a range of design approaches and must be capable of modelling a range of interesting groupware systems. As a minimum, we suggest that the conceptual level should support the core architectural features identified in sections 2.1.2 through 2.1.7: separation of user interface from state, provision of intermediate layers, tree structure, a notification mechanism, collaboration through shared state, and asynchronous messaging.

Distribution transparency. It must be possible to represent design time components and their run time architectural configurations without unnecessary reference to how those components or configurations are to be implemented in a distributed system.

3.1.3 Toolkit implementor requirements

Toolkit implementors need to provide programming interfaces and runtime systems that support the application programmer and user needs listed above. In order to do this, they need a formal representation of the syntax and semantics of the systems they are implementing, an appropriate range of distributed system implementations to choose among, and a mechanism that allows for the automatic computation of distributed implementations. The implications of these requirements on our model are:

Formal representation. The architectural representations at both the conceptual and implementation levels must have well-defined syntax and semantics, in order to allow precise representation of evolution, at either level, and of the mapping between levels.

Implementation expressiveness. At the implementation level, the model must allow for replicated, centralized and hybrid approaches to managing shared state and must provide an appropriate range of implementations for any given conceptual architecture. The implementation level must be represented in a way that supports reasoning about performance, security, fault-tolerance, and other key distributed system attributes.

Refinement. The model must provide a formal relation (refinement) between the two architectural levels such that implementation architectures may be derived automatically from conceptual architectures. In order to be useful and implementable, the refinement relation must refine all possible conceptual level architectures to fully refined architectures, and the computation of a refined architecture must be tractable for large architectures.

3.1.4 Summary

This desiderata constitutes the requirements for a model of groupware architecture that have guided us in our development of the Workspace Model. In the remaining sections of this chapter we provide an overview of the model itself, and provide explanations of how its particular features were chosen to meet these design aims. The full formal definition of the Workspace Model follows in the next two chapters.

3.2 Conceptual architecture

In section 1.4.2 we introduced some of the core features of the Workspace Model by means of our very simple “clicker” example. In this section we return to the multi-user CASE tool example of section 1.1.1 and show how it can be represented using the Workspace Model.

Recall that in our scenario Dave and Ian were working on their design for a software system using a CASE tool, on a large touch-screen display across the hall from Dave’s office. They had moved the CASE tool’s tool palette off the touch-screen onto Ian’s tablet computer in order to make more room for their design. Towards

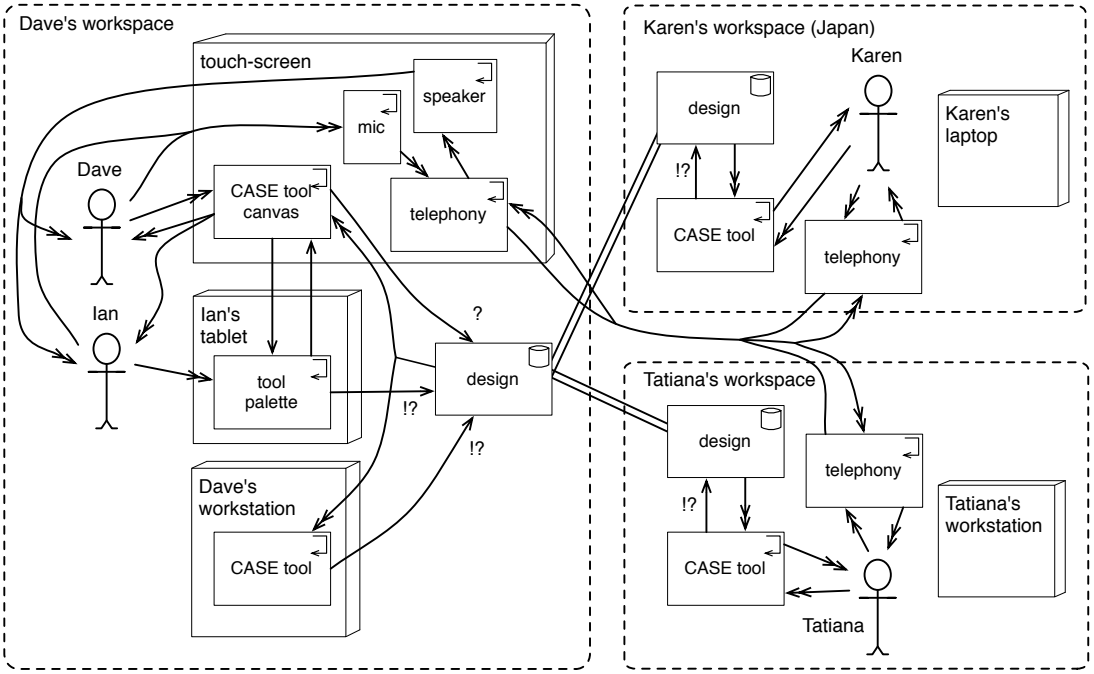


Figure 3.1: Conceptual level view of the multi-user CASE tool.

the end of the scenario, they had invited Karen and Tatiana to join them remotely to discuss what they had created.

A conceptual architecture depicting this final collaborative situation is shown in figure 3.1, using the Workspace notation.[1] Workspace diagrams like this figure represent point-in-time snapshots of the state of a system of interest. The scenario-based analysis [23] of a system will normally require the generation of several such diagrams, with the changes between them specified using the evolution calculus. We show an informal scenario-based analysis of this CASE tool example in section 6.1.

3.2.1 Workspaces

The top-level feature of figure 3.1 is its three *workspaces*, indicated by the dotted lines. A workspace is essentially a scoping mechanism and may contain people, computational *nodes*, software and hardware components, and connectors between them. In the diagram we see Dave, Karen and Tatiana's workspaces and their con-

[1] A summary of the notation is provided in appendix A, starting on page 216.

tents.

For scenario modelling, workspaces provide a useful way to depict independent contexts of use. For runtime implementation, the nodes found in a workspace (discussed below) provide the computational platforms on which software supporting workspace owners may be instantiated.

3.2.2 People

The people found in workspaces represent direct system users and others who may be affected by the system's use, here including our scenario's four participants. Ian is shown in Dave's workspace since he is making use of tools and objects "belonging to" Dave. People are able to provide inputs to, and receive output from, other system elements including other people. This communication is mediated by subscription connectors (the double-headed arrows), which are discussed further below. People are normally the main initiators of activity in the workspace.

The explicit representation of users in architectural models is relatively rare. Since the Workspace Model's conceptual level is intended to allow the representation of collaboration using groupware, people are an essential component of the model.

3.2.3 Nodes

The *nodes* found in workspaces represent identifiable computational platforms that are controlled by the workspace's owner. Dave's workspace includes his desktop, the large touch-screen, and Ian's tablet computer, which Ian has "donated" for his use. In general a single computer may support multiple nodes, which is expected to be the normal case for server computers; a node is essentially defined by an executing process that acts on behalf of the node's owner.

In this scenario we have explicitly represented the three nodes in Dave's workspace, as well as one node each in Karen and Tatiana's workspaces. In some cases we have explicitly located components on nodes in the diagram; this is the *anchoring*

relation discussed below.

A node's presence in a workspace indicates that components in the workspace may be *instantiated* on the node; nodes thus provide a bridge between the conceptual and implementation levels.

3.2.4 Components

Components are represented by rectangles. A component may be physical, like a pen or a whiteboard; software-based, like a document or a user interface; or a component may be an *adapter* between the physical and software worlds [43, 44].

Adapters include devices like mice, keyboards, microphones, speakers, cameras and displays. In figure 3.1 we have shown the mic and speaker adapter components that are associated with the touch-screen display in Dave's workspace and that are used to support the voice connection between the four scenario participants. There would necessarily be other adapters in the represented workspaces; these are abstracted in the connectors from the people to the software components with which they interact.

The user interface and adapter components in Dave's workspace in figure 3.1 are shown *anchored* to particular nodes, while the remaining components are *floating*. The anchoring relation indicates that a component is directly associated with a particular node. As shown in section 3.3, a software component that is anchored to a node must be instantiated on that node; however, a component that is floating may be instantiated on any node in its workspace. The anchor relation allows us to indicate that, for example, adapter components are associated with particular computers, that user interfaces are to appear on particular displays, or that critical information is to be stored on high-availability servers. Since these are all issues of direct interest to users, it is appropriate to represent them at the conceptual level.

The conceptual level includes three kinds of components. *Reactors*, such as the mic, CASE tool canvas (canvas for short) and tool palette components, react to input calls or messages arriving on connectors but are otherwise inert. *Stores* such as the design components are purely passive. They are analogous to the model of

the model-view-controller architecture [74] or the abstraction of PAC [31], with the added feature that they may be used to represent shared state, like the abstraction of ALV [66]. Finally *actors*, which are not found in this particular model, have independent threads of control and may therefore initiate activity in a workspace.

Providing the three different component types in the Workspace Model's conceptual level allows us to reason about the activity and the flow of information through the system, as well as about what information is shared, or potentially shared, between a collaboration's participants.

3.2.5 Connectors

Connectors represent available communication paths between components; the conceptual level includes three kinds of connectors. *Call connectors*, shown with single open arrow heads, are point to point, blocking, and analogous to procedure calls. Complementing call connectors are *subscription connectors* (double open arrow head), which are asynchronous and provide multi-source to multi-target message delivery. The third connector type is the *synchronization* (double line or "equal sign"), which provides an abstract representation of information sharing without defining how that sharing is actually to be implemented.

A single call connector may support a *vocabulary* that includes multiple individual calls. Calls which modify the state of the called component are *updates* and may be indicated by an exclamation mark (!) annotation; calls which return values are *requests* and may be indicated by a question mark (?). If a call connector includes both requests and updates, or if a single call in the connector both modifies state and returns a value, then the connector is a *request-update* connector and may be annotated with both a question mark and an exclamation point (!?).

In the figure, Dave and Ian's tool palette is connected to the design by a request-update connector. This implies that the tool palette can query the current design and also modify it. The canvas is connected to the design by a request connector, indicating that it may only request information from the design, such as its current contents for display purposes.

On call and subscription connectors, the direction of the arrowhead indicates the direction of communication; for call connectors, return values go in the direction opposite the arrow.

Stores that are synchronized (like the design components in figure 3.1) return consistent results for requests and emit consistent message streams; that is, they may be thought of as representing "the same thing." So, in figure 3.1 all the scenario participants are viewing the same design.

Call connectors provide for the normal request-reply semantics of procedure calls, which is a well-understood and frequently-used programming model. Subscription connectors provide asynchronous message streams, including audio and video streams, and thus support collaboration through asynchronous messaging. Synchronization connectors, which are unique to the Workspace Model, provide for collaboration through shared state. As in the example, synchronized stores typically use subscription connectors as a notification mechanism—subscription connectors thus play a dual role in the model.

3.2.6 Ports

Components are attached to connectors at *ports*, which are represented by small circles found on component edges. The symbols within the circles represent the type of connection supported. For call and subscription ports, arrows pointing into the component indicate *target* ports while arrows pointing out of the component indicate *source* ports.

As a shorthand (as in figure 3.1), we often omit ports from our diagrams since the attachment of a connector to a component implies the presence of a port. However, when we are concerned with the design of an individual components, the definition of which ports that component will provide, along with the ports' vocabularies, becomes critical. The formal refinement of a conceptual level architecture to its implementation requires the explicit representation of ports. Ports are shown in the "clicker" example in section 1.4.2 and are shown in connection with the CASE tool scenario in section 6.2.3.

3.2.7 Interpretation

Interpreting the diagram in figure 3.1 requires an understanding of the intended patterns of collaborations between the components. We show how this may be explicitly represented using a UML-like notation [115] in section 6.2.3. Here we provide a short informal description.

The CASE tool in this example is intended to operate using a shared-state, multi-user model-view-controller style of interaction [74, 101]. In Dave's workspace, Dave and Ian are both looking at the canvas component, which shows the current state of the design. Ian controls the tool palette and changes the currently selected tool as necessary. Dave is working at the touch-screen, modifying the canvas. His inputs to the canvas are passed to the tool palette, where they are interpreted by the currently active tool. The tool may query the design and modify the canvas to provide incremental feedback reflecting Dave's actions; when an action is complete, the active tool modifies the design. When the design is modified, it sends out a notification of change via its outgoing subscription connector. This allows the canvas and CASE tool components in Dave's workspace to provide an appropriately updated view.

Since Karen and Tatiana are working with the same design as Dave and Ian (*i.e.*, their design components are synchronized with the one in Dave's workspace), changes originating in Dave's workspace are reflected in their design components, which provide appropriate notifications via their subscription connectors. This allows each participant in the collaboration to be kept aware of changes made by the others. Changes made by either Tatiana or Karen via their CASE tools will similarly be visible to the other participants.

The voice connection specified in the scenario is provided by telephony components, which are interconnected by a multiple-input, multiple-output subscription connector. All four scenario participants provide input to, and receive output from, their telephony components; they can therefore all hear one another speak.

In a scenario-based design process our next step would be to make the diagram more complete, for example by adding in the missing ports. Following this, we would

precisely define intended patterns of interaction between components and the vocabularies required for each of the connectors and ports. The sum of all port vocabularies on a component represents its complete interface; the sum of its expected behaviours in response to each call or message in its vocabularies constitutes its expected behaviour. Thus, from the component's port descriptions we may design internal data structures, algorithms, and required collaborations.

3.3 Refinement

As discussed in section 2.3, a traditional problem with the use of software architectures in groupware development is that many proposed architectural styles are of such a conceptual nature that they bear little obvious correspondence to the technologies used to implement the system. For example Dewan's "zipper" model [37] and the Clover model [77] both provide high-level abstractions for modelling multi-user systems, but neither defines how an application developer should actually implement the models using the facilities provided by available programming languages, frameworks or libraries. Architectural descriptions rarely address how to move from a conceptual architecture to an implementation, leaving it to users of the architectural style (*i.e.*, application developers) with the problem of deciding how to do so.

By contrast, the Workspace Model provides a refinement relation R that precisely defines the legal implementations of any conceptual architecture. This helps developers by providing rules they can follow in the implementation of their conceptual architectures, thus assisting the transition from conceptual to implementation view. The refinement relation also helps toolkit builders by providing precise semantics for implementation decisions that must be embodied in a toolkit. The refinement relation is compositional, has a provably terminating computation, and may be efficiently computed for real systems. See chapter 8 for further discussion of these properties.

Ideally, a toolkit would allow application developers to work entirely at the conceptual level. Their work would consist of providing definitions for conceptual level

components and of defining conceptual level configurations of components and connectors. The toolkit and an associated runtime would then implement the designed conceptual architecture automatically as an appropriate distributed system. We describe a toolkit and runtime that does exactly this in chapter 7.

Refinement applies only to software components and connectors between them. People are not refined, nor are adapter components. The connections between adapter components and software components are assumed to be provided by a lower-level service such as a windowing system.

3.3.1 Refinement rules

The refinement relation R is specified as a set of rules, written as a graph grammar, which show how conceptual elements may be transformed to implementation elements.[2] Each rule specifies one step of a refinement. The refinement relation R is the reflexive transitive closure of the rule set.

A refinement rule consists of a *pattern* and a *replacement*. The pattern, which appears on the left-hand side of the rule, is compared against the current conceptual or partially-refined architecture. When a pattern matches in the architecture, it is replaced by the result found in the right-hand side of the rule. The wavy arrow between the two sides is pronounced “may be refined to”.

The refinement process consists of repeatedly applying refinement rules in any order until no rule matches; at this point we have a fully refined implementation architecture. Different orders of rule application will result in different refinements; the set of all possible refinements of a conceptual architecture c defines the implementation architectures i for which the relation $R(c, i)$ obtains.

A practical implementation of the Workspace Model must provide for incremental application of refinement rules in response to changes at either the conceptual or implementation levels; we address this issue in sections 7.2 and 8.3.3.3.

The refinement rules are primarily of interest to toolkit developers. Application

[2] The rules could also be specified in a more conventional specification language, such as Z [121]. However, in our experience the graphical formalism is easier to read and to reason about.

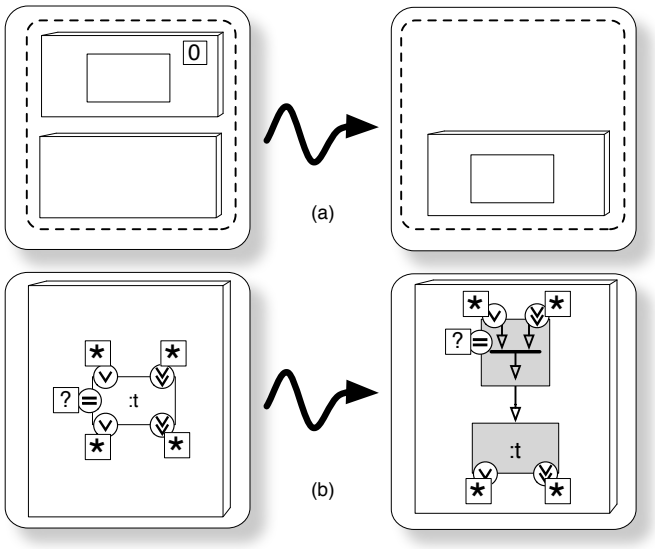


Figure 3.2: Example refinement rules for implementing components.

programmers would normally work at the conceptual level, leaving refinement to the toolkit and runtime system.

3.3.2 Examples

Figure 3.2 shows two of the refinement rules for implementing components. Analogous rules specify the possible refinements for ports and connectors. All of the refinement rules that define the Workspace Model's refinement relation are specified in section 5.4.

The rule in figure 3.2(a) specifies that a component that is not currently anchored to a node may be anchored to any node in its enclosing workspace. On the left-hand side of the rule we see a pattern in which a component is inside a workspace and anchored to exactly zero nodes (the zero in the square is meta-notation indicating "exactly zero"; the placement of a component over a node indicates the "anchored" relation; the enclosure of components and nodes within the workspace dotted line indicates workspace containment). The pattern requires that there also be at least one node in the workspace. On the right-hand side we see the component anchored to the available node.

The rule in figure 3.2(b) specifies how an anchored component of arbitrary type t may be implemented on its anchoring node. The conceptual level component is replaced by an implementation level component of type t (recall that all implementation level components are shown shaded). This is connected to a *concurrency-control and consistency maintenance component* (CCCM), identified by the arrows and crossbar symbol. The CCCM is a special-purpose *infrastructure component* that must be provided by any Workspace runtime system. CCCMs are responsible for mediating conflicting calls and messages and for maintaining the consistency of replicated implementations of synchronized conceptual level stores. Infrastructure components are further discussed in section 3.4 and the full suite of infrastructure components is defined in section 5.2.

The rule in figure 3.2(b) also indicates that the component to be refined may have any number of call and subscription source and target ports, as well as zero or one synchronization ports. The "*" (zero or more) and "?" (zero or one) symbols in the boxes are borrowed from regular expression languages. The right-hand side of the rule shows that the synchronization port and all target ports will be attached to the CCCM component after refinement, while all source ports will be attached to the type t implementation component.

The refinement rules defined in section 5.4 specify the allocation of components to computational nodes, the refinement of conceptual connectors into the types of physical connectors available in real distributed systems, and the introduction of special components to deal with concurrency control, consistency maintenance, message broadcasting, the marshalling of network calls and return values, and caching. In the next section we present an overview of the implementation-level in which these concerns become visible. The full definition of the implementation level is presented in chapter 5.

3.4 Implementation architecture

Like the conceptual level, the implementation level includes connectors, components, ports, and the allowed attachments between them. However, the implemen-

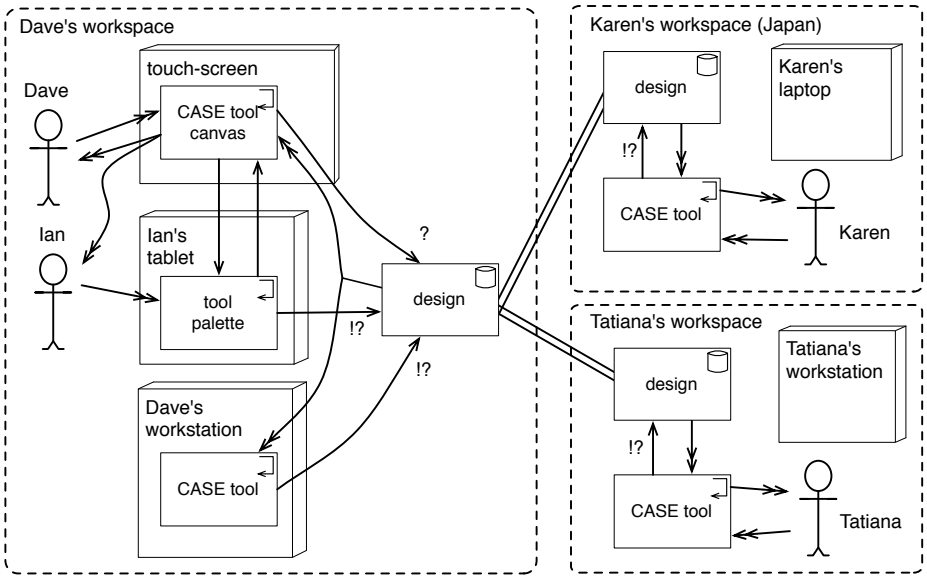


Figure 3.3: A simplified version of figure 3.1 showing only the CASE tool-related components and connectors.

---

tation level elements are simpler and more concrete than conceptual level elements and are designed to be directly implementable in common programming languages.

In this section we illustrate the implementation level using one valid refinement of the CASE tool portion of the conceptual architecture from figure 3.1. For convenience, a diagram of this portion of the conceptual architecture is shown in figure 3.3. The implementation level refinement of this architecture is shown in figure 3.4.

3.4.1 Components

Unlike the conceptual level, which has separate actors, reactors and stores, the implementation level has only one kind of general purpose component, the *implementation component*. However, the implementation level also includes a set of special purpose *infrastructure components*, which are implementation components providing particular services or capabilities necessary for implementing the conceptual level semantics. An infrastructure component is represented by a shaded square

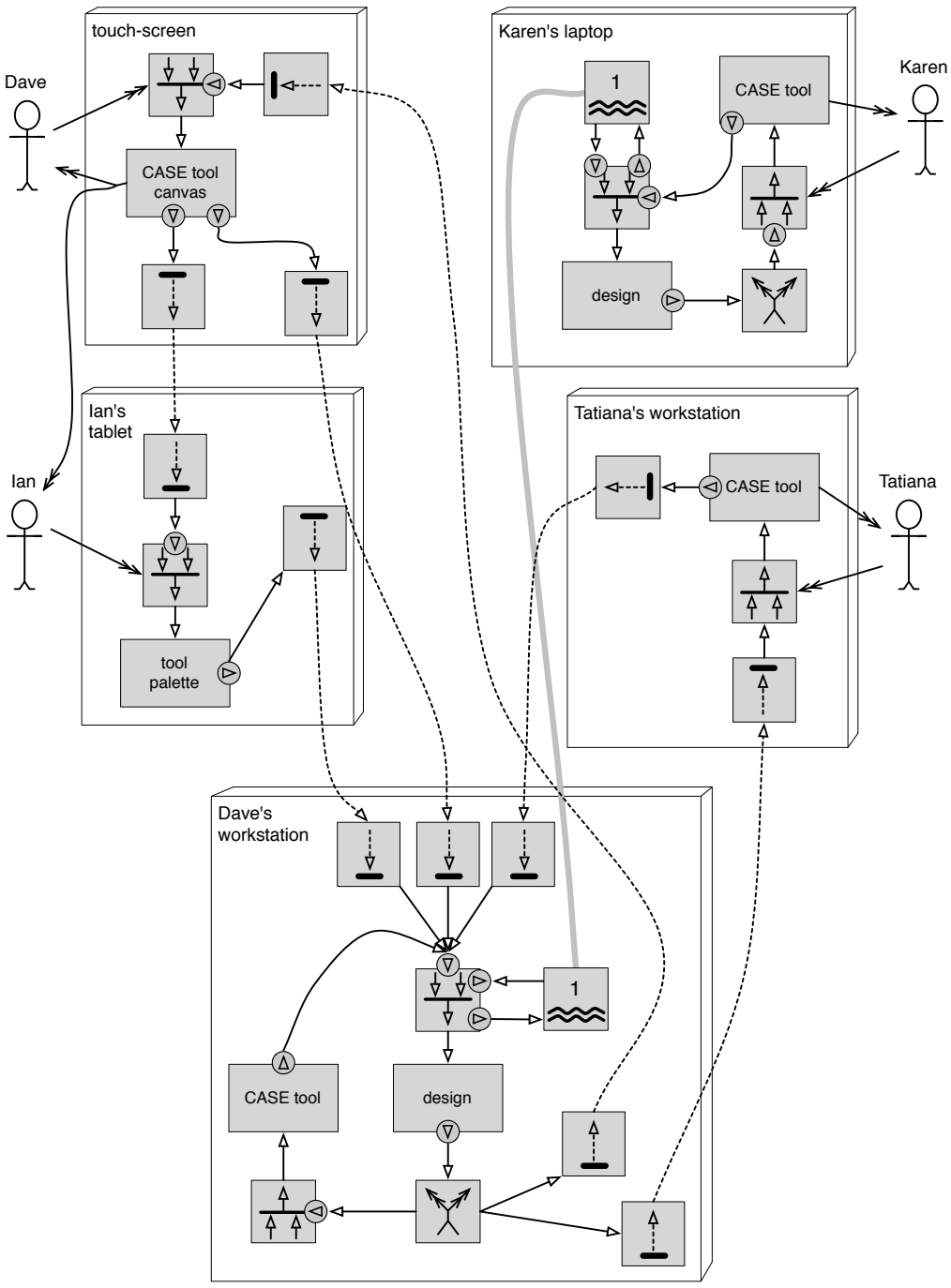


Figure 3.4: One possible implementation of figure 3.3.

with a symbol indicating its kind.[3]

Figure 3.4 shows implementation level counterparts for the CASE tool, CASE tool canvas, tool palette and design components of figure 3.3. Where a component was anchored to a particular node at the conceptual level, its implementation has been instantiated on that node. Components that were floating at the conceptual level have been assigned to nodes by the refinement process. For example, the conceptual level design component in Dave's workspace that is floating in figure 3.3 has been instantiated on the Dave's workstation node in figure 3.4. In addition, the infrastructure components required by the various refinements have also been instantiated.

Four of the types of infrastructure components seen in figure 3.4 (CCCMs, transmitters, receivers and message broadcasters) were introduced in section 1.4.2. Here we review and expand the descriptions presented in that section, and also introduce a new infrastructure component, the *channel endpoint*.

-  Concurrency control and consistency maintenance components (CCCMs) mediate concurrent access to application components. This allows application components to be implemented with an assumption of single threaded semantics, but then deployed in a multi-threaded, multi-user system.

  In addition, CCCM components are responsible for enacting the replica consistency maintenance protocols necessary for keeping replicated stores in sync with one another. In the figure, the two CCCMs supporting the design components on Dave's workstation and Karen's laptop are communicating by means of channel endpoints (below) to enact such a protocol.

-  Transmitters and  receivers allow communication over network links between nodes. They are responsible for marshalling and un-marshalling call and message names, parameters, and return values, and for providing a reliable request-response protocol.

[3] Infrastructure components may be drawn in any orientation; orientation is aesthetic and has no semantic implication.

-  Message broadcasters provide the asynchronous message delivery semantics specified for subscription connectors. When a message is received by a message broadcaster it is queued and the sending thread immediately returns. The message broadcaster then delivers the message using its own thread or threads, effectively decoupling message senders from message receivers. Message broadcasters are also used to support communication between the CCCMs supporting replicated store implementations, which is also asynchronous.

- 

Channel endpoints represent access to multi-point messaging channels. Endpoints with the same channel number (here, channel 1) will all deliver the same messages on their outgoing connectors. Channel endpoints are used in the implementations of some subscription connectors and some synchronization connectors. Channels and channel endpoints are provided by many group communication frameworks, including for example Spread [3] and Horus [133]. They offer a useful and highly efficient asynchronous multi-point message distribution abstraction with ordering and performance guarantees.

There are further infrastructure components including caches and centralized ordering components, all of which described in detail in section 5.2. Infrastructure components provide the support required to implement conceptual level semantics.

3.4.2 Connectors

The implementation level includes two types of connectors: *local connectors*, which may only connect components on the same node, and *remote connectors*, which may connect components on different nodes. Local connectors are indicated by solid lines and remote connectors by dotted lines, both with closed arrowheads. Both connector types are point to point and blocking. Local connectors may be implemented by direct references and procedure calls or method invocations. Remote connectors represent network messaging and are found only between transmitters and receivers.

The thick grey line shown between the channel endpoints is not part of the formal Workspace notation, but is a useful visual indication that two or more channel endpoints are communicating on the same channel. It can be considered as a pseudo-connector.

3.4.3 Ports

At the implementation level the only type of port is the ⊳ *local port*. As with conceptual call and subscription ports, an arrowhead pointing into a component indicates a target port and an arrowhead pointing out of a component indicates a source port.

Local ports are found only in the implementations of conceptual level ports and not in other connections between infrastructure components. They are responsible for providing the pass-by-value semantics of conceptual level connectors. A local source port may act as the source of exactly one connector, but a local target port may be the target of multiple connectors. An infrastructure component may be both the source and the target of multiple local connectors.

3.4.4 Synchronization implementations

The Workspace Model refines synchronized stores to use any of the centralized, replicated and hybrid shared state strategies illustrated in figure 2.2(a) and figures 2.3(a) and (c). The other distribution architectures discussed in section 2.2 are also supported by the Workspace Model; however, these are not automatic refinements for synchronized stores since they would have an effect on the semantics of other application components.

In the conceptual architecture of figure 3.3 there are three design stores that are synchronized with one another. The implementation shown in figure 3.4 provides this synchronization using a hybrid architecture: the stores in Dave and Karen's workspaces are replicated, while the stores in Dave and Tatiana's workspaces have a centralized implementation on the Dave's workstation node. In this particular sce-

nario this is probably a reasonable architecture: Tatiana's workstation is connected to Dave's by a high-speed local network, so any latencies in her interactions with the shared design will be minimal; Karen in Japan has a much slower network so a local replica may provide her with a more responsive interaction.

As mentioned above, the replicated shared state approach requires that any changes to one state replica also be reflected in all others using some *replica consistency maintenance* algorithm, for example using locking, two-phase commit, a distributed operation transform [126] or an undo/redo protocol like ORESTE [69]. The interactive performance of the replicas will be strongly affected by the choice of algorithm. Enacting the algorithm is the responsibility of the replicated components' associated CCCMs; the communications required is provided by shared channels between the CCCMs. In figure 3.4 these channels are represented by the channel endpoints; other implementations of channels, which do not rely on group communication frameworks, are also allowed by the refinement rules.

As discussed in section 2.2, the various strategies for the distributed implementation of shared state provide different quality attributes in different situations. The Workspace Model is unique among groupware architectures in allowing any combination of strategies to be chosen and dynamically modified at runtime.

3.4.5 Multiple implementations are possible

It is important to note that figure 3.4 represents just one valid refinement of the conceptual architecture of figure 3.3. For example, different decisions could have been made regarding the allocation of components to nodes or the implementation strategy for shared data. Further, a *cache* might be introduced on the Tatiana's workstation node to retain copies of previously-seen design information, improving responsiveness when this information is revisited.

3.5 Evolution calculus

As discussed in section 2.4, an important characteristic of groupware systems is their need to support runtime evolution. Evolution can come as a result of participants entering or leaving a collaborative session; as a result of participants moving from one location to another, perhaps using different devices; as a result of participants' goals changing, affecting their tools and how they are used; and as a result of changes to the underlying distributed system such as network failure or the introduction of a new node.

The Workspace Model's evolution calculus allows us to model change resulting from any of these stimuli. Changing users, locations, tasks or goals typically result in change at the conceptual level. Distributed system changes typically result in change at the implementation level. When an evolution occurs at one level, the refinement rules are used to find a sequence of further evolutions at either or both levels such that the refinement relation R between the levels may be restored.

The evolution calculus consists of a set of operations at each of the two levels. Operations are defined using a graph-based notation similar to that used for refinement rules. Each evolution operation definition consists of an operation signature, a pattern and a result. When the operation is invoked on an architecture that matches the pattern, the architecture is transformed such that the elements of the pattern now match the result. Where an operation fails to match a pattern the architecture is not modified.

Two sample operation definitions are shown in figure 3.5, one at the conceptual level and one at the implementation level. There are a total of twenty-nine operations in the calculus, twenty at the conceptual level and nine at the implementation level. These are defined using a total of fifty operation definitions. These definitions are given in sections 4.3 for the conceptual level and 5.3 for the implementation level.

Figure 3.5(a) partially defines the attach operation that allows a store to join a synchronization. The operation's signature is attach(A, k, p) where A is the archi-

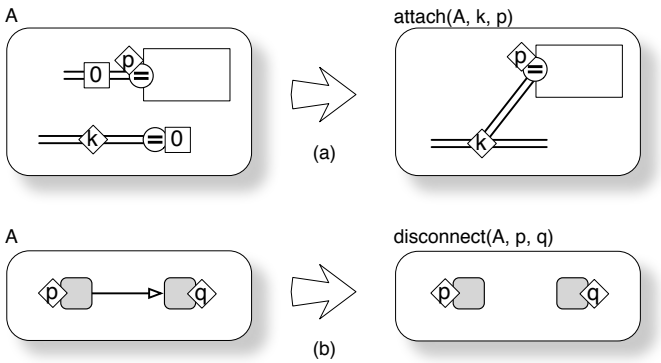


Figure 3.5: Example evolution operation definitions.

tecture to which the operation is applied and p and k are the identifiers of a synchronization port and a synchronization connector respectively. The pattern for this operation will match if A contains a synchronization port p (identifiers are shown in diamonds) that is not attached to any synchronization connectors (the zero in the square) and A also contains a synchronization connector k that is attached to no ports. The result of the operation is identical to A except that k is attached to p. There is another rule with the same signature allowing a store to attach to a synchronization connector that is already attached to other stores.

Figure 3.5(b) shows a disconnect operation at the implementation level. The rounded-cornered squares are generic symbols that represent implementation level components or ports. Thus, this definition matches if there is a connector or port p that is connected to a connector or port q via a local connector. The result of the operation is to disconnect the two, destroying the connector in the process. This evolution might be invoked in response to a conceptual level change or as a result of a network failure.

In response to evolutions at the conceptual and evolution level, further evolutions may be carried out at one or both levels that return the system to a state where the current conceptual architecture refines to the current implementation architecture, as illustrated in figure 1.1. In this way, traceability between the two levels is maintained. Additionally, it is possible to apply evolutions at either the conceptual or implementation level, depending on which is more appropriate for the

evolution being specified. For example, adding a new participant to a collaboration would initially be reflected as a change at the conceptual level (with corresponding changes to the implementation), whereas the addition of a cache to a link in order to improve performance would be an evolution at the implementation level only.

3.6 Conclusion

The Workspace Model provides a distribution-transparent conceptual level, a precise formal process for the implementation of conceptual architectures as distributed systems, and explicit support for the sorts of runtime evolution that occur over the lifetimes of groupware applications. The Workspace Model has been specifically designed to meet the requirements for a model of groupware architecture that were laid out in section 3.1.

The next two chapters provide the full, formal definition of the Workspace Model. Chapter 4 presents the conceptual level and its evolution operations. Chapter 5 presents the implementation level, its evolution operations, and the refinements between levels. Readers wishing to proceed to an overview of applications of the Workspace Model may prefer to skip to chapter 6 on page 121.

Chapter 4

Core Elements and the Conceptual Level

As discussed in the previous chapter, The Workspace Model is divided into a *conceptual level* and an *implementation level*, with two core constructs (people and computational nodes) that are visible at both levels. This chapter specifies the core constructs and the conceptual level including the conceptual level evolution calculus and a set of reflection operations allowing inspection of the current conceptual architecture. Chapter 5 specifies the implementation level, its evolution calculus, and the refinement relation that maps between the two levels.

The terms *side-effect free*, *request*, *update*, *request-update*, *passive*, *active*, *deterministic*, *non-deterministic*, and *consistent* are used in specific technical senses in these chapters. For definitions, see appendix B on page 218.

4.1 Core model elements

The notation for the Workspace Model's core elements is shown in figure 4.1. The two core elements are *person* and *node*. These are briefly introduced below. However, since people and nodes are present at both the conceptual and implementation levels of the Workspace model, they are discussed further in the presentations of those levels.

4.1.1 Person

The people in workspace diagrams are the *raison d'être* and main initiators of activity in the system. Other elements in the system exist to support their activities. Because of this, we often describe the Workspace Model as a "human centred architectural style".

People are also treated as a special type of component in workspace diagrams. See section 4.2.5 for more details.

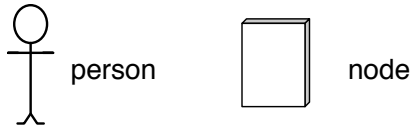


Figure 4.1: Core notation.

4.1.2 Computational node

In order to support virtual objects, workspaces contain computational nodes. A node represents an identifiable element of computing power available to the owner within the workspace. For example, a node might be a laptop computer or a process running on the owner's behalf on a remote server. A node is always contained within a single workspace; this is indicated by graphical containment. Nodes are non-overlapping.

4.2 Conceptual level model elements

The conceptual level of the Model is intended to serve two purposes. First, it supports the description of multi-user interaction scenarios, which may be employed in requirements gathering or in early-phase architectural exploration for a given system. This use is described in more detail in chapter 6 and in [102].

Second, the conceptual level can be used to provide a precise but abstract software architectural description which may later be transformed automatically into a running implementation. This transformation is described in detail in section 5.4.

The conceptual level notation is summarised in figure 4.2. In addition to the two core constructs already presented in figure 4.1, the conceptual level includes a small set of *component*, *connector* and *port* types. Roughly speaking, components are things, connectors are communication paths between things, and ports are attachment points for connectors, found on the surfaces of components.

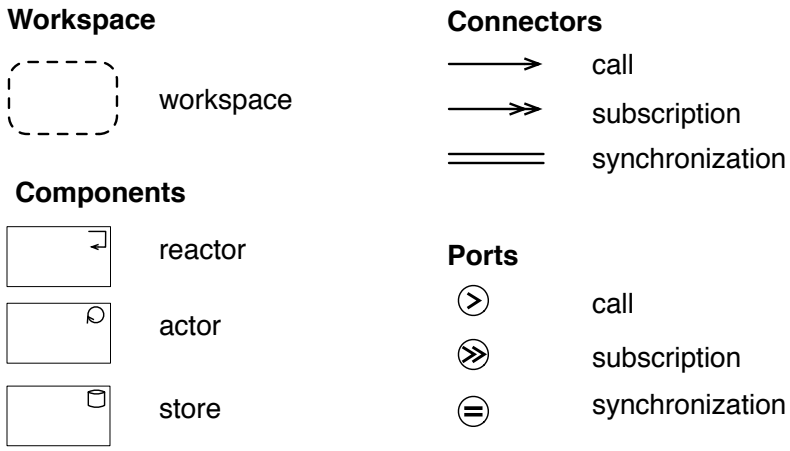


Figure 4.2: Conceptual level notation.

4.2.1 Workspace

A workspace serves to bound a collection of people and the physical and virtual objects (components) that support their activities. Every workspace has an *owner*, who is a person. Items in the workspace belong to the owner. Ownership is indicated as an *attribute* (see section 4.2.6).

Workspaces are always distinct. In the visual language, this is indicated by the constraint that workspaces may not be drawn as overlapping.

4.2.2 Components

Components represent the objects found within a workspace. The objects may be purely physical (*e.g.*, a whiteboard), purely virtual (*e.g.*, a slide in an electronic presentation), or they may act as adapters between the physical and virtual worlds (*e.g.*, a computer display, video camera, or mouse). Each component exists in exactly one workspace and is owned by the owner of that workspace.

Within the Workspace model we distinguish between three kinds of components as shown in figure 4.2: *reactors*, *actors*, and *stores*.

It is difficult to explain characteristics of the three kinds of components without first discussing connectors and ports. We therefore delay complete definition of components to section 4.2.5. However, in the interim an approximate intuition is:

- A reactor is a software or hardware component that is passive (inert until acted upon, see appendix B). Once acted upon it may send messages to, or directly operate on, other components to which it is connected.

- An actor is a software or hardware component that is active; that is, it has the ability to independently initiate activity within the workspace by sending messages or directly operating on other components.

- A store is a software or hardware component that is passive. It may send asynchronous messages to other components but may not act directly on them. In addition, stores may represent information that is shared between workspaces.

Components have types and may also have names. These may be specified using a textual notation in the centre of the component symbol. For example, a component called “myEditor” of type “emacs” might be shown with myEditor:emacs in its centre; an unnamed component of type “emacs” might be shown with the label :emacs. The name and type are visible as attributes of the component (see section 4.2.6).

4.2.3 Connectors

In the Workspace Model’s conceptual level, connectors are first-class entities that may be attached to components at ports. There are three kinds of connectors: *call*, *subscription*, and *synchronization*. Call connectors allow synchronous method invocations; subscription connectors allow asynchronous one-way message delivery; and synchronization connectors mediate state-sharing within and across workspaces.

Call and subscription connectors are directed, so they have *source* and *target* ends. Communication may be initiated only at source ends. In the diagrammatic notation, the target end is indicated by an arrowhead.

Synchronization connectors are undirected; however, order of attachment to a synchronization connector has semantic significance. See section 4.2.3.4 for details.

In this section we first discuss an important restriction on the values passed by workspace connectors, then describe the three kinds of connectors in more detail.

4.2.3.1 Values passed By Workspace connectors

All values passed by workspace connectors must be either immutable or passed by value. This prevents a component from having a direct reference to another component's internal state, which is undesirable since it would allow for inter-component communication that is not architecturally visible.

In effect, this restriction creates a strong semantic division at the component boundary. Objects on the "inside" of a component may hold arbitrary references to other objects inside the same component. However, all communication between components must be mediated by workspace connectors.

4.2.3.2 Call

A call connector allows a source component to invoke methods[1] provided by a port on a target component. An individual method invocation is referred to as a call. Call connectors have the following characteristics:

- Call connectors connect a single source to a single target.
- The source and target of a call connector must be within the same workspace.
- Calls have a blocking semantics; that is, the thread of control that initiated the call at the source end of the connector blocks until the call completes on the target.
- Call connectors have an associated vocabulary which identifies the calls they support. The vocabulary of a call connector may include requests, updates, and update-requests in any combination (see appendix B).
- If a call connector's vocabulary consists entirely of requests it is a *request connector*, which may be indicated graphically by a ? annotation on the connector arrow.
- If a call connector's vocabulary consists entirely of updates it is an *update connector*, indicated graphically by a ! annotation.

[1] "Methods" is used in a generic sense in this document and should not be taken to imply that components need be implemented in object-oriented languages.

- If a call connector's vocabulary consists of a combination of requests and updates, or includes one or more request-updates, it is a *request-update* connector, indicated graphically by a !? annotation. Call connectors without annotations are also assumed to be request-update connectors.

4.2.3.3 Subscription

A subscription connector allows one or more sources to provide a stream of *messages* to one or more targets (subscription connectors are many-to-many). The term "message" is used here in a broad sense: for example, notification of a mouse click might be carried by a message, as might a frame of video forming part of a video stream.

Subscription connectors have the following characteristics:

- Subscriptions may connect one or more source ports to one or more target ports.
- Sources and targets of a given subscription connector need not be in the same workspace.
- Subscription connectors are non-blocking: that is, delivery of a message into a subscription connector may return before the message has been delivered to the subscription's target(s).
- Each subscription connector has a vocabulary of messages it can pass.

4.2.3.4 Synchronization

A synchronization connector allows two or more stores, which need not be in the same workspace, to be mutually synchronized. The intuition is that if stores are synchronized then they are intended to represent "the same object". For example, if there is a store representing a document in one workspace and a similar store in another workspace that is synchronized with the first, then the two stores represent the same document.[2]

[2] Note that this says nothing about the actual implementation of the document object or objects.

A group of mutually synchronized stores is referred to as a *synchronization group*. Within a synchronization group, components converge to consistency (see appendix B) and all message streams emitted from the components are consistent. More precisely, the first condition means that at any time t_1, there exists time $t_2 \geqslant t_1$, such that if the system is quiescent from t_1, at t_2 the synchronized components will be consistent.

Synchronization connectors have the following properties:

- A synchronization connector may connect any number of stores.
- The connected stores may be in different workspaces.
- The connected stores must be of the same concrete type.[3]
- Unlike call and subscription connectors, components do not explicitly communicate with one another over synchronization connectors. Rather, synchronization connectors reflect the presence of mechanisms in the underlying runtime system which keep components mutually consistent.
- As a notational convenience, we allow multiple point-to point synchronization connector symbols to represent a single synchronization group. There is no ambiguity in this representation since a store may be a member of at most one synchronization group. Figure 4.3 illustrates five semantically equivalent synchronization group depictions. The canonical depiction is shown at the centre of the diagram.
- Synchronizations are undirected; that is, there is no concept of source and target in synchronization groups.
- When a store *s* joins an existing synchronization group g and *s* is inconsistent with the members of g, it is *s* that is modified to bring about consistency.

4.2.4 Ports

Components may provide any number of *ports*, which represent attachment points for connectors. Ports are created dynamically on component surfaces. If “static”

[3] In future versions of the Workspace Model, we hope to be able to relax this definition to some form of type compatibility.

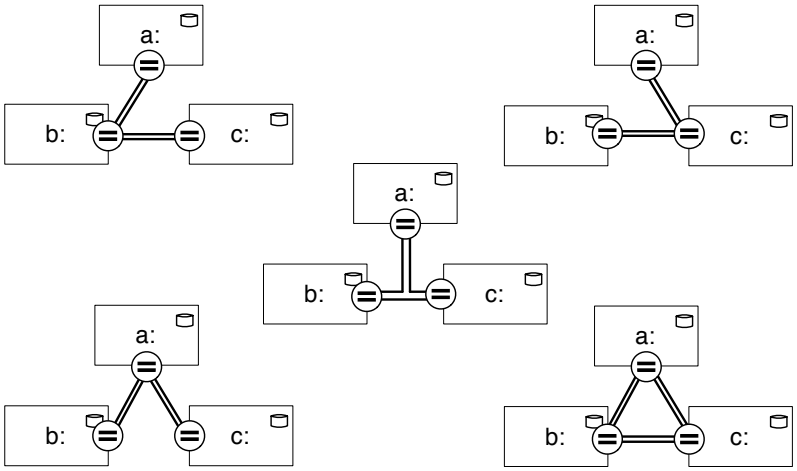


Figure 4.3: Semantically equivalent synchronization group depictions, canonical description in the centre.

---

or “permanent” ports are required, they can be created at component instantiation time and not destroyed. The kinds of ports a component may provide is constrained by component kind; see section 4.2.5 for details.

For directional connectors (calls and subscriptions) there are separate source and target ports. Communication is initiated at source ports and delivered to target ports. In the graphical representation of directional ports, source ports are shown with the arrow pointing out of the component and target ports are shown with the arrow pointing into the component.

Each port is given a name at the time of its creation. Particularly in the case of source ports, this name is expected to be used by the runtime system to map ports to syntactically visible elements in the providing component’s code. However, it is possible to create a port that serves no useful purpose; it is up to the component designer and runtime user to ensure that port creations are sensible.

4.2.4.1 Call ports

A call source port may be attached to at most one outgoing connector. If a component requires multiple references to similar objects — for example a set of shapes in a drawing editor or a list of pages in a presentation editor — this is implemented

by creating one call source port for each reference.

A call target port may have any number of incoming connectors.

Each call port defines a vocabulary of calls that it supports — in effect, an interface. If the vocabulary of the target port is not a superset of the vocabulary of the source port, runtime errors (reported as exceptions or by some other means) may result.

4.2.4.2 Subscription ports

A subscription source port may be attached any number of outgoing connectors. As with call source ports a component that requires multiple outgoing subscription connectors (perhaps to serve different groups of target components) may provide multiple subscription source ports.

A subscription target port may have any number of incoming connectors.

Each subscription port defines a vocabulary of messages that it supports. Messages received at a target port which are not in that port's vocabulary may be ignored.

4.2.4.3 Synchronization ports

A synchronization port may be attached to at most one synchronization connector (in the canonical representation). The vocabulary of a synchronization port is implicitly defined by the type of the store on which it is found. A store may provide no more than one synchronization port.

4.2.5 Components redux

Now that we have defined connectors and ports, we return to the definition of the three kinds of components: reactors, actors and stores. We also discuss people, who play a role similar to that of an actor component in workspace architectural descriptions.

4.2.5.1 Reactor

A reactor is a component that may store data, perform computation and act directly on other components by means of calls. A reactor:

- must be passive and deterministic,
- may provide call source and target ports,
- may provide subscription source and target ports, and
- must not provide a synchronization port.

Physical objects that react to their environment are often modelled as reactors, as are hardware components that generate messages only in response to external action (such as keyboards or mice). Software components without their own threads of control (such as the objects in most object-oriented languages) are also modelled as reactors.

4.2.5.2 Actor

An actor is a component that may store data, perform computation, act directly on other components and initiate activity in a workspace. An actor:

- may be active and non-deterministic,
- may provide call source and target ports,
- may provide subscription source and target ports, and
- must not provide a synchronization port.

Hardware components that independently generate messages (*e.g.*, cameras and displays) or that behave in a non-deterministic manner according to the definition of appendix B (*e.g.*, clocks) are often modelled as actors. Software components with internal threads of control (*e.g.*, servers) are likewise modelled as actors.

4.2.5.3 Store

A store is a component that may store data and perform computation, but that may not act directly on other components. In essence, stores are data storage end points.

In addition, stores may be sharable across workspaces; that is, they may be synchronized with other stores. A store:

- must be passive and deterministic.
- may provide call target ports,
- must not provide call source ports,
- may provide subscription source and target ports, and
- may provide one synchronization port.

Physical objects that act principally as data stores (*e.g.*, books and whiteboards) may be modelled as stores, as would the Model component in the Model-View-Controller architecture [74] or the Abstraction component in the PAC architecture [31]. A telephone call might be partially modelled as two synchronized stores, one in the workspace of each call participant.

4.2.5.4 Person

In workspace architecture diagrams, people may be viewed as a particular kind of component, similar in nature to actors. From this perspective, they have the following characteristics:

- People may be active and non-deterministic.
- A person is assumed to have subscription source and target ports, to be able to create messages (*e.g.*, by physical movement) and to receive messages (*e.g.*, by direct sensory perception). For example, a person may manipulate the location of, and click the buttons of, a mouse, may draw on a whiteboard, or may observe the contents of a display. All of these are modelled using subscription connectors with appropriate vocabularies.
- A person provides neither call ports nor synchronization ports.

4.2.6 Attributes

Workspaces, connectors, components and ports may have arbitrary attributes associated with them. For example, a workspace might have an owner attribute and a

component might have a name attribute. There are two kinds of attributes:

Observed. An observed attribute represents currently-observed state and may be a dynamically computed value.

Intent. An intent attribute may be set to any one of its allowed values using the appropriate evolution calculus operation.

Each intent attribute α has a corresponding observed attribute α'. Components may have observed attributes that are not associated with intent attributes. For instance, a connector might have a "lag" attribute that gives currently-observed communication delay on the connector.

Attributes may be used to provide "implementation hints" to the underlying run-time system. For example, a synchronization connector might be given an attribute suggesting that a centralized implementation would be most appropriate.

4.2.7 Relations among conceptual level elements

Conceptual level elements may be related to one another in several ways, which are depicted in workspace diagrams using simple diagrammatic conventions. The relations and diagrammatic conventions are described below.

Containment. Workspaces may contain nodes, components and connectors. If a node, component or connector e is depicted inside the boundary of a workspace w, then e is contained within w. The exceptions to this rule are subscription and synchronization connectors, which are not subject to the containment relation. The containment relation affects certain evolution and refinement operations.

Port Provision. Components provide ports. If a port p is depicted on the boundary of a component c, then p is provided by c.

Attachment. Connectors may be attached to components at ports. If the end of a connector k is shown touching a port p, then k is attached to p. In cases where the meaning is clear, we occasionally omit ports in workspace diagrams. If

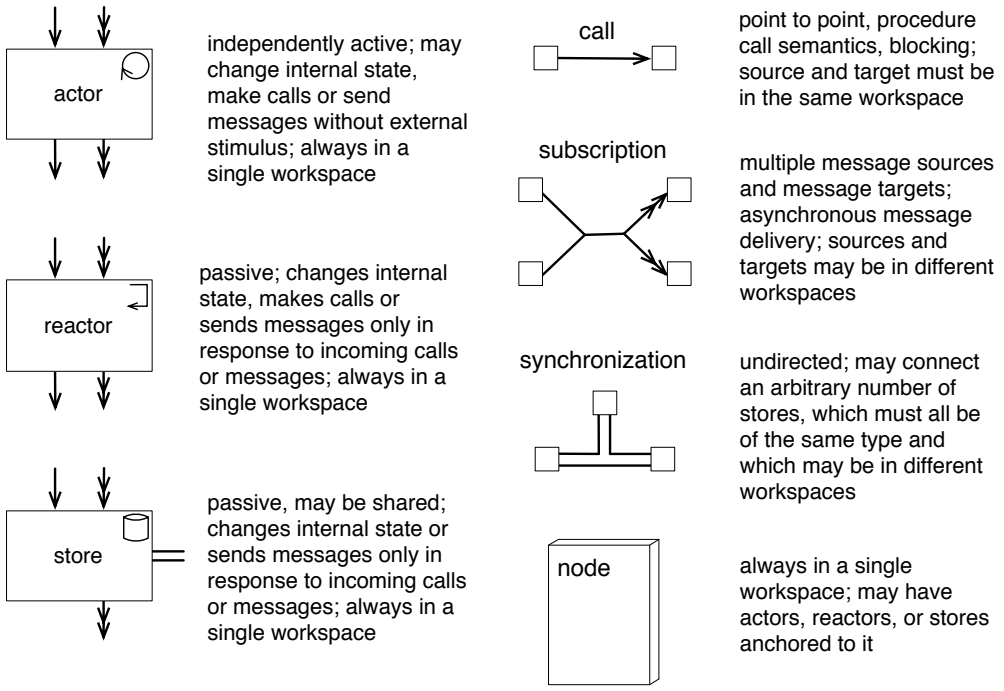


Figure 4.4: A simplified summary of the main elements of the conceptual level.

---

a connector k is shown touching the boundary of a component c, then k is attached to some (unseen) port p provided by c.

Anchoring. Components may be anchored to nodes. If a component c is shown superimposed on a node n in a workspace diagram, then c is anchored to n. A component that is anchored to a node must be instantiated on that node.

4.2.8 Conceptual level summary

A simplified summary of the conceptual level's main elements is provided in figure 4.4. In the figure, ports are not shown but should be assumed to exist everywhere that connectors attach to components.

Each kind of component (actor, reactor and store) is shown with the kinds of connectors to which it may be attached. An arbitrary number of connectors of each type may be attached to a component, with the exception that stores may be attached to at most one synchronization connector.

4.3 Conceptual level evolution calculus

The complete configuration of conceptual level workspace elements at any point in time is referred to as a *runtime conceptual level architecture* or simply *architecture*. Architectures change over time as workspace elements are added to or removed from them, or as the relationships between those elements are altered.

The *evolution calculus* is an algebra consisting of the universe of architectures and allowed operations over architectures. In this section we define the conceptual level of the evolution calculus. There is also an implementation level calculus, which is defined in section 5.3.

Operations in the evolution calculus are specified in an algebraic style using a diagrammatic notation. The specifications make use of the core and conceptual level notation already introduced in figures 4.1 and 4.2, as well as a meta-notation, which is introduced in figure 4.5 and described in section 4.3.1.

Each operation in the calculus is of the form $\omega(A, p^*)$, where ω is the operation, A is the current architecture, and p^* is a list of parameters. The result of each operation is a new architecture A'.

The effect of each operation $\omega(A, p^*)$ in the calculus is specified using one or more diagram pairs, each consisting of left- and right-hand sides (see the evolution specification element in figure 4.5). The left-hand side of the diagram represents a pattern that must be matched in A (that is, the pattern must appear as a part of A) for the operation to succeed. Essentially, the left-hand side is a precondition.

If the left-hand pattern can be matched for an operation $\omega(A, p^*)$, then the architecture A' resulting from $\omega(A, p^*)$ differs from A in exactly the same ways that the left-hand side of the diagram differs from the right-hand side. Differences may include the presence or absence of workspace elements as well as alterations in any of the relations defined in section 4.2.7. Any workspace elements or relations not explicitly depicted in the left-hand side are unchanged in A'. The one exception is for operations which destroy components: if a component is destroyed, it is removed from any relationships in which it previously participated even if that relationship

is not explicitly shown on the diagram. (For example, we can prove by structural induction that the node shown on the left side of figure 4.7 (b) is necessarily contained in a workspace; after the node is destroyed it no longer participates in the containment relation.)

Some operations are specified by multiple diagram pairs (*e.g.*, createCallSource in figure 4.9). An operation $\omega(A, p^*)$ that can be matched against any one of its corresponding diagrams will complete.

If no pattern corresponding to an operation can be matched, then the operation is an identity on the architecture, that is, $\omega(A, p^*) = A$. In an implementation of the Workspace Model, an operation resulting in an identity on the architecture would be expected to raise an exception, since its use likely represents an error.

Any architecture that may be produced by starting from an *empty architecture* (one containing no elements) and applying some sequence of evolution calculus operations is a syntactically correct (well-formed) architecture. The evolution calculus is therefore analogous to a *production system* that generates syntactically correct architectures, where each operation specification is a production. In this view, the set of evolution specifications constitutes a graph grammar for the language of architectures.

4.3.1 Meta-notation and pattern matches

The meta-notation used in evolution calculus and refinement definitions is shown in figure 4.5. The meta-notation includes a template for evolution calculus specifications, a set of generic workspace element symbols, a means of identifying particular workspace elements, and symbols for cardinality constraints. For convenience, the generic symbols for both the conceptual and the implementation level are shown here; implementation level elements are shown shaded. See chapter 5 for further discussion.

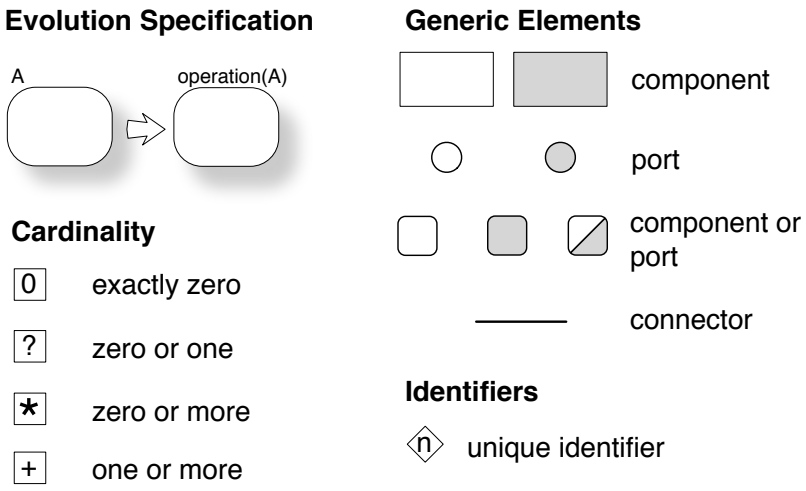


Figure 4.5: Meta-notation used in evolution calculus and refinement diagrams. Shaded elements are implementation level.

4.3.1.1 Evolution specification

As discussed in the previous section, an operation $\omega(\mathcal{A}, p^*)$ is specified by one or more diagrams, which take the form of the evolution specification of figure 4.5. The left-hand side of the diagram (labelled A) represents a precondition which must be matched in $\mathcal{A}$ for the operation to succeed.

For a pattern to match, all elements shown in the left-hand side of the diagram must be present in $\mathcal{A}$, with the identifiers and cardinalities given, and taking part in any relations depicted in the diagram (see section 4.2.7).

4.3.1.2 Cardinality

Cardinality constraints on pattern matches are given using standard symbols borrowed from regular expression languages, enclosed in squares. Where a cardinality symbol is shown on an element that participates in one or more relations with other elements, a match requires that there be that many elements participating in the given relation. Relations bind tighter than cardinality constraints. An element with no cardinality symbol must have a cardinality of exactly one.

The “exactly zero” cardinality allows a match only where there are exactly zero

of the indicated elements in any depicted relations. For example, the left-hand side of figure 4.6(a) will match if A includes a workspace w that contains exactly zero components, zero nodes, and zero call connectors.

One slightly tricky case is where an element is both shown "naked" (cardinality one) and separately with a cardinality indicator of zero. This means that the element appears exactly once in the depicted relations. For example, in figure 5.12(a) the conceptual channel must be the source of exactly one call connector and the target of exactly one call connector for the left-hand side of the diagram to match.

All cardinality indicators are *greedy*; that is, if an appropriate match is found then the match includes all matching elements.

4.3.1.3 Generic elements

Generic elements are useful to indicate that a particular rule will match more than one kind of element. (They are not essential, but reduce the number of diagrams necessary to specify the full set of evolution and refinement rules.)

Generic components match reactors, stores and actors. Generic ports match call, subscription and synchronization ports. The generic "component or port" symbol matches all components and ports. The generic connector matches calls, subscriptions and synchronizations. Unshaded generic component, port and "component or port" symbols match only conceptual level elements; shaded ones match only implementation level elements, and the half-shaded "component or port" symbol matches all components or ports at either level.

4.3.1.4 Identifiers

Each workspace element in an architecture has a unique identifier indicated by a diamond. These are used in evolution calculus operation signatures to specify the operation's target(s).

Where an operation will result in creation of a new workspace element, that element's identifier forms part of the operation's signature and the identifier must

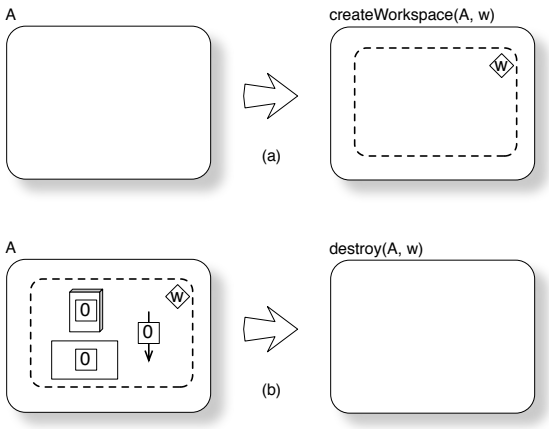


Figure 4.6: Conceptual level operations on workspaces.

not be in use in A. This ensures that an identifier identifies at most one workspace element.

In the following sections we define the effects of the conceptual level evolution calculus operations using our diagrammatic notation. Each diagram is accompanied by explanatory text intended as an aid to its interpretation. Uses of the meta-notation are explained in the description of the diagram in which they first appear.

4.3.2 Workspaces

Figure 4.6 (a). It is always possible to create a new workspace. (The left-hand side of the diagram is empty indicating the null precondition). New workspaces are initially empty. (This is indicated by the fact that the only change between the two sides of the diagram is the appearance of workspace w; no other relationships in the architecture, including workspace containment, are changed.)

Figure 4.6 (b). A workspace may be destroyed only if it is empty. (The zero cardinality indicator on the node, component and connector symbols matches in a greedy fashion within the scope of any represented relations, here containment within the workspace w.)

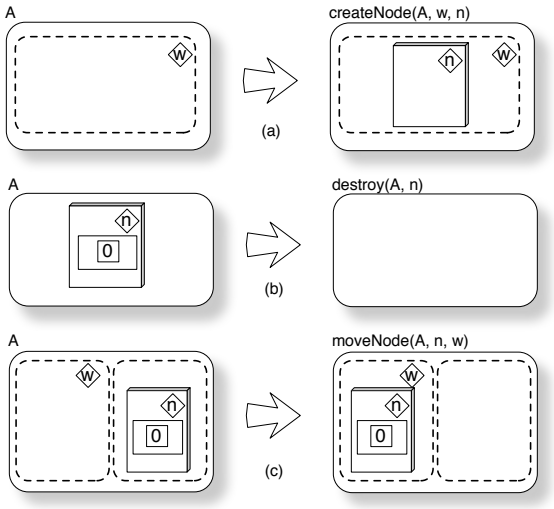


Figure 4.7: Conceptual level operations on nodes.

4.3.3 Nodes

Figure 4.7 (a). A node may always be created within an existing workspace. The created node is contained in the workspace and initially has no components anchored to it. Anchoring is defined in section 4.3.6.

Figure 4.7 (b). A node with no anchored components may be destroyed.

Figure 4.7 (c). A node with no anchored components may be moved into a particular workspace. This moves it out of the workspace in which it had previously been contained. An attempt to move a node into the workspace in which it is already contained will fail to match the left-hand side of the diagram (which shows two distinct workspaces, w and an anonymous one); however, it will still have the desired (null) effect.

4.3.4 Components

Figure 4.8 (a), (b) and (c). A component may be created within a workspace. A newly-created component has no ports and is neither attached to any connector nor anchored to any node.

Figure 4.8 (d). A component may be destroyed, provided none of its ports are at-

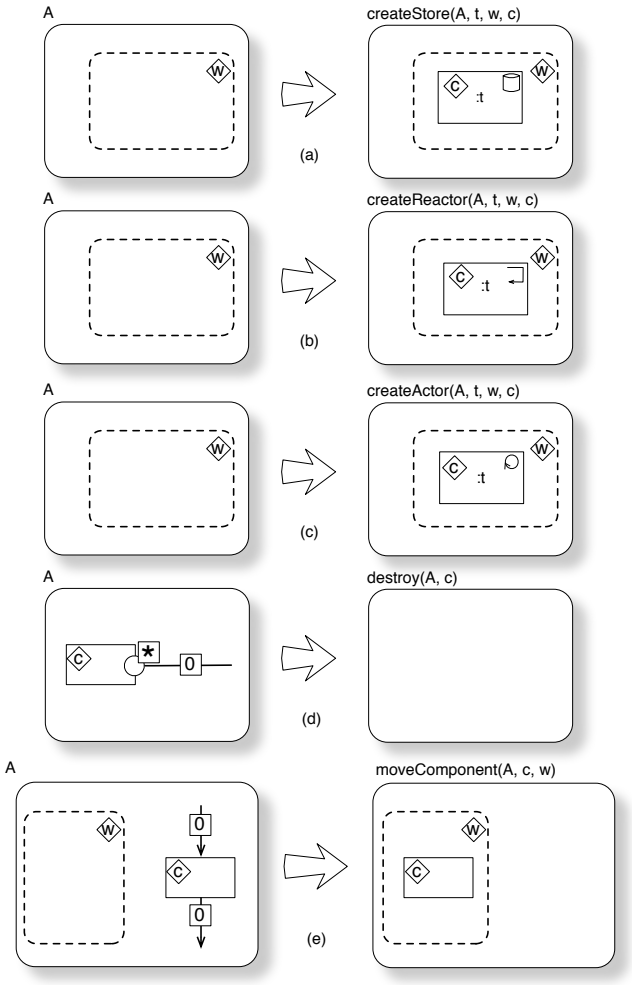


Figure 4.8: Conceptual level operations on components.

---

tached to any connectors (see section 4.3.7). Destroying a component also destroys all ports present on its interface (see section 4.3.5).

Figure 4.8 (e). A component may be moved from one workspace to another, provided none of its ports are attached to call connectors. Such a move leaves synchronization and subscription connectors attached (since they are not shown in the figure) and does not change the provision relation of the component's ports. For clarity, ports are not shown in this figure.

4.3.5 Ports

Ports may be created on component boundaries. The kind of the component constrains the kinds of ports that may be created on it. On creation a port is assigned a name, indicated here by n. Ports are always unattached when created. Attachment is defined in section 4.3.8.

Figure 4.9 (a) and (b). Call source ports may be created only on actors and reactors.

Figure 4.9 (c), (d) and (e). Call targets ports and subscription source and target ports may be created on any component.

Figure 4.9 (f). Synchronization source ports may be created only on stores. A store may provide a maximum of one synchronization port.

Figure 4.9 (g). A port may be destroyed only if it is not attached to any connector.

4.3.6 Components and nodes

Software components need not be associated with particular nodes at the conceptual level of the workspace calculus. However, it frequently makes sense to associate hardware components with particular nodes, *e.g.*, to indicate that a particular mouse is attached to a particular computer. It may also make sense to associate software components to particular nodes, *e.g.*, for performance reasons or to ensure that certain information is present on a laptop that is about to be disconnected from a network.

Figure 4.10 (a). A component may be *anchored* to a node that is contained within the same workspace, as long as it is not already anchored to another node. Anchoring a component has no effect on ports or connectors.

Figure 4.10 (b). A component that is anchored to a node may be floated off of it. Floating a component leaves it in the same workspace. Floating a component has no effect on ports or connectors. A component that is floating may be instantiated on any available node in its containing workspace.

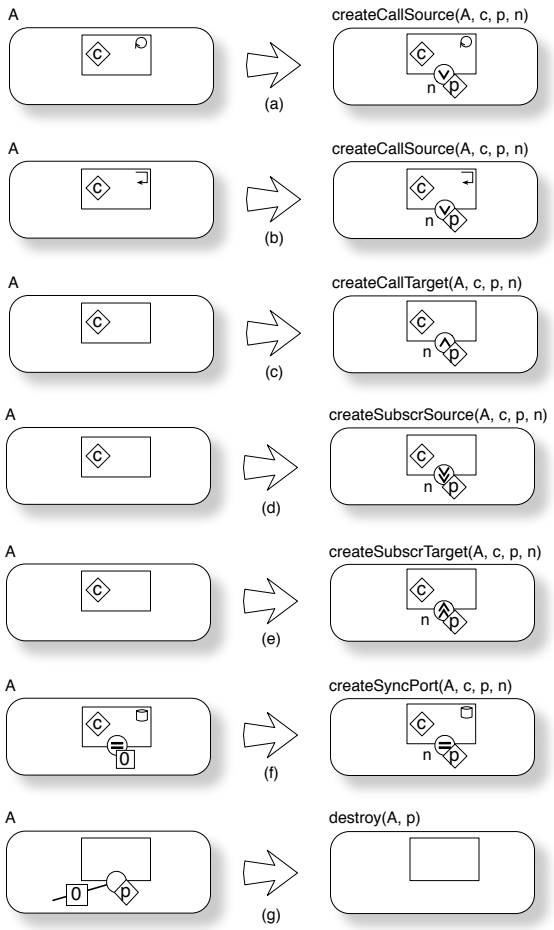


Figure 4.9: Port creation and destruction.

4.3.7 Connectors

Figure 4.11 (a) and (b). Call connectors may be created within workspaces. Newly created connectors are not initially attached to ports.

Figure 4.11 (b) and (c). Subscription and synchronization connectors are not subject to workspace containment and may simply be created.

Figure 4.11 (d), (e) and (f). Connectors that are not attached to ports may be destroyed. See section 4.3.8 for the definitions of attachment and detachment.

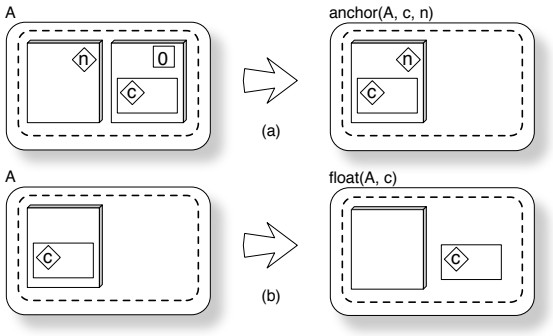


Figure 4.10: Anchoring and floating components.

4.3.8 Connectors and ports

Connectors may be attached to, and detached from, corresponding ports. Call connectors may be attached only to ports in the same workspace; subscription and synchronization connectors are not restricted by workspace boundaries. Since ports are always on components, we occasionally refer to connectors as being attached to components where this does not result in a loss of clarity.

Figure 4.12 (a). A call connector with an unattached source end may be attached to a call source port that is also unattached.

Figure 4.12 (b). A call connector with an unattached target end may be attached to a call target port, regardless of any other connectors attached to that port. (This allows a call target port to have multiple incoming connectors.)

Figure 4.12 (c). A subscription connector may be attached to a subscription source port. A subscription source may be the source of multiple connectors, and a subscription connector may have multiple sources.

Figure 4.12 (d). A subscription connector may be attached to a subscription target port, regardless of any other connectors attached to that port. As with sources, a subscription connector may be attached to multiple target ports and a subscription target port may accept multiple incoming connectors.

Figure 4.12 (e). A synchronization connector that is not attached to any ports may be attached to an unattached synchronization port.

Figure 4.12 (f). A synchronization connector that is already attached to one or more

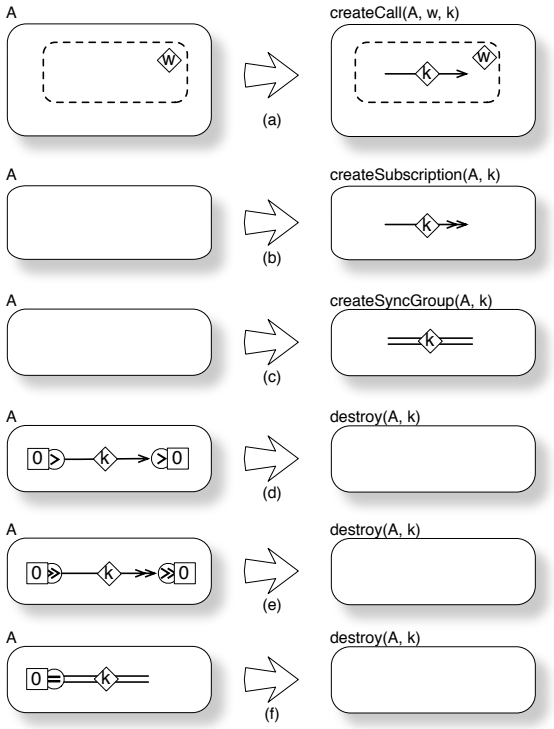


Figure 4.11: Conceptual level operations on connectors.

components may be attached to a free synchronization port on another component, provided that all attached components are of the same type (here indicated by t). *Figure 4.13 (a) through (e).* Any attached connector/port pair may be detached.

4.3.9 People

As discussed in section 4.2.5.4, a person may provide subscription source and target ports which may be attached to subscription connectors. These represent the person's ability to perceive the environment and supply input to the system. Subscription connectors between people and components do not form part of the formal model and so are not specified here.

4.3.10 Attribute modification

As discussed in section 4.2.6, any workspace element may have arbitrary attributes associated with it. The attributes are not normally shown in the architectural di-

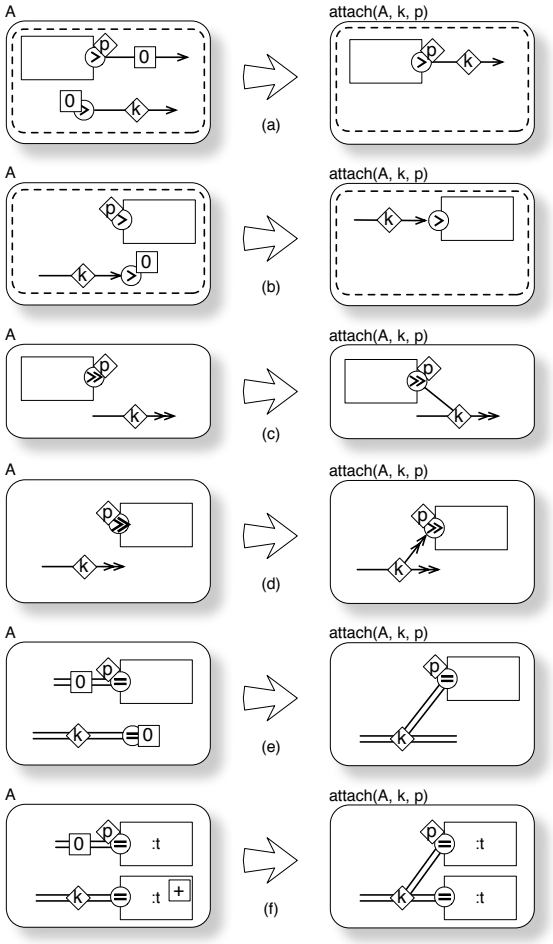


Figure 4.12: Attaching connectors to ports.

---

agrams in this specification. The list of these attributes and values of the intent attributes may be modified at runtime.

The operation for setting the value of an attribute is:

$setAttribute : A \times e \times \text{string} \times \text{value} \rightarrow A$

The effect of this operation on an architecture in A is to set the value of an intent attribute, named by the string, on an element from e to the given value. The value may be of any type.

Some attributes, such as a component's type, are immutable. An attempt to modify an immutable attribute will fail.

Attributes may also be deleted. The operation to delete an attribute is:

$delAttribute : A \times e \times string \rightarrow A$

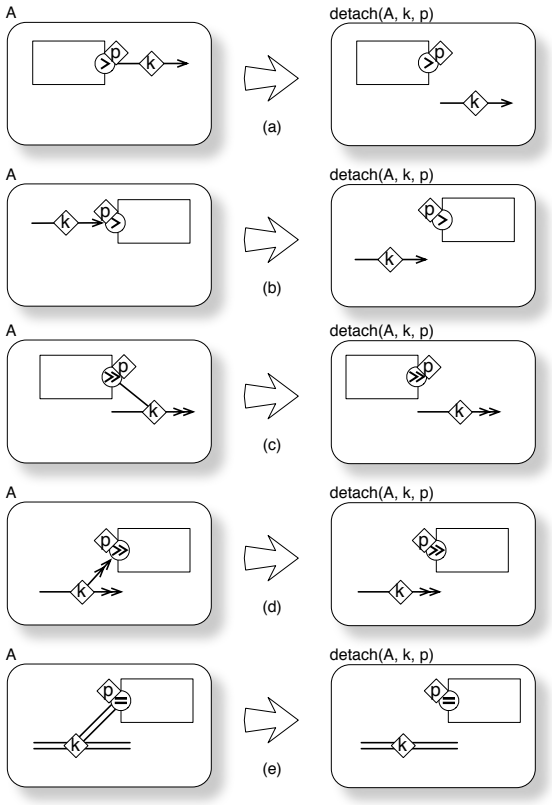


Figure 4.13: Detaching connectors from ports.

4.3.11 An example evolution sequence

Figure 4.14 illustrates a sequence of conceptual level evolution operations, which creates a portion of the architecture illustrated in figure 1.2 on page 12. The figure is a time sequence, read from top to bottom, showing successive snapshots of the conceptual architecture A. At each point in time, we show the evolution or evolutions performed on the left, the resulting architecture in the centre, and explanatory notes on the right. In the explanatory notes, the numbers in square brackets refer to the figure numbers of the evolution operation specifications used in the evolutions.

For this example, we assume that we start with an empty architecture A. In step (a) of figure 4.14 we apply the createWorkspace operation to A, supplying an identifier of 1 for the new workspace.[4] The pattern shown in the left-hand side of the

[4]Recall that identifiers used in the creation of new objects must not be in use in the architecture prior to the object's creation; that is, all identifiers must be unique.

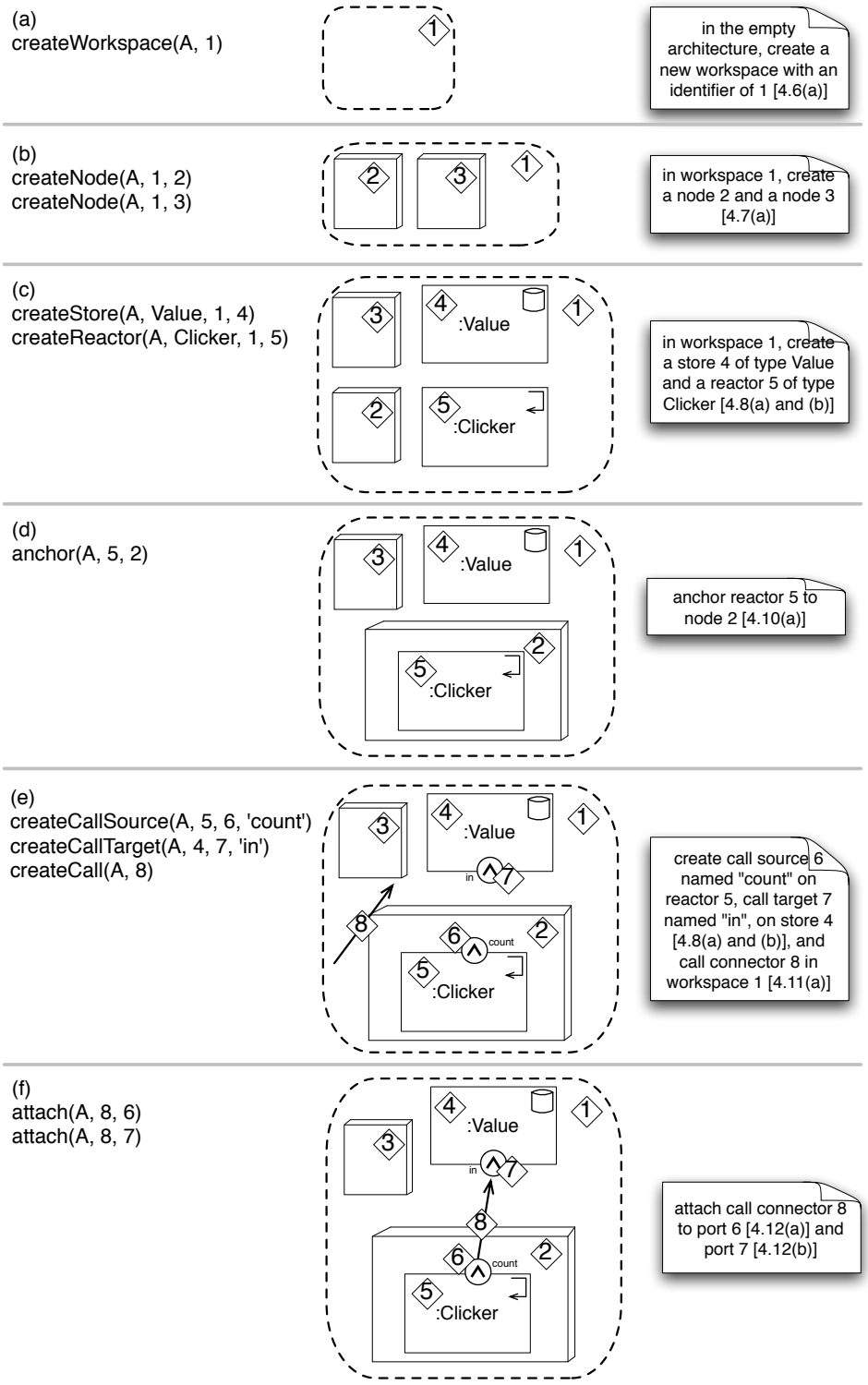


Figure 4.14: A sample evolution sequence, showing the creation of part of the Clicker example from section 1.4.2.

evolution specification in 4.6(a) matches the empty architecture (in fact, it matches any architecture), so the architecture A is modified such that it matches the right-hand side of the specification. This results in the architecture shown in the centre column of figure, which now includes a workspace with an identifier of 1.

In step (b) we then apply the createNode operation twice, creating two nodes in the workspace 1, identified by the numbers 2 and 3. Note that if we had attempted to apply the createNode operations in step (b) before having created the workspace, the operations would have failed since the left-hand side of the operation specification in figure 4.7(a) would have failed to match.

In step (c) we create a store 4 of type Value and a reactor 5 of type Clicker, both in workspace 1. The types Clicker and Value are assumed to be defined external to the Workspace Model.

In step (d) we anchor the Clicker reactor 5 to the node 2. An anchored conceptual component may be instantiated only on the node to which it is anchored; this is discussed further in section 5.4.1. The Value store 4 is left floating, which means that it may be instantiated on any available node.

In step (e) we create a call source 6 named “count” on the Clicker reactor and a call target 7 named “in” on the Value store. As discussed in section 4.3.5, port names are used to associate ports with internal features of components and are not required to be unique in the architecture. We also create a call connector 8 in workspace 1. This call connector is initially unattached.

Finally, in step (f) we attach the call connector 8 to ports 6 and 7. The evolution specifications in figures 4.12(a) and (b) ensure that the source end of the connector attaches to the source port and the target end of the connector attaches to the target port.

4.4 Conceptual level reflection operations

In order to request any of the operations specified in the preceding section, the user or component must be able to specify the parameters that appear in each operation’s signature (*e.g.*, component or connector identifiers). It is therefore a practi-

cal necessity that the system provide reflection operations allowing the discovery of workspaces, nodes, components, ports and connectors of interest. For example, if two users decide to work together on a document, at least one of them will need a mechanism for determining the identity of an attached synchronization connector that would support the required sharing.

For such operations to be useful, user or system provision of attributes like names of components and owners of workspaces will likely be essential. Reflection operations therefore appear to be a practical necessity for the effective use of evolution operations. A minimal list of the required reflection operations is given here.

In each definition below, the signature of each operation is provided. In the signatures, A is the set of architectures, e of workspace element identifiers, w of workspace identifiers, n of node identifiers, c of component identifiers, k of connector identifiers, p of port identifiers, "string" of character strings and "value" of arbitrary values.

- Given the identity of any workspace element (workspace, node, component, port, or connector), returns the names of its attributes.
  $getAttributes : A \times e \rightarrow \mathbb{P}(\text{string})$
- Given the identity of any workspace element and the name of one of its attributes, returns the appropriate intended or observed attribute value.
  $getIntendedValue : A \times e \times \text{string} \rightarrow \text{value}$
  $getObservedValue : A \times e \times \text{string} \rightarrow \text{value}$
- Returns the identities of all workspaces in the current architecture.
  $getWorkspaces : A \rightarrow \mathbb{P}(w)$
- Returns the identities of all synchronization connectors in the current architecture.
  $getSynchronizations : A \rightarrow \mathbb{P}(s)$
- Given the identity of a workspace, returns the identity of all nodes found within it.
  $getNodes : A \times w \rightarrow \mathbb{P}(n)$

- Given the identity of a workspace, returns the identity of all components found within it.

  $getComponents : A \times w \rightarrow \mathbb{P}(c)$
- Given the identity of a workspace, returns the identities of all connectors found within it.

  $getConnectors : A \times w \rightarrow \mathbb{P}(k)$
- Given the identity of a node, returns the identity of all components anchored to it.

  $getAnchored : A \times n \rightarrow \mathbb{P}(c)$
- Given the identity of a component, returns the identities of all its ports.

  $getPorts : A \times c \rightarrow \mathbb{P}(p)$
- Given the identity of a port, returns the identity of its providing component.

  $getProvidingComponent : A \times p \rightarrow c$
- Given the identity of a port, returns the identity of all attached connectors.

  $getAttachedConnectors : A \times p \rightarrow \mathbb{P}(k)$
- Given the identity of a connector, returns the identities of all attached ports.

  $getAttachedPorts : A \times k \rightarrow \mathbb{P}(p)$

Reflection operations are also required at the implementation level and to map between the implementation level and the conceptual level. These operations are defined in sections 5.3.4 and 5.4.7, respectively.

4.5 Summary

This completes the definition of the conceptual level of the Workspace Model. The definition of the implementation level and of the refinement relation between the two levels is found in the following chapter.

Chapter 5

Implementation Level and Refinements

Configurations of components, connectors and ports at the conceptual level are refined into corresponding configurations of lower level components, connectors and ports constituting an actual implementation, with the nodes identified in the previous chapter providing the bridge between the two levels. This chapter presents the implementation level and the rules making up the refinement relation.

The chapter is structured as follows. In section 5.1 we introduce the components, connectors and ports that make up the implementation level. This is followed in section 5.2 by a discussion of the special purpose infrastructure components used at the implementation level to realize the conceptual level semantics. The evolution calculus operations for the implementation level are defined in section 5.3. In section 5.4 we present the refinements that map from the conceptual level to the implementation level.

5.1 Implementation level model elements

The implementation level notation is summarised in figure 5.1. It includes one kind of component, two kinds of connectors and one kind of port (the *local port*), which has both source and target versions.

5.1.1 Implementation components

Conceptual level components are implemented by configurations of implementation level components and connectors. In diagrams, implementation level components are shown shaded to distinguish them from conceptual level components. (In hand-drawn diagrams, implementation level components may be identified by shading in a triangle at the upper left corner of the component.) At the implementation level

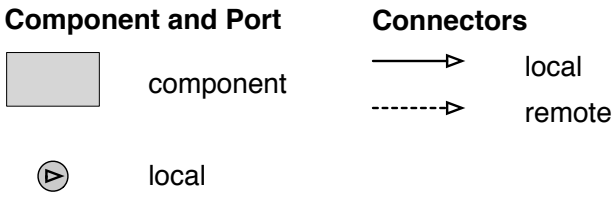


Figure 5.1: Implementation level notation.

there is only one kind of component. It may be either passive or active and may provide local source and target ports.

In analysis diagrams it is occasionally useful to annotate implementation level components as representing actors, reactors or stores using the symbols from the conceptual level. Such annotation, while allowed, does not form part of the formal model.

5.1.2 Connectors

Conceptual level connectors are implemented by configurations of implementation level connectors and components. The two implementation level connector types are:

Local. A local connector enables local procedure calls or method invocations as defined in most imperative programming languages. The call may include parameters, may produce a return value, and transfers the thread of control to the called component (*i.e.*, the calling component blocks until the call returns). A single local connector is sufficient to implement a call. The call is made in the direction of the arrow and the return value goes in the opposite direction.

Remote. A remote connector provides inter-nodal messaging. Remote connectors provide for a request-reply protocol; that is, a reply may be returned on the same connector on which the request was sent. However, senders need not block on replies. Where a request-reply protocol is desired, *e.g.*, for implementing remote calls, this must be implemented by the components at either

end of the connector. (See the transmitter and receiver components in section 5.2.1.) A remote connector may be used in place of a local connector, *i.e.*, on a single node; however, this would normally have a negative performance impact.

5.1.3 Ports

The implementation level provides only local ports, used with local connectors. (Remote connectors are attached only between transmitters and receivers; see section 5.2.1.) Local ports have source and target versions. As at the conceptual level, a source port is shown with its arrow pointing out of the host component and a target port is shown with its arrow pointing into the target component.

An implementation source port is responsible for ensuring that all parameters of procedure calls or method invocations which traverse it are passed by value, as discussed in section 4.2.3.2. Similarly, implementation target ports must ensure that all return values are passed by value. This is to ensure that no direct references to the internal state of components are ever passed on local connectors.

Connections between infrastructure components (described in section 5.2) are direct and do not require ports.

As with conceptual level ports, implementation level ports are named. This allows for mapping between a port and some construct visible within the the providing component. For source ports this is normally a syntactic variable or an element of a list or array. A target port may correspond to a callable interface on the containing component.

5.1.4 Relations among implementation level elements

As at the conceptual level, implementation level elements may be related to one another in several ways, which are depicted in workspace diagrams. The relations and diagrammatic conventions are described below.

Note that the sets of implementation level workspace elements are identified

by capital letters to distinguish them from the sets of conceptual level workspace elements.

Instantiation. Implementation level components are instantiated on nodes. If an implementation level component C is shown superimposed on a node n, then C is instantiated on n.

Port Provision. Implementation level components provide ports. If a port P is depicted on the boundary of a component C, then P is provided by C.

Connection. Implementation level connectors connect ports and infrastructure components. If the end of a connector K is shown touching two ports P_1 and P_2, then K connects P_1 and P_2 (and the same for infrastructure components). For visual clarity we frequently omit ports in workspace diagrams. If a connector K is shown touching two components C_1 and C_2 then K connects two unseen ports P_1 provided by C_1 and P_2 provided by C_2. The exception is where a connector is shown touching an infrastructure component, since these have no ports.

5.2 Infrastructure components

The implementation level of the Workspace Model includes a number of components with specialised functions, which are used in the implementation of conceptual level constructs. These are referred to as *infrastructure components* and must be included in any implementation of the Workspace Model. The infrastructure components are illustrated in figure 5.2 and defined in the remainder of this section.

5.2.1 Transmitter and receiver

Transmitter-receiver pairs provide bridges between local connectors and remote connectors. Where a conceptual level connector must be implemented across node boundaries, these pairs will be inserted into the connector implementation to provide the required communication means.

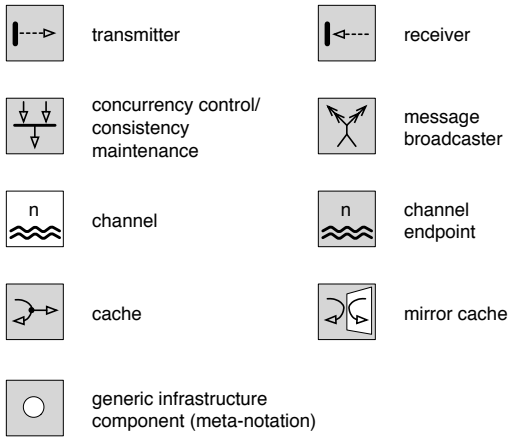


Figure 5.2: Implementation level infrastructure components.

Transmitter-receiver pairs are used to implement two-way reliable messaging using a request-reply protocol. An invocation on a transmitter by a local connector will block while the request-reply protocol completes.

A transmitter may be attached to one outgoing remote connector. A receiver may accept any number of incoming remote connectors.

5.2.2 Concurrency control and consistency maintenance

The simplest workspace components are coded for use in a single-threaded environment with no replication, however they are normally used in a multi-threaded environment and stores may be replicated in support of synchronization. The concurrency control and consistency maintenance (CCCM) component is responsible for resolving concurrency control issues and for maintaining replica consistency.

CCCM components implement the concurrency control policies necessary for the use of single threaded components in a multi-threaded environment. They also provide for synchronization of replica stores and for cache invalidation (see section 5.2.5).

CCCMs may implement a range of concurrency control policies (*e.g.*, unconstrained, simple locking, transactional locking, optimistic, *etc.*) and a range of consistency maintenance algorithms (locking, two phase commit, distributed op-

eration transform [49], ORESTE [69], *etc.*). Concurrency control algorithms and consistency maintenance policies are not explicitly represented in the Workspace Model; Workspace implementations are responsible for ensuring that only mutually-compatible CCCM implementations are connected for consistency maintenance purposes.

5.2.3 Message broadcaster

Message broadcasters are used in the implementation of subscription connectors and in centralized implementations of channels (see below). They accept messages as inputs and broadcast them to one or more subscribers. A message broadcaster provides messages to each of its outputs in the same order they were received on its inputs; ordering is the same on all outputs.

A message broadcaster is an actor and delivers messages in its own thread or threads of control. It therefore acts as a bridge between synchronous calls on its input side and asynchronous message delivery on its output side.

5.2.4 Channel and channel endpoint

In implementing groupware it is frequently necessary for a group of components to communicate with one another. While it is possible to connect n components using $n(n-1)$ transmitter-receiver pairs, it is normally simpler to consider these communication paths as channels [15], particularly for large values of n. Channels are asynchronous. Messages sent into a channel at any one of its endpoints are received at all endpoints, including the sender.

Figure 5.2 shows conceptual level channels and implementation level channel endpoints. The conceptual level channels appear only in the refinements of subscription connectors and synchronization groups specified in section 5.4 and are not used directly at the conceptual level. Channels may be refined using a set of distributed channel endpoints or by making use of a centralized message broadcaster to provide channel ordering. In the special case of a channel whose only source

is a single message broadcaster, a channel may be implemented simply as a call connector. See section 5.4.6 for channel refinements.

Channels and channel endpoints are identified by a channel number, shown as n in figure 5.2. In workspace diagrams, any two channel endpoints with the same channel number represent endpoints on the same channel. While not part of the formal model, it is frequently useful to draw lines between endpoints on a channel to visually indicate their connection. This is normally done using a wide, lightweight line; see figure 3.4 on page 56 for an example. In hand drawings another convention such as a wiggly line or a special colour may be used.

As a minimum, channels must implement local FIFO ordering of messages. That is, messages originating at a given source must be delivered to all targets in the order sent. Messages originating at different sources may be interleaved differently at different targets.

For some applications, causal or total ordering may be required. Where this is the case, it may be necessary to specify a particular channel implementation through an implementation hint. Some consistency maintenance algorithms require particular ordering policies on their communications channels; the correct match of CCCM and channel implementations must be provided by the runtime system.

5.2.5 Cache and mirror cache

Where deterministic pure requests are implemented using remote connectors, we may use caches to reduce latency by eliminating unnecessary communication on the network. Caching is implemented using cache and mirror cache components.

Cache. A cache will be found at the source end of the call. Its function is to store the responses to requests. Caches may receive, and are responsible for acting on, cache invalidation messages from mirror caches.

Caches may be either *simple* or *prefetch*. On receipt of a cache invalidation message a simple cache will discard any corresponding cache entries; however, a prefetch cache may initiate requests to determine appropriate new

values for those entries.

Mirror cache. Also called cache controller. A mirror cache will be found at the target end of a call. Its role is to keep track of what entries the corresponding cache is currently maintaining and to invalidate or update the cache as necessary.

The mirror cache may receive, and is responsible for acting on, cache invalidation messages from its connected CCCM when the state of the target changes. It must pass these on to its corresponding cache as appropriate.

Mirror caches may be *simple* or *presend*. Based on the messages received from the CCCM component the mirror cache computes what invalid entries its corresponding cache is holding and then either transmits the minimum invalidation messages (simple cache), or computes new values by making requests of the target component and transmits update messages (presend cache).

Pure updates have no return values and therefore need not be cached. Request-updates are assumed to modify the call target and so cannot reliably be cached. At least some pure requests on non-deterministic components may not be cacheable either. Since a request-update connector may contain both requests and updates it may be necessary to provide selective caching in such cases. The architectural employment of cache and mirror cache components is defined in section 5.4.3.

5.2.6 Generic infrastructure component

The generic infrastructure component is not truly a component, but rather an extension to the meta-notation introduced in section 4.3.1. It is used to represent “any infrastructure component” in figure 5.5(b).

5.3 Implementation level evolution calculus

Implementation level operations are defined by an evolution calculus in the same manner as for the conceptual level operations. The semantics of the implementa-

tion level calculus operations are specified in this section, using the meta-notation introduced in figure 4.5 on page 79.

5.3.1 Components

The allowed operations on implementation level components are defined in figure 5.3.

Figure 5.3 (a). A component may be instantiated on a node. Newly instantiated components are not connected to other components.

Figure 5.3 (b). A component may be destroyed. In contrast to the conceptual level, where a component must first have all attached connectors removed before it is destroyed, destroying an implementation level component also destroys all attached connectors. This is because implementation level connectors are not truly first-class workspace objects. See section 5.1.2 for further discussion. Note that in this diagram connectors are destroyed whether they are attached via ports or directly to the destroyed component.

Figure 5.3 (c). A component may be moved from its current node to another node. All attached connectors will be destroyed by the move. The component's state will be observationally equivalent after the move (this is not explicitly represented on the diagram.) An attempt to move a component onto the node on which it is currently instantiated will have no effect; such an attempt will fail to match the left-hand side of this diagram.

Figure 5.3 (d) and (e). Components may be copied, either onto a new node (d) or onto the node on which the original is instantiated (e). As with moves, a copy of a component must be observationally equivalent to the original. Initially, a copied component will be unconnected. The original component is unaffected.

5.3.2 Ports

Figure 5.4 (a) and (b). Local source and target ports may be created on component boundaries. Created ports are initially unconnected. All ports must be named.

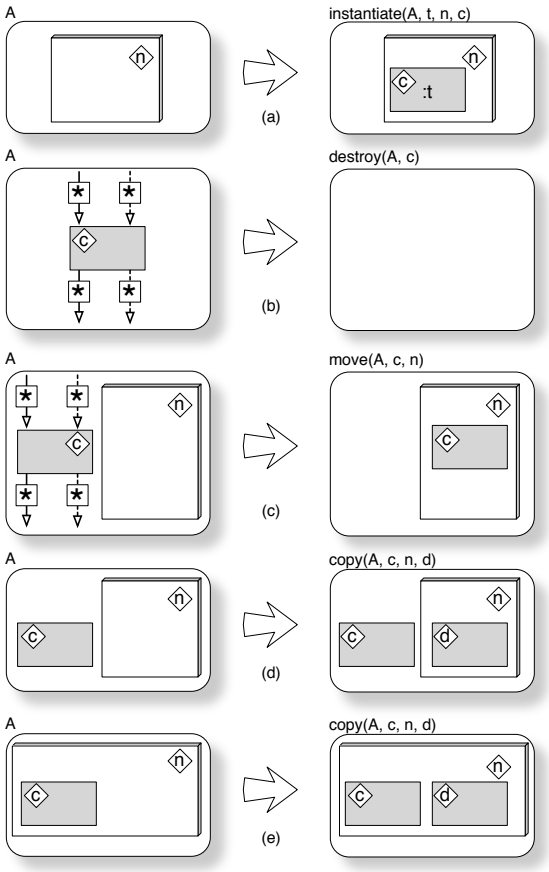


Figure 5.3: Operations on implementation level components.

Figure 5.4 (c) and (d). Local source and target ports may be destroyed. At the implementation level, connectors are second-class objects and may not exist independently of ports. Therefore destroying an implementation level port also destroys any connectors for which the port was either a source or a target.

5.3.3 Connect and disconnect

Figure 5.5 (a). An unconnected local source port may be connected to any implementation level port or component on the same node, by a new local connector. Local source ports may be the source of not more than one connector.

Figure 5.5 (b). An infrastructure component may be connected by local connector to any implementation level port or component on the same node. The infrastructure component may act as the source of any number of local connectors.

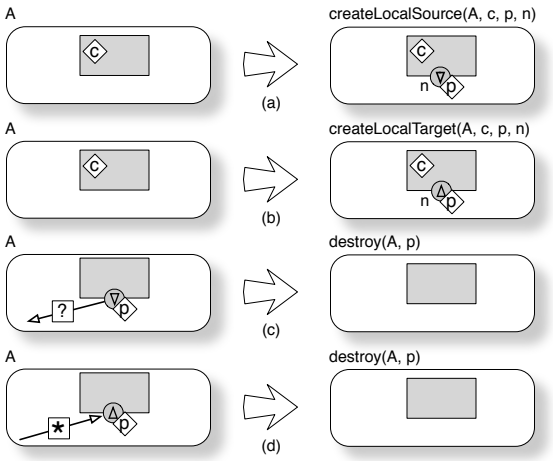


Figure 5.4: Operations on implementation level ports.

Figure 5.5 (c). A transmitter that is not currently the source of a remote connector may be remotely connected to a receiver. Receivers may accept any number of incoming connectors. The transmitter and receiver may be on the same or different nodes.

Figure 5.5 (d) and (e). Two connected elements may be disconnected. This action implicitly destroys the connector.

5.3.4 Implementation level reflection operations

The implementation level requires a set of reflection operations similar to those for the conceptual level introduced in section 4.4 on page 91. As in that section, we here describe a minimum set.

As noted in section 4.2.7, upper case letters are used to identify the sets of implementation-level workspace element identifiers in the signatures that follow.

The conceptual level reflection operations $getAttributes$, $getIntendedValue$, $getObservedValue$, $getPorts$ and $getProvidingComponent$ that are defined in section 4.4 are also defined at the implementation level, with the substitution of upper case letters for lower case letters in their signatures.

In addition, the following operations are unique to the implementation level:

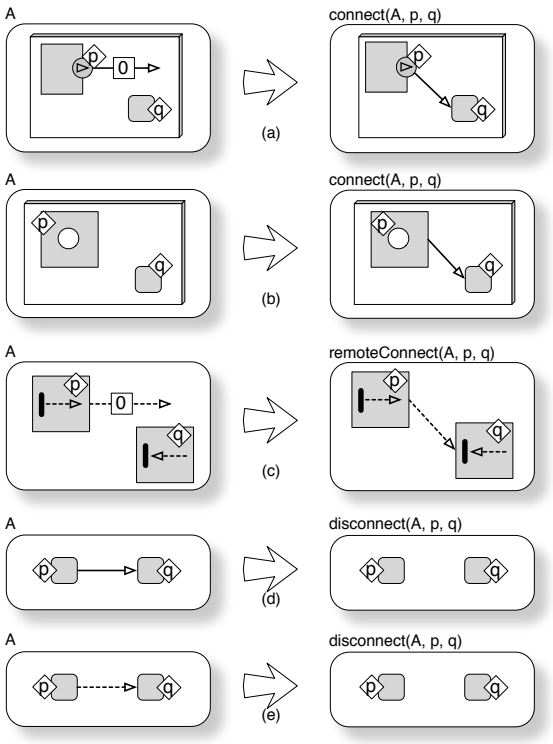


Figure 5.5: Implementation level port connection and disconnection.

- Return the identities of all implementation level components.

  $getImplComponents : A \rightarrow \mathbb{P}(C)$

- Given the identity of a node, return the identities of all components instantiated on it.

  $getInstantiated : A \times n \rightarrow \mathbb{P}(C)$

- Given the identity of a component or port, return all connected components or ports.

  $getConnected : A \times (P \cup C) \rightarrow \mathbb{P}(P \cup C)$

5.4 Refinements from the conceptual to the implementation level

A key concept in the Workspace Model is that conceptual level architectures are realized by corresponding implementation level architectures. By design, the model provides multiple possible implementations for any but the simplest conceptual level architectures. The process of deriving an implementation level architecture from a

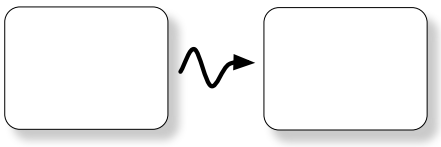


Figure 5.6: The schema used for refinement rules.

conceptual level architecture is called “refinement” and is the subject of this section.

The allowed refinements are presented in the form of a graph grammar consisting of pattern matches and replacement patterns. This grammar defines the total space of allowable implementations for any given conceptual level architecture.

In principle, for a conceptual level architecture A we begin by finding a match m between A’s structure and the left-hand side of some refinement rule. We then replace m by the implementation specified on the right-hand side of that rule. This gives us a partially-refined architecture A'. We repeat the process until no further rule matches are possible.

The refinement rules use the meta-notation and pattern match rules introduced in section 4.3.1 on page 78. Refinements rules are written in the refinement schema illustrated in figure 5.6. The squiggly arrow means “may be refined to” or “may be implemented as”.

As in the operation definitions, if the left-hand pattern of a refinement rule can be matched for an architecture A', then the rule operates by transforming A resulting in an A' that differs from A in exactly the same ways that the left-hand side of the rule differs from the right-hand side. Differences may include the presence or absence of workspace elements as well as alterations in any of the Workspace relations. Any workspace elements or relations in A that are not explicitly depicted in the left-hand side of the rule are unchanged in A'. The one exception is for operations which destroy components: if a component is destroyed, it is removed from any relationships in which it previously participated even if that relationship is not explicitly shown on the diagram.

The refinement rules have been designed to meet the toolkit implementor requirements of section 3.1.3. In chapter 8 we prove that the rules are implementable

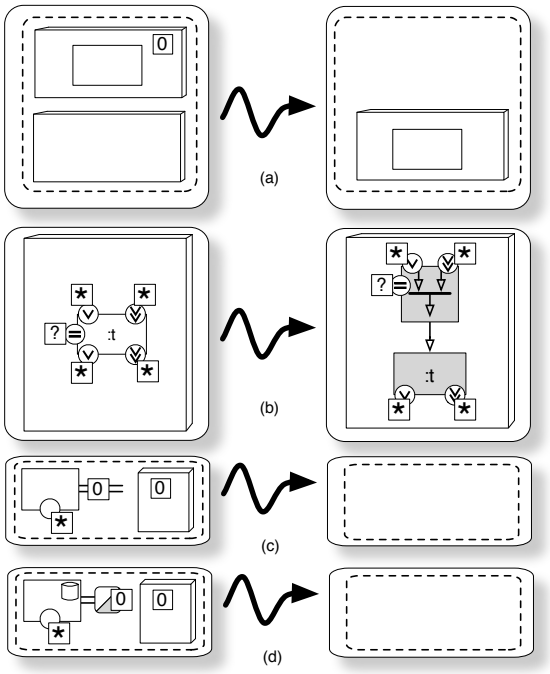


Figure 5.7: Refinements of components.

in that they tend monotonically towards fully refined architectures and the computation of the refinement relation is provably terminating.

Each of the sections that follow has one or more associated figures defining a set of related refinement rules. For greater clarity, each refinement rule is also described in the section's text. The operation of the refinements is illustrated by a small example in section 5.4.8.

5.4.1 Components

The main refinements for components are illustrated in figure 5.7. Stores which are part of synchronization groups may also be implemented according to the refinements presented in section 5.4.5.

Figure 5.7(a). A conceptual level component that is not anchored to any node may be anchored to any node in its workspace. (Note that this is anchoring as part of the refinement process; it is not the conceptual level anchoring evolution specified in figure 4.10(a).)

Figure 5.7(b). A conceptual level component of type t that is anchored to a node may be implemented on that node. The implementation will include an implementation level component of type t plus a CCCM. Any synchronization port and all incoming call and subscription ports appear on the CCCM component and any outgoing ports appear on the implementation component of type t.

Note that the refinement in figure 5.7(b) does not apply to conceptual channels, which are not true conceptual components and have no type t. The refinement of conceptual channels is defined in section 5.4.6.

Figure 5.7(c) and (d). Components which have no semantically valid implementation may be removed from the architecture, along with all their ports. Actors, reactors and stores that are in workspaces with no nodes may not be implemented unless they are members of a subscription group that includes at least one other member. Figure 5.7(c) will match any conceptual level component that is not attached to a synchronization connector; figure 5.7(d) will match stores that are the sole member of a synchronization group. Recall that actors and reactors may not have synchronization ports. (Stores that are members of synchronization groups with at least one other member may be refined by the rule in figure 5.11(a).)

5.4.2 Ports

Figure 5.8(a) and (b). Call ports refine to single local ports, either source or target as appropriate. Any connectors which were attached to the call port are attached to the new local port. The name of the local port is the same as the name of the call port.

Figure 5.8(c). Subscription target ports refine to single local target ports. Any connectors that were attached to the subscription port are attached to the new local port. The name of the local port is the same as the name of the subscription port.

Figure 5.8(d). A subscription source port refines to a local source port that is connected to a message broadcaster by a local connector. The message broadcaster is instantiated on the same node as the component providing the refined port. Any connectors that were attached to the subscription source are attached to the mes-

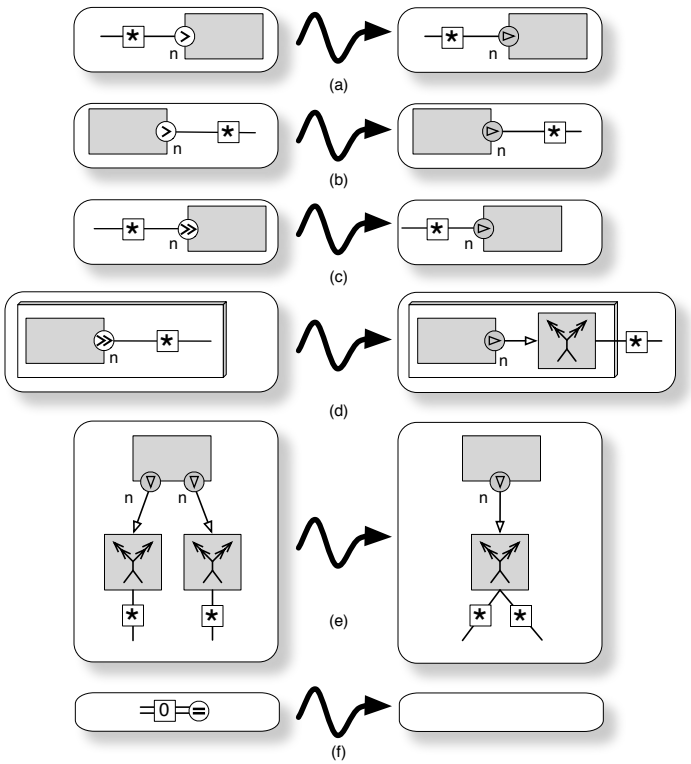


Figure 5.8: Refinements of ports

---

sage broadcaster. The name of the local source port is the same as the name of the subscription port.

Figure 5.8(e). Two subscription source refinements having the same name may be merged. Any connectors which were attached to the message broadcasters in the subscription source refinements are attached to the remaining message broadcaster. This refinement is used to merge the multiple, identically named subscription sources on a component that may result from the refinement in figure 5.11(a).

Figure 5.8(f). Synchronization ports that are not attached to synchronization connectors require no refinement and may be removed from the architecture.

The refinements for synchronizations (see section 5.4.5) convert synchronization ports to pairs of local ports, so no other synchronization port refinements are required.

5.4.3 Calls

Figure 5.9(a). Calls between components or ports on the same node may be implemented directly by local connectors.

Figure 5.9(b and c). Calls between components and ports may be implemented by inserting transmitter-receiver pairs in the call path to mediate the inter-nodal communication. This is possible for components and ports on the same node and on different nodes; however, there may be an unnecessary performance penalty incurred by this approach where the ports are in fact on the same node.

Figure 5.9(d). Call paths may include caches in order to reduce unnecessary network latency. The intended operation of a cached call path is as follows:

- When a request is initiated at the call source connector, the call goes first to the cache. If a result value corresponding to the request is cached, the value is returned immediately.
- If the result is not currently cached, the call proceeds through the transmitter-receiver pair, mirror cache, CCCM, and ultimately to the target component.
- The result is returned via the reverse path. The mirror cache, which is on the same node as the target, caches a copy of the request-result pair, as does the cache.
- Any updates to the target component are detected by the CCCM. The CCCM computes a conservative characterisation of what cached results would be invalidated by the update. For more precise characterisations, the application programmer could provide a specialised CCCM. However, in the worst case, the CCCM can conservatively declare all cached results from that component to be invalid.
- The CCCM calls the mirror cache via its outgoing local connector to inform it of the cache invalidation. The mirror cache then determines exactly which of the currently-cached entries are invalid and both removes them from its cache mirror and advises the cache of the invalidation by means of the update connector.

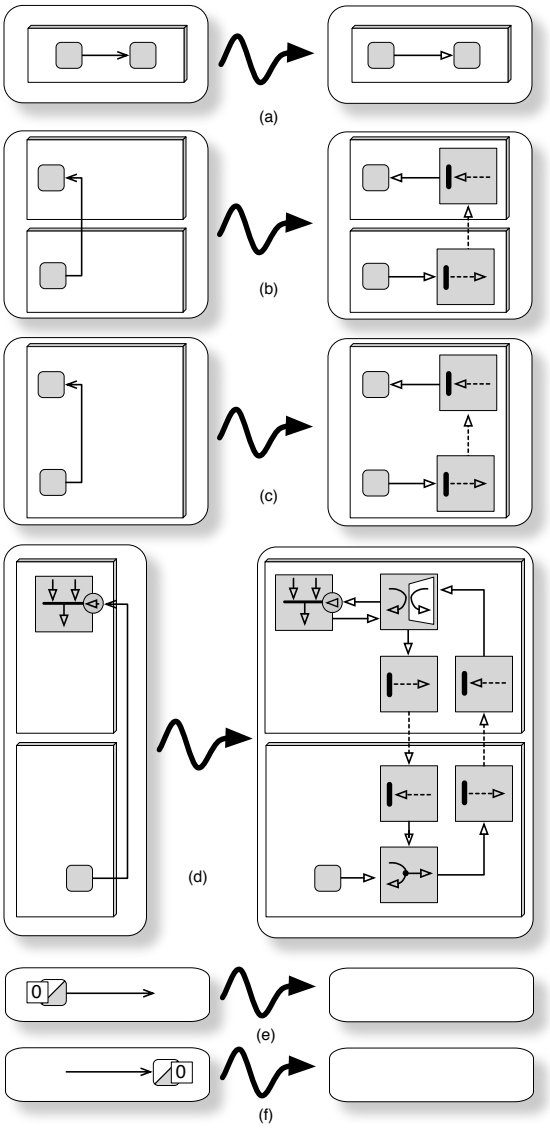


Figure 5.9: Refinements for call connectors.

Note that if a conceptual level component has many incoming call connectors, its implementation may have many attached mirror caches. All mirror caches must be notified of all cache invalidations.

Pure updates are not cached since they return no result. Request-updates are likewise not cached, since the result returned by a request-update is not necessarily valid after the update completes. Finally, requests made of actors cannot normally be cached because of actors' inherent non-determinism, unless the developer explicitly declares that a request on an actor is cacheable.

5.4.4 Subscriptions

5.4.4.1 People and subscriptions.

At the conceptual level, people may act as the source and target of subscription connectors. These connectors represent people's provision of input using (for example) voice, keyboards and mice, as well as people's attention to audible, visible or other signals. Hence, these connections are inherently conceptual.

For this reason, subscription connections to and from people are *not* refined and are *not* the subject of the refinement rules in this section.

5.4.4.2 Refinable subscription connections.

The asynchronous semantics of subscriptions is provided by the message broadcasters which form part of the implementation of subscription source ports (see figure 5.8(d)).

The multi-point to multi-point nature of subscriptions is provided by channels. The refinement of a subscription connector into a channel is defined in figure 5.10. The refinement of channels is addressed in section 5.4.6.

Figure 5.10(a). Where two message broadcasters on the same node are both sources of a common subscription connector and not sources of any other connectors, the message broadcasters may be combined. This means that one of the two is removed

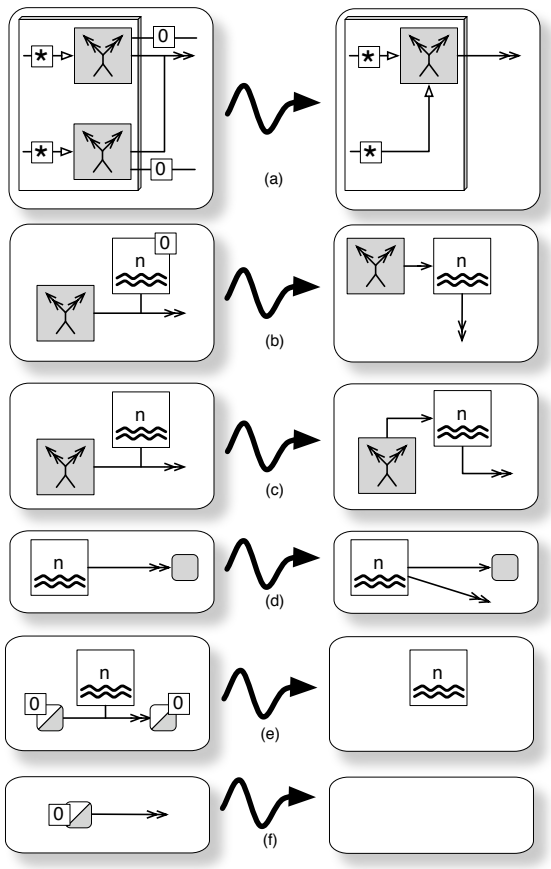


Figure 5.10: Refinements for subscriptions.

---

and the incoming connectors that were attached to the removed broadcaster are rerouted to the remaining broadcaster.

Figure 5.10(b). This refinement allows creation of a new channel. If a message broadcaster is a source of a subscription connector that has no attached channel, then a new channel may be created and the message broadcaster attached to that by a call connector. The message broadcaster is detached from the subscription connector, and channel is attached to the subscription connector as a source. The number of the new channel must be different from any other channel or channel endpoint in the architecture.

Figure 5.10(c). If a message broadcaster is a source of a subscription connector that does have a channel as a source, then the message broadcaster may be attached to the channel by a call connector. The message broadcaster is detached from the

subscription connector.

Figure 5.10(d). Where a channel is the source of a subscription connector, and that subscription connector has a target which is an implementation level port or component, the channel may be connected to that target by a call connector. The target is disconnected from the subscription connector.

Figure 5.10(e). Where a subscription connector has a channel for a source and has no other sources or targets at either the conceptual or implementation levels, that subscription connector may be refined away.

Figure 5.10(f). A subscription connector with no targets may be removed from the architecture.

5.4.5 Synchronization

The refinements for synchronization allow for any combination of centralized and replicated state within a synchronization group.

Figure 5.11(a). A store that is synchronized with an already-implemented component (see section 5.4.1) may be refined by simply moving its subscription and call ports to the implemented component. As in component refinement, call and subscription target ports are attached to the CCCM component and subscription source ports are attached to the component implementation. (Recall that stores may not have call source ports.)

In effect, this is a centralized implementation, since it refines two or more conceptual level stores into a single store implementation.

Figure 5.11(b). Where a CCCM component is attached to a synchronization group, and that group has no attached channel, a new channel may be created and attached to the synchronization group. As in figure 5.10(b), the new channel must have a unique number not currently in use in the architecture.

Figure 5.11(c). Where a CCCM component is attached to a synchronization group and that group has an attached channel, the attachment may be refined to local source and local target connectors attached the the channel via call connectors.

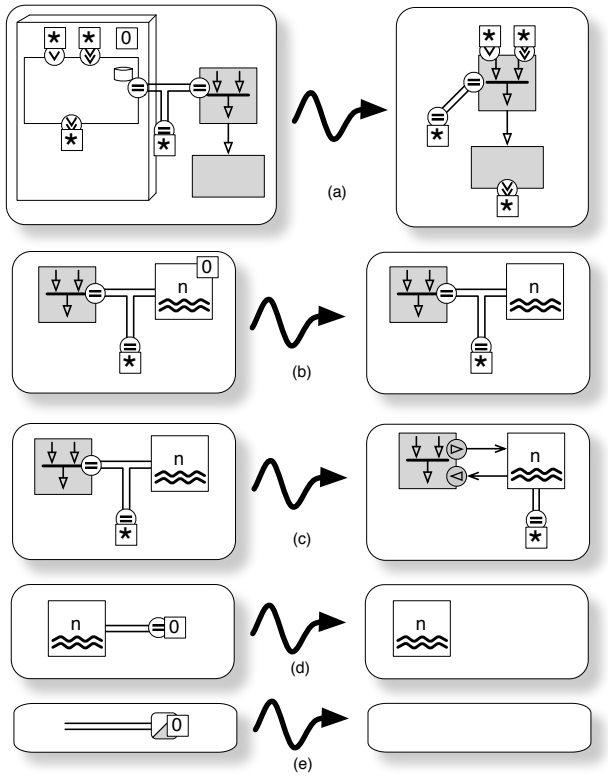


Figure 5.11: Refinements for synchronizations.

In effect, figures 5.11(b) and (c) allow for a replicated implementation of synchronization. The CCCMs are required to enact some replica consistency maintenance protocol, communicating with one another via the channel. Where a synchronization group has three or more end points, a mix of centralised and replicated implementations is possible.

Figure 5.11(d). A synchronization group attached to a channel that has no other attached synchronization ports may be refined away.

5.4.6 Channels

In figures 5.10 and 5.11, conceptual level channels were introduced to handle the n-way communication required for the implementation of subscriptions and of replicated synchronization groups. Figures 5.12 and 5.13 and this section present the refinements that transform conceptual level channels to channel implementations.

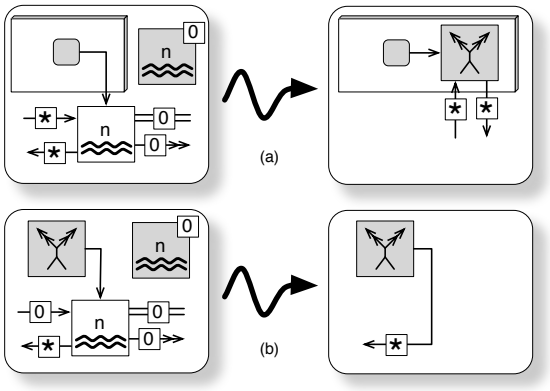


Figure 5.12: Centralised refinements for channels.

Figure 5.12(a) This refinement applies to a channel that has an arbitrary number of incoming and outgoing call connectors, no attached subscription or synchronization connectors, and that has not yet been partially refined to include any channel endpoints. In this case, the channel may be refined by replacing the channel by a message broadcaster, which provides the required asynchronous channel semantics and the strongest ordering guarantee. The message broadcaster is instantiated on the same node as one of the incoming call connector sources.

Figure 5.12(b) This refinement applies to a case similar to that of figure 5.12(a), except that there may be only one call connector incoming to the channel and its source must be a message broadcaster. In this case each outgoing call connector may simply be attached to the message broadcaster.

Figure 5.13(a) and (b). A call connection from a component c to a conceptual level channel designated n may be refined to a connection to a channel implementation for n, where the new channel implementation is instantiated on the same node as c. The conceptual level channel is otherwise unaffected.

Figure 5.13(c) and (d). If a component c is connected to a conceptual level channel designated n and a channel implementation for n already exists, then the connection from c to the channel may be refined as a connection from c to the channel implementation.

The refinements in figure 5.13 allow connected channels to be implemented us-

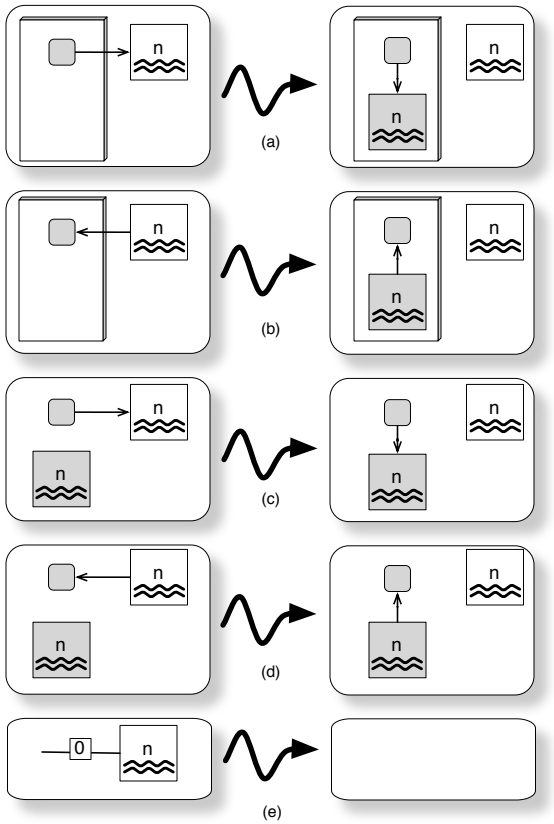


Figure 5.13: Distributed refinements for channels.

ing any number of channel implementations between one and the number of connections from components to the channel. So, a fully centralised implementation (where all components are connected to a central channel implementation) is allowed, as is a fully distributed implementation.

5.4.7 Inter-level reflection operations

Supporting dynamic evolution at runtime requires a mechanism for relating the current conceptual level architecture to the current implementation level architecture. The following reflection operations are therefore provided:

- Given the identity of a conceptual level element, return the identities of all implementation-level elements that form part of its implementation.
  $getImplementation : A \times e \rightarrow \mathbb{P}(E)$

- Given the identify of an implementation level element, return the identities of all conceptual level elements that it implements.
  $getConceptual : A \times E \rightarrow \mathbb{P}(e)$

5.4.8 An example refinement

The application of refinement rules is illustrated by the example refinement sequence shown in figure 5.14, which is read left to right, top to bottom.

Figure 5.14(a) shows an initial conceptual level architecture and figure 5.14(f) shows its final refinement. The other figures show partially-refined architectures that include both conceptual and implementation level elements. The text to the right of each architecture diagram indicates the refinement rule applied to transform it into the subsequent architecture. The elements that the rule matches are shown with dark outlines for ease of identification.

In figure 5.14(a) we can match rule 5.7(a), which states that an unanchored component may be anchored to any node in its workspace. The result of this refinement is shown in figure 5.14(b).

Figures 5.14(b) and (c) illustrate component instantiation. In (b), the Value component and the node it is anchored to are matched by rule 5.7(b), resulting in the conceptual level Value's replacement by a CCCM, an implementation level Value, and a local connector. In (c) a similar transformation is performed on the Clicker component. As specified by the transformation rule, the Value's call target port is hosted on its implementation's CCCM while the Clicker's call source port is hosted on the Clicker implementation component.

In figure 5.14(d) the call source and target ports can be matched by refinement rules 5.8(b) and (a), respectively, which transform them into local source and target ports. The ports' names remain unchanged.

Finally, in figure 5.14(e) the call connector, its source and target, and the nodes the source and target are on match rule 5.9(b), which replaces the call by a transmitter-receiver pair, two local connectors and a remote connector. The final, fully refined architecture is shown in figure 5.14(f).

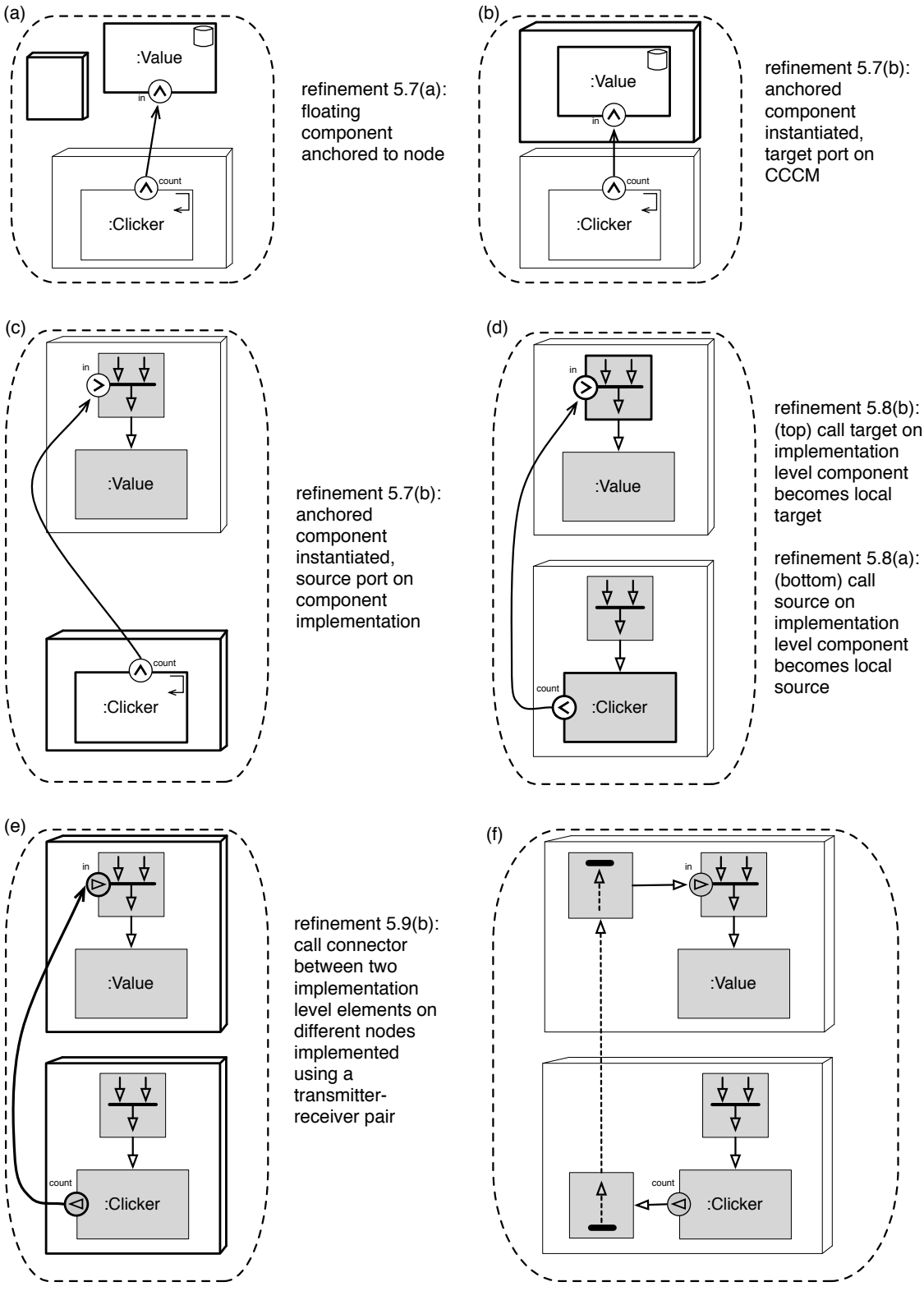


Figure 5.14: An example of the application of refinement rules.

Recall that refinements may be applied in any order and that any matching refinement may be applied at any time. So, for example in the architecture shown in figure 5.14(a) we could have matched the same refinement chosen (5.7(a)) using the other node; we could also have matched refinement 5.7(b) to the Clicker component and its node before anchoring the Value component. The number of possible refinements expands significantly for more complex conceptual level architectures, particularly those including subscription connectors or synchronization groups. Some examples of alternate refinements for a slightly more complex conceptal level architecture (that shown in figure 6.5 on page 134) are shown in figures 1.4, 6.6, 6.7 and 6.8 on pages 15, 136, 137 and 138.

5.5 Summary

This concludes the definition of the Workspace Model. In the next chapter we describe how the model may be used for scenario-based analysis, component-level design, architectural analysis of recovery and restoral from partial failure, and as a programming framework.

Chapter 6

Applying the Workspace Model

In this chapter we examine how the Workspace Model can be applied to a range of tasks that occur in the development of groupware systems.

Scenario-based modeling is a technique that is often used in the requirements analysis and early design phases of interactive software development, and includes a rich body of notations and practices [23, 113]. We illustrate how the Workspace Model can be used to supplement these in order to provide an early architecture-based model of groupware systems. The Workspace Model's inclusion of high-level, groupware-appropriate abstractions like separate contexts of work, shared information, and communication via asynchronous message streams makes it well suited to this task.

We then demonstrate how the Workspace Model can be applied to the component level design of a groupware system, in terms of determining what components the system should consist of and what their patterns of communication should be.

The Workspace Model's clean separation of conceptual and implementation levels also makes it useful for the analysis of "non-functional" run time properties such as fault tolerance, security, and performance. In section 6.3 we provide a detailed example illustrating how the model can be used to reason about service restoral and recovery from partial failure.

Finally, in section 6.4 we illustrate how the Workspace Model may be directly applied to the problem of programming groupware systems. This is presented in terms of the programming interface provided by our fiia toolkit, by means of a small sample program.

6.1 Scenario-based modelling

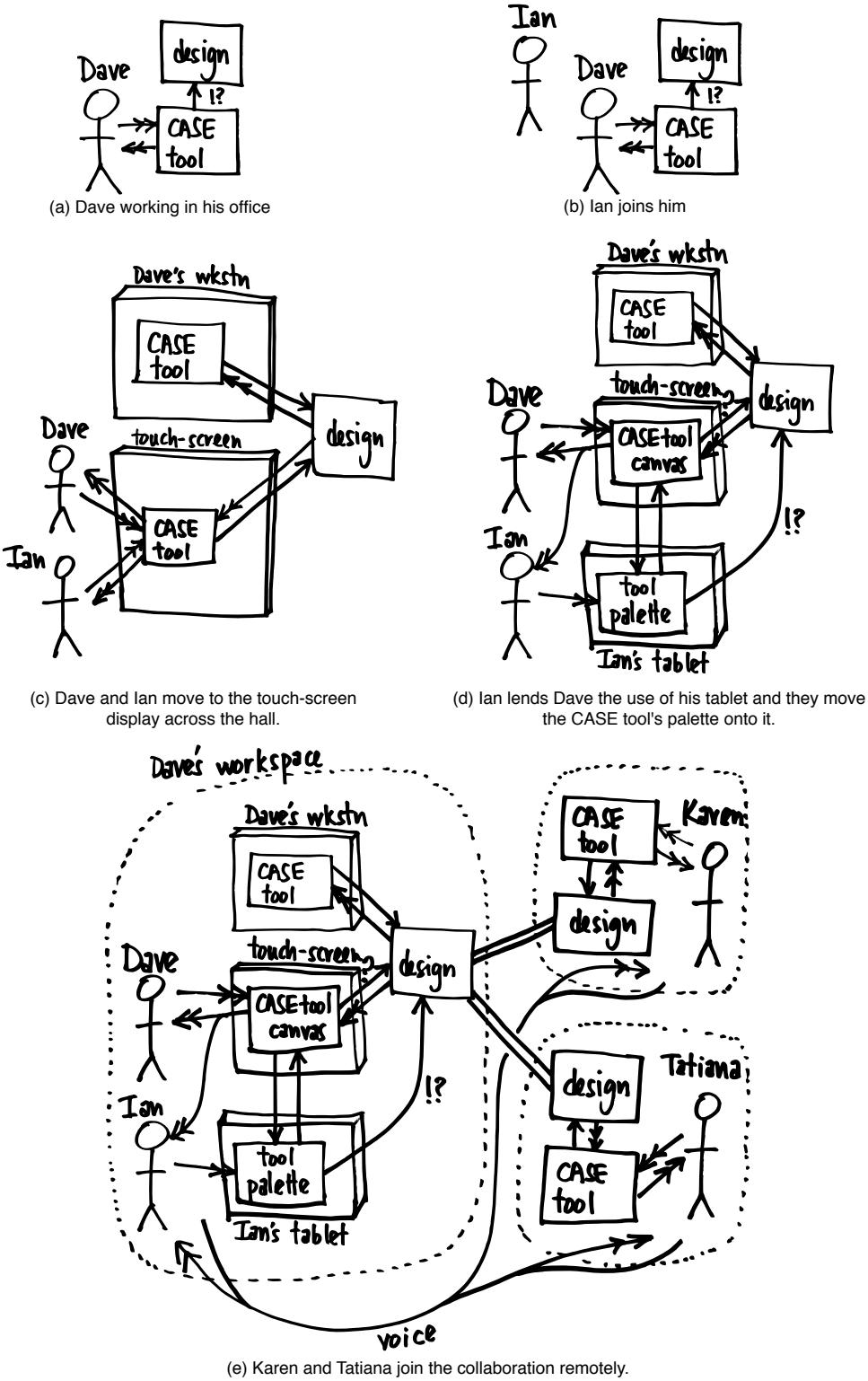
Scenario-based analysis and modelling is frequently used to support the requirements definition and early design of software systems, particularly interactive systems [23, 113]. As practised by Carroll and others, scenarios are typically modelled as narrative text, with supporting pictures or diagrams where useful. The pictures normally document possible user interface designs and relationships between users and the systems under consideration. Scenarios may be used to understand the behaviour of current systems or to assist in envisioning the behaviour of desired systems.

The Workspace Model provides a mechanism to extend classic scenario-based design by providing a lightweight representation of possible architectures for groupware (or single user) systems, in their contexts of use. In this section we return to the CASE tool scenario initially presented in section 1.1.1 and show how the Workspace Model can be used to model the scenario informally and to draw architectural implications from it.

Figure 6.1 illustrates the CASE tool scenario as a series of Workspace diagrams, each of which represents a point-in-time snapshot of a possible system architecture. The designs grow progressively more detailed as the scenario progresses and more decisions are made.

In figure 6.1(a) we see Dave working in his office, using his CASE tool's user interface to interact with a design. This diagram, and the rest in the figure, are very informal and represent the kinds of things that designers might draw in a whiteboard session. For instance, in this diagram the types of the components (actor, reactor, or store) are not shown and there are no nodes or workspaces depicted.

The next step in the scenario is Ian walking into Dave's office, depicted by figure 6.1(b). The fact that there are no arrows between the CASE tool and Ian represent the fact that Dave's office is not well laid out for collaborative brainstorming. This is why Ian and Dave move across to the large touch-screen, as shown in figure 6.1(c).



(a) Dave working in his office

(b) Ian joins him

(c) Dave and Ian move to the touch-screen display across the hall.

(d) Ian lends Dave the use of his tablet and they move the CASE tool's palette onto it.

(e) Karen and Tatiana join the collaboration remotely.

Figure 6.1: The scenario from section 1.1.1, represented as a series of informal Workspace diagrams.

Ian now has arrows to and from the CASE tool on the touch-screen, indicating that he has the ability to participate actively in the collaboration. At this point it also becomes interesting to explicitly represent nodes; this allows us to illustrate the fact that Dave left his CASE tool running on his workstation and has another interface, connected to the same design, on the touch-screen. We have left the design component floating to indicate that the actual computer it is instantiated on is unimportant. The design component now has subscription arrows leaving it, indicating that if either of its two connected interfaces alter it, it has the ability to notify the other.

In figure 6.1(d) Ian has allowed Dave the use of his tablet computer, and Dave has moved the CASE tool's palette off the touch-screen and onto it. In representing this we have had to add some detail to the CASE tool's design, splitting its interface into a palette and a canvas and indicating the communication links between them. We have not bothered to elaborate the design of the CASE tool on Dave's workstation. The diagram shows that Dave is working on the canvas at the touch-screen and Ian is manipulating the tool palette on the tablet.

Finally, figure 6.1(e) depicts the scenario after Karen and Tatiana have remotely joined the collaboration. Workspaces are shown to indicate distinct work contexts. Each workspace contains a design component and the three designs are synchronized, indicating that the participants at all three locations are working with the same design. A subscription connector annotated "voice" has also been added to the diagram, indicating that the participants can talk to one another. This final sketched diagram is an informal version of figure 3.1 on page 45.

As seen in diagrams of figure 6.1, the Workspace Model can be used to concisely capture architectural snapshots of interesting systems. It clearly represents essential groupware concepts including distinct contexts of use, data sharing between session participants, where user interfaces are instantiated, and where stream-based communication is intended to occur. The Workspace Model's visual notation has been specifically designed to allow for partially-complete sketches, so if some decisions have not yet been made, details may be omitted where appropriate and

easily added later.[1]

In the next section we take a particular aspect of this scenario and explain how we can use it to drive the component level design of a groupware system.

6.2 Component design

Towards the end of the scenario in section 1.1.1, Dave, Ian, Tatiana and Karen are all working together on the design. "At one point Karen needs to check something on an overview diagram while Dave, Ian and Tatiana are discussing a more detailed one, so she temporarily decouples her view from the group's. After finding the information she needs, she rejoins the group view...."

6.2.1 Structural view

This scenario fragment implies that the design is made up of a number of different diagrams, that it is possible for the collaborators to ensure that they are always looking at the same diagram (the "group view") and that it is possible to decouple from, and later rejoin, the group view.

In the Workspace Model, this suggests that the concepts of "the design" and "the current diagram" should be represented as separate stores that can be independently shared. One store would hold all the diagrams that constitute the design; the other would hold an index into the design, perhaps as simple as a single integer, indicating which diagram is currently being viewed and modified.

This idea is illustrated in figure 6.2. In part (a) of the figure both the design and the current diagram are being shared by the group.[2] In part (b) of the figure Karen has detached her current diagram store from the group's view. This gives her the ability to independently browse through the diagrams. By re-attaching her current diagram store to the current diagram synchronization connector, she immediately

[1] An earlier version of the Workspace model, presented in [102], did not have this property; it was added as a result of user feedback obtained during an informal usability test of the notation.

[2] Only Dave and Karen are shown, but the intent is that Tatiana's design and current diagram stores are also members of the appropriate synchronization groups.

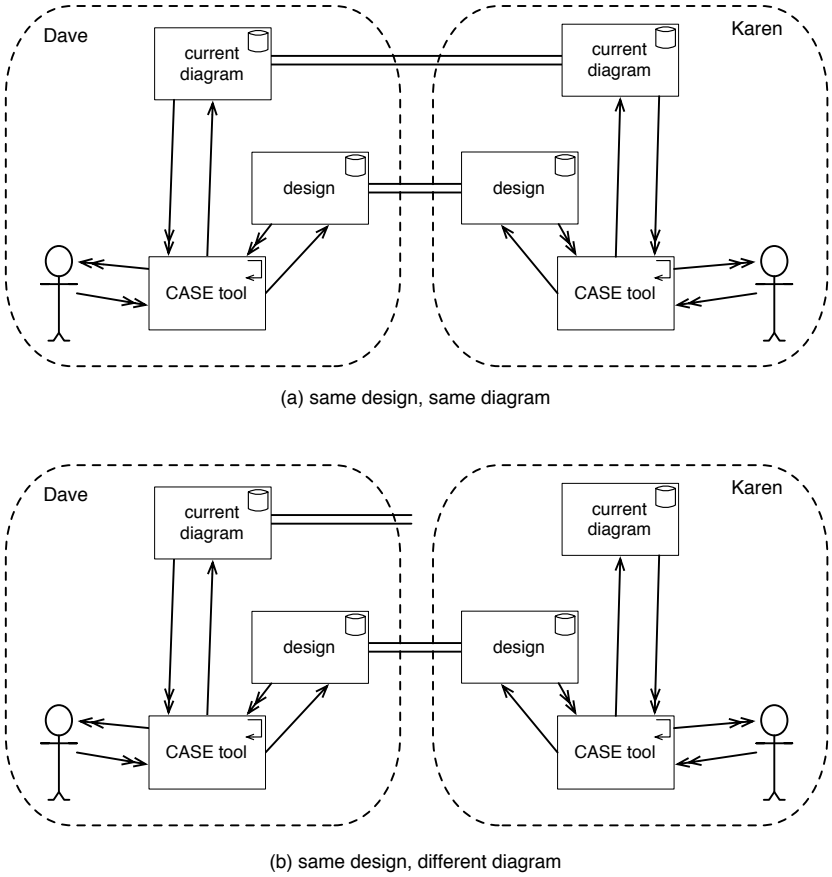


Figure 6.2: Component design for the CASE tool, motivated by the need to both collectively and independently browse diagrams.

---

rejoins the group view. (Recall that when a component joins an existing synchronization group, it is the joining component whose state is modified to match that of the group.)

In addition to the structural implications shown in figure 6.2, this aspect of the scenario has implications on the CASE tool user interface. For instance, the interface will have to provide mechanisms meaning “change diagrams independently” and “join the group view”. The creation of such mechanisms is a complex user interface design problem in its own right. One of the goals of the Workspace Model is to provide an appropriate set of high-level abstractions (like shared stores) that free designers from worrying about low level issues and allows them more time to carry out such design [61].

The representation of shared information is the key issue to be considered in the analysis and design of collaborative systems [97]. Because the Workspace Model provides a clear and explicit representation of information sharing, the modelling technique encourages questions regarding how information is to be shared and how shared information should be divided into components. The separate components for the design and current diagram components shown in figure 6.2 arise simply and naturally from questions that the model forces us to ask in order to draw appropriate diagrams.

6.2.2 Evolution view

We can also consider the evolutions that would be required to create the architecture shown in figure 6.2(a) to derive some insight into the dynamic properties of the system's architecture.

In order for Karen to join Dave's design session as shown in figure 6.2(a), two synchronization connectors must be created and attached to the design and current diagram components in Dave's workspace. Appropriate components of the same types must be created in Karen's workspace and then connected to the same synchronization connectors. In principle either Dave or Karen could perform any of these actions. In practice, it appears reasonable that Dave would create the synchronization connectors and attach his components to them, then provide the identifiers for those synchronization connectors to Karen so that she could likewise attach her components.

Of course, Karen and Dave should not be expected to think in terms of connectors and components, which are concepts more suited to application developers than to end users. Ideally, there would be a user interface in Dave's workspace (perhaps as part of his CASE tool) that would allow him to say, in effect, "share this design with Karen", and a similar interface in Karen's workspace that would allow her to say "work with Dave on his design".[3]

[3]This kind of requirement is normally referred to as *session management* and mechanisms and user interfaces supporting it are a necessary part of any groupware system [111].

Continued analysis would involve the creation and modelling of further scenarios to fully explore the expected behaviours of the system and the implications of those behaviours on system architecture and user interface.

6.2.3 Component interaction design

As illustrated in the previous sections, the Workspace Model provides support for an initial division of a running system into logically-necessary components. In figure 6.2 we see three components for each user: a user interface, a design and a current diagram. This provides a reasonable starting point for detailed design.

Diagrams like figure 6.3 represent runtime snapshots, similar in nature to the Unified Modelling Language's (UML) *object diagrams* [115]. As object diagrams may be extended with message sequences to create UML *communication diagrams*,[4] so may Workspace models be extended to create Workspace communication diagrams. This extension allows us to design the sequence of calls and messages that result from any initial activity in the system, effectively designing the dynamic behaviour of the configured system.

Figure 6.3 shows a simple collaboration imposed over the diagram of figure 6.2(a). This illustrates one design for the sequence of events that is to unfold when Dave indicates he wants to move to the next diagram in his design. As with UML communication diagrams, the numbers on each message or call indicate the sequence of events. Here, the intended sequence is:

1. Dave activates some feature of the CASE tool interface that means "go to the next diagram".

2. The CASE tool component calls an inc() method provided by the current diagram component. This changes the state of Dave's current diagram component, as well as all current diagram components synchronized with it.

[4] Formerly called *collaboration diagrams*, but fortunately the new name clashes less with common groupware usage!

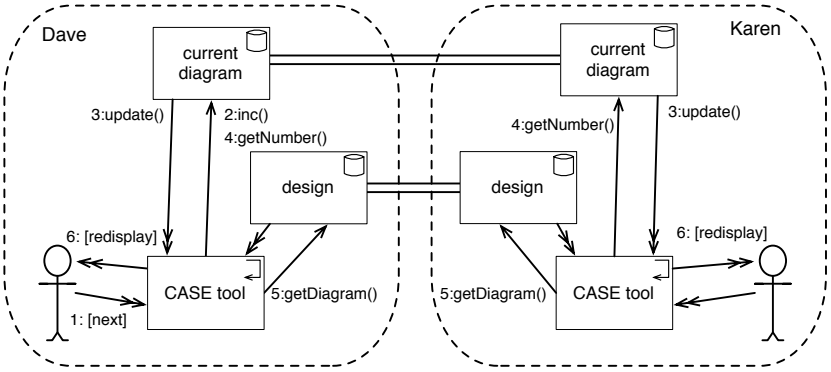


Figure 6.3: Workspace communication diagram.

---

3. Dave and Karen's current diagram components emit asynchronous update() messages on their outgoing subscription connectors. These messages are received by their respective user interface components.

4. The CASE tool components query the current diagram components for the current diagram, then...

5. ... get that diagram from the design components and...

6. ... update Dave and Karen's displays to show the appropriate slide.

The sequence shown in steps 2 through 4 is the classic Model-View-Controller (MVC) object interaction [74], with the CASE tool component playing the part of the view and the controller, and the current diagram component the part of the model.

6.2.4 Analysis of alternatives

For stores to be generally useful in representing shared information, it is almost always necessary for them to provide some form of notification when their internal state changes. This allows, *e.g.*, Karen's user interface component to be notified when Dave advances to the next diagram. The exact mechanism used for the notification may vary considerably and is a significant application design decision. For instance, here we have used a simple "update" message, without any other information, as in the original Smalltalk-80 conception of MVC. It would also be possible to

use an information-bearing message that includes the new diagram number in the message itself, as well as any of several other variations [118].

The important point is that Workspace collaboration diagrams render design choices of this kind explicit and allow them to be reasoned about. For instance, it is always the case that workspace components may be located remote from one another across networks. Thus, the sequence shown in steps 2 through 4 of figure 6.3 may require six network transmissions in the worst case, one each way for each call or message. This could be reduced to four network transmissions by including the new slide number in the update message, which might make Dave's display appear more responsive to his commands.

The responsiveness of Dave's display could be further improved by keeping a local copy of the current diagram in the CASE tool component and fetching the new diagram before issuing the increment command to the shared current diagram.

The cost of this approach in comparison to the approach shown in figure 6.3 would be duplicated code and added complexity in the CASE tool component, as well as the possibility of new failure modes. For instance, knowing to ignore the incoming update() message requires the addition of new logic to the CASE tool component and current diagram components, possibly including a serial number to link inc()s to update()s. It also introduces the possibility of race conditions (what if Karen had moved to a new diagram at about the same time?) as well as incoherent behaviour (what happens if the local copy of the current diagram in CASE tool gets out of sync with that maintained by the current diagram component?) Reasoning about these kinds of trade offs is an important part of the design process; modelling them using workspace collaboration diagrams renders the reasoning more explicit and therefore more rigorous.

In order to fully understand the required dynamic behaviour of a system it is necessary to consider the interesting scenarios and activities that might arise in its use. Each of these may then be modelled in a collaboration diagram to describe its dynamic behaviour. The collection of collaboration diagrams may then be considered as representing an initial dynamic specification of the required system behaviour.

6.3 Architectural analysis

In addition to providing support for the design of dynamic groupware systems, the Workspace Model can also be used to analyze certain non-functional properties of groupware systems. In the next few sections we provide a detailed discussion of the use of the Workspace Model to reason about fault tolerance. Similar analyses are also possible for properties such as security, and performance issues such as bandwidth usage, feedback and feedthrough.

What makes the Workspace Model particularly well suited to these kinds of analyses is its clean separation between the conceptual and the implementation levels. The conceptual level and component implementations specify the desired functional behaviour of the system. The implementation level expresses the actual distributed system configuration at runtime. Since many non-functional properties emerge from distributed implementations, the Workspace Model's explicit representation of distribution, as separate from desired semantics, allows for precise analyses of these properties.

6.3.1 Fault tolerance

In a distributed system, fault tolerance refers to the ability of a running system to continue to operate in the face of partial failure. The failure may be of one of the nodes on which a running system is implemented; of one or more of the communications links between systems; or of a key piece of software on which the system relies. Failures may be transient or long-lived.

In the Workspace System, the response of a running system to partial failure may be either *restoral* or *recovery*. The best case is a restoral, in which all user-visible system functions are restored, for example by re-establishing a failed network connection. Where this is impossible, the aim of a recovery is to put the system into a semantically coherent state with the minimum impact on the users.

In the Workspace Model, a partial failure manifests itself as one or more unrequested evolutions at the implementation level. For example, the loss of contact

with a node will initially be recognised as the unplanned disconnection of any remote connectors targeting components on that node. In an implemented workspace system, this would first be detectable by the transmitter and receiver components implementing the ends of the failed connection.

The Workspace Model representations of restoral and recovery are illustrated in figure 6.4(a) and (b), using the same notation as figure 1.1 on page 10. Both parts of the figure begin with an implementation level architecture i that is a valid implementation of the current conceptual architecture c; that is, $R(c, i)$. The initial failure is represented as an evolution e_{if} at the implementation level that results in a new implementation architecture i_f, where i_f is not a valid refinement of c. Here, e_{if} is used as a shorthand to represent either a single evolution or a sequence of evolutions; in general, an actual system-level failure will manifest as multiple evolutions.

Part (a) of figure 6.4 represents a restoral. Here, a further sequence of implementation level evolutions e_{i1} through e_{in} are applied to i_f to produce i_r such that $R(c, i_r)$ — that is, such that the resulting implementation level architecture i_r is a valid refinement of c.

Where restoral is not possible, it is necessary to modify the conceptual architecture in some way in order to effect a recovery. This is represented in figure 6.4(b). Here, a sequence of evolutions e_{c1} through e_{cm} are applied to c to produce c_r while a sequence of evolutions e_{i1} through e_{in} (not the same sequence as in (a)) are applied to i_f to produce i_r, such that $R(c_r, i_r)$.

It is always possible to find sequences of e_c and e_i such that $R(c_r, i_r)$, with the trivial solution being one in which all elements of c and i_f are deleted to arrive at empty architectures at both levels. Clearly, this is unlikely to be satisfactory to the system's users! To provide good failure properties, the aim is to find the minimally disruptive sequence of e_c such that there exists a possible sequence of e_i allowing $R(c_r, i_r)$. Just how disruptive that sequence of e_c will have to be is dependent on the initial conceptual level architecture c, its initial implementation i, the nature of the failure e_{if}, and just what the users consider "disruptive".

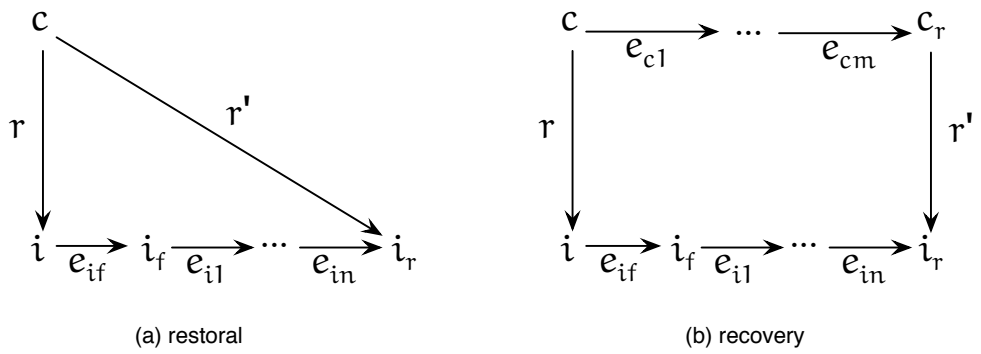


Figure 6.4: Workspace model representations of (a) restoral and (b) recovery.

We do not pretend to have an algorithm meeting this requirement—rather we merely suggest that the Workspace Model's clean separation of the conceptual and implementation levels may provide an appropriate framework for reasoning about it.

6.3.2 Recovery and restoral examples

To illustrate these concepts concretely we return to the "clicker" example of section 1.4.2.

Recall that the scenario has Shona and Tristan are working at two doors of an auditorium, counting the number of people who enter. They are each provided with a "clicker" device that increments the count by one whenever they click its button and also displays the current count. Since the current count is represented by shared stores, Shona and Tristan are updating the same count at the same time.

We suppose that Shona and Tristan each have small tablet computers on which their Clicker user interfaces are to be implemented (that is to say, their Clicker interfaces are anchored to their tablet nodes). Additionally, Shona's workspace includes a server node hosted on another computer; this server node is "headless" and therefore does not allow the instantiation of user interface components. The tablets and the server are all connected by a wireless networks. This conceptual-level architecture is illustrated in figure 6.5.

Applying the refinement rules given in chapter 5, we see that there are many

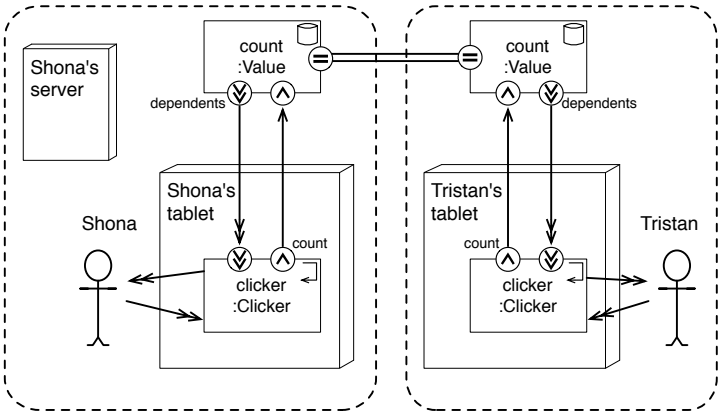


Figure 6.5: The conceptual level architecture of the clicker example, with available nodes shown.

---

possible implementations of this conceptual level architecture. For the purposes of this section, we consider four of these:

- a pure peer-to-peer implementation, in which the Value store has implementations on both tablets and the server is not used (figure 6.6 on page 136);
- a server-based peer-to-peer implementation, in which the Value store has implementations on Shona's server and Tristan's tablet (figure 6.7 on page 137);
- a centralized store implementation, in which the Value store is implemented only on Shona's tablet (figure 6.8 on page 138); and
- a centralized store implementation, in which the Value store is implemented on Shona's server (originally presented in chapter 1, figure 1.4 on page 15).

Likewise there are many possible failures e_{if} that we might imagine; here we consider two. The first example is a transient hardware failure on Shona's tablet, e_{trans}, which causes it to lose all data; however, the tablet remains connected to the network and is still a usable node in Shona's workspace. An analysis of restoral or recovery from this failure is presented in section 6.3.3. The second example is a catastrophic failure of Shona's tablet, e_{cat}, which removes it from her workspace entirely; its analysis is presented in section 6.3.4.

6.3.3 Transient failures, e_{trans}

At the instant after the transient failure of Shona's tablet, all three implementation architectures $\mathfrak{i}_f$ will look the same as shown in their respective figures, except that the node representing Shona's tablet will be empty and the internal state of all components that had been on that node will be irretrievably lost. The analysis regarding restoral or recovery begins by comparing this implementation architecture with the conceptual architecture of figure 6.5, and then determining whether a recovery or a restoral is possible.

In the case of the transient failure, all nodes remain available in the architecture. This means that the conceptual architecture $\mathfrak{c}$ is unaffected and that a complete structural restoral is possible in principle for all three implementation architectures. By "structural restoral" we mean that it is possible to create all necessary implementation components, on appropriate nodes, such that we have an implementation $\mathfrak{i}_r$ where $R(\mathfrak{c}, \mathfrak{i}_r)$. However, there will be varying degrees of disruption to Shona and Tristan's use of their clicker interfaces, and in one case the current count will be lost.

6.3.3.1 Peer-to-peer implementation, e_{trans}.

In the peer-to-peer implementation illustrated in figure 6.6 at the instant after the transient failure, both Shona's Clicker and Value will have been removed from the architecture, along with all supporting infrastructure components and their connectors.

Since $\mathfrak{c}$ shows that Shona has a Clicker anchored to her tablet node, and her tablet node is available, we can instantiate a new Clicker there. Obviously this will not have any of the local state that her former Clicker had, since that state has been irretrievably lost. However, since the only local state of a Clicker is its window title, this is unlikely to be significant.

The conceptual level also requires that Shona's Clicker be attached to a Value (synchronized with Tristan's Value) via call and subscription connectors. To provide

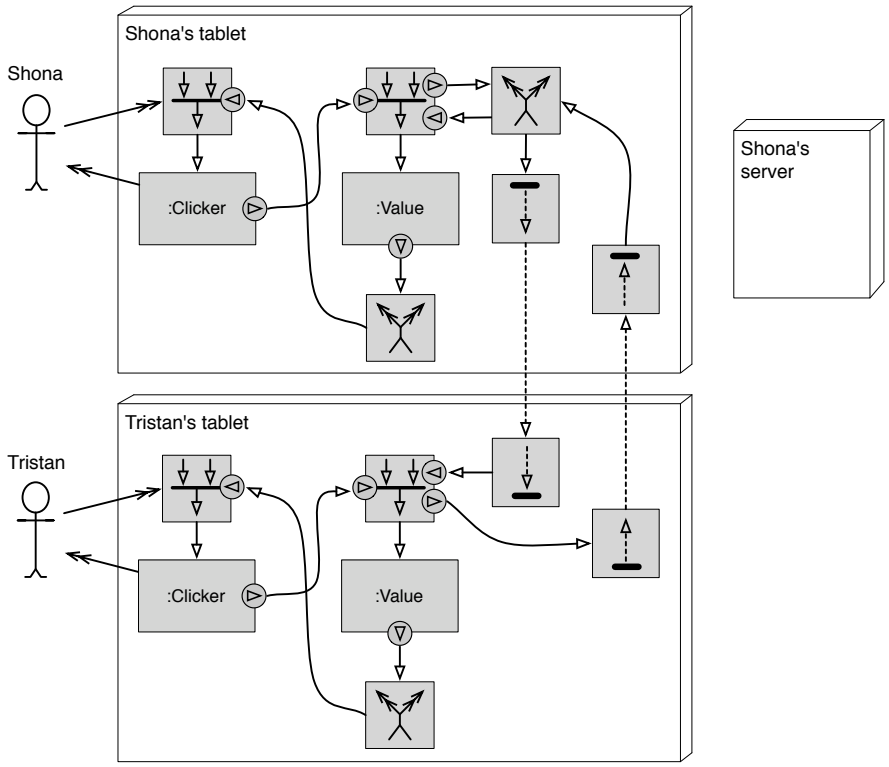


Figure 6.6: A peer-to-peer replicated store implementation of the clicker example.

Shona with this Value we have several choices. The simplest would be to have Shona remotely access Tristan's Value component; this effectively moves us to a centralized implementation like that of figure 6.8 except with the Value and its associated infrastructure components on Tristan's tablet rather than Shona's.

Alternatively we could create a new Value for Shona by copying the state of Tristan's Value and instantiate it one either Shona's server or her tablet. We could then create the required connector implementations, either local or remote as necessary. With either choice, the synchronization connector could be implemented using any one of the allowed channel implementations, for example with a centralized message broadcaster (as in figure 6.6) or channel endpoints.

Structurally, this represents a complete restoral. In terms of the state of the application, it is near-complete: Shona's Clicker window title has been lost; however, the count has been maintained. Shona will temporarily lose access to her Clicker while the system is being restored; however, Tristan retains the ability to use his

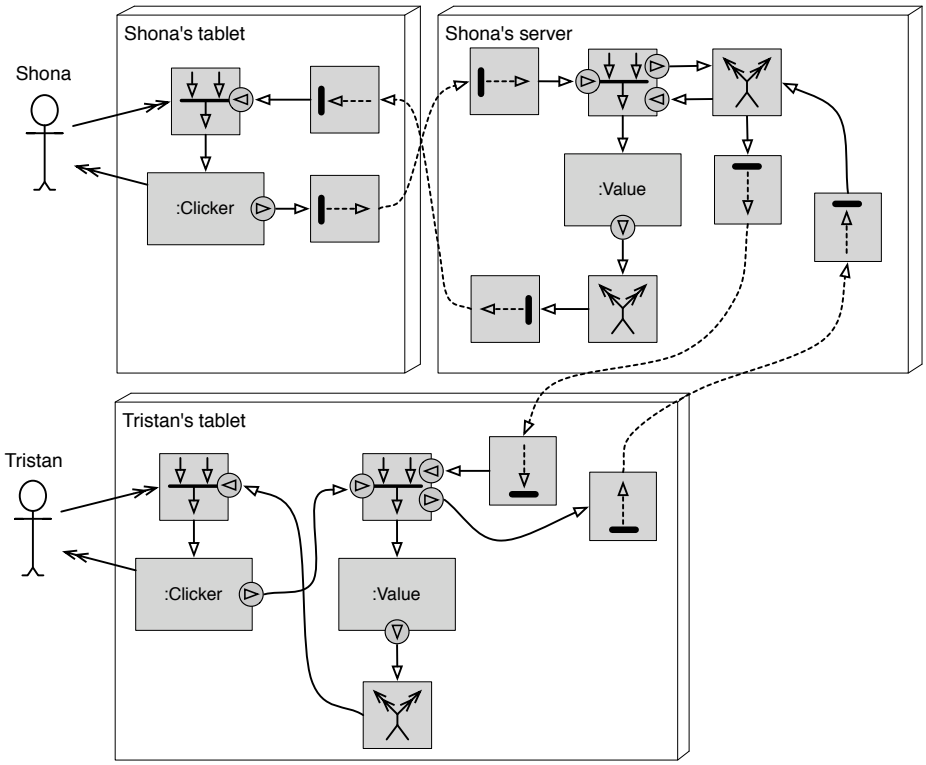


Figure 6.7: A replicated store implementation of the clicker example, in which one replica is on Shona's server.

Clicker user interface throughout the process.

6.3.3.2 Centralized implementation on Shona's tablet, e_{trans}.

We now consider the effects of the transient failure, e_{trans}, on a centralized implementation in which the the sole Value implementation is on Shona's tablet. This implementation architecture is illustrated in figure 6.8.

As in the case of the peer-to-peer architecture, the implementation architecture immediately after the failure is the same as figure 6.8, but with all the components and connectors on Shona's tablet erased. Also as in that case, we can provide a structurally complete restoral, using any one of a number of implementation architectures.

The most significant difference in this case is that the current state of the Value object (the current count) will have been irretrievably lost when Shona's tablet

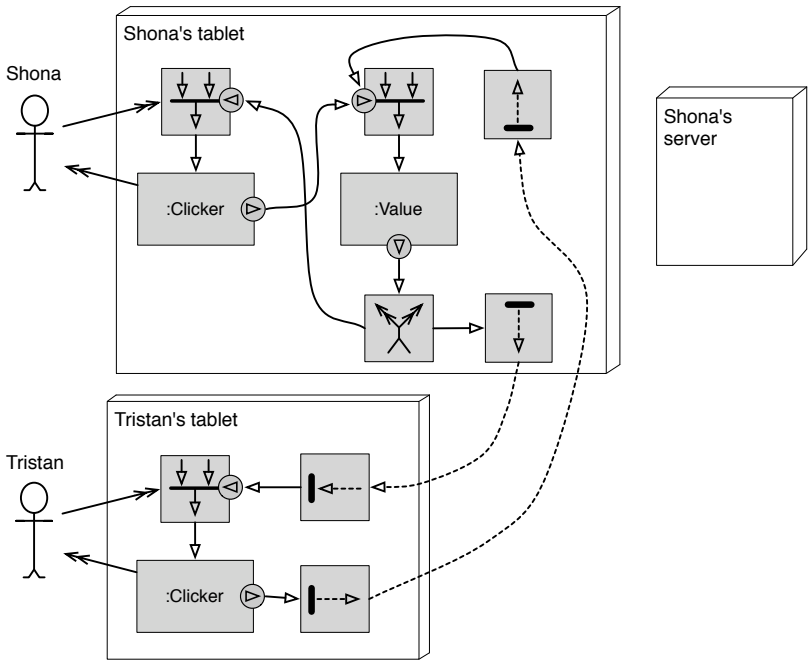


Figure 6.8: A centralized store implementation of the clicker example, with the store on Shona's tablet.

---

failed, since that was the only place it was stored. When the first of the required new Value components is created and connected to a Clicker it will be initialized to a count of zero. Clearly this will be disruptive to both Shona and Tristan.

6.3.3.3 Centralized implementation on the server, e_{trans}.

The transient failure of Shona's tablet in the centralized store on server case (figure 1.4) is very similar in effect to the transient failure on the peer-to-peer implementation.

As in that case, Shona will experience a transient loss of access to her Clicker while the restoral is in progress and Shona's restored Clicker will have lost its previous window title. However, as in the peer-to-peer case the count will be undisturbed and Tristan will retain the ability to use his Clicker throughout.

6.3.4 Catastrophic failures, e_{cat}

In the case of a catastrophic failure of Shona's tablet, the three initial implementation architectures i_f will be the same as shown in figures 6.6 through 1.4, except with Shona's tablet, all the components on it, and all the connectors either on it or to or from it, removed.

Since Shona's tablet is not longer physically present we clearly cannot provide a complete restoral, so some series of conceptual level evolutions e_c will be necessary to create an architecture c_r that will allow recovery.

Because of the failure of Shona's tablet we are required to remove its node from the conceptual architecture c. In order to do this we must first float any components anchored to it (see figures 4.7(b)) and then destroy the node itself (figure 4.10(b)). In principle this would allow Shona's conceptual-level Clicker to remain in her workspace, as long as it could be implemented on some other node. However, since Shona's server does not permit the instantiation of user-interface components, there is nowhere to actually create a new Clicker for her (Tristan's tablet may not be used since it is not in Shona's workspace.) The only way to resolve this is to remove Shona's Clicker and all its ports from the conceptual-level architecture through a destroy evolution (figure 4.8(d)); however, in order to do this we must first detach all attached connectors (figures 4.13(a) and (d)).

Shona's Value component is unaffected by these changes: since it was not anchored to the tablet node it need not be floated; and since it can still be implemented on either Shona's server (or Tristan's workspace, as part of a group of synchronized stores) it need not be destroyed.

The end result of these evolutions is the conceptual architecture for recovery c_r illustrated in figure 6.9.[5]

We can then determine the implementation level evolutions necessary for each of the three i_f such that we have a recovered implementation i_r which is a legal implementation of c_r, that is, $R(c_r, i_r)$.

[5] We could also destroy the two unterminated connectors in this architecture, but that is not strictly necessary since they will have null implementations.

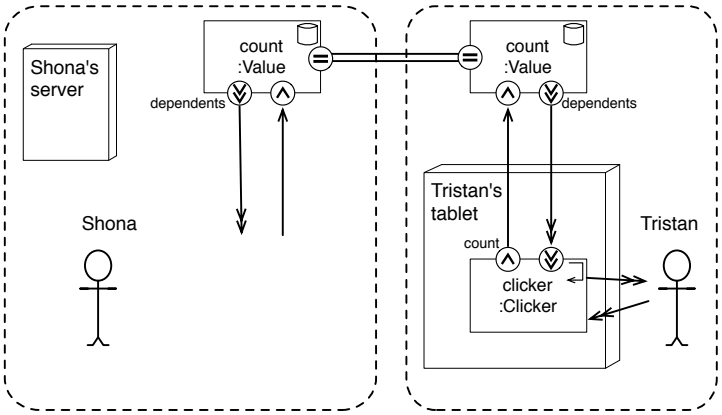


Figure 6.9: A possible conceptual architecture for recovery, c_r, resulting from the catastrophic failure of Shona's tablet.

6.3.4.1 Peer-to-peer implementation, e_{cat}.

In the case of the peer-to-peer implementation and catastrophic failure of Shona's tablet, the main implementation evolutions requires required support the implementation and connection of Shona's Value component. We can either instantiate a new Value component on Shona's server, synchronizing it with Tristan's by any legal synchronization implementation, or we can simply switch to a centralized implementation of the synchronization in which the Value on Tristan's tablet is used to implement both conceptual level stores.

The end result is that Shona loses her Clicker user interface; however, Tristan's interface and Value are unaffected, as is the current count value.

6.3.4.2 Centralized implementation on Shona's tablet, e_{cat}.

If the catastrophic failure occurs to Shona's tablet while the implementation is as shown in figure 6.8 then we have lost not just Shona's Clicker component, but also the Value component that was implementing both Shona and Tristan's Valuestores. As in the analogous case for the transient failure, the current value of the count will be unavoidably lost.

In this case we must create one or more implementations for Shona and Tristan's

Value stores. (Since they are synchronized, a single implementation will suffice; however a peer-to-peer implementation is not precluded.)

The end result is that Shona loses her Clicker interface, the current value of the count is lost, and Tristan experiences the temporary inability to update the count.

6.3.4.3 Centralized implementation on the server, e_{cat}.

Finally, for the centralized implementation of the Value store on the server, only Shona's Clicker and the connections to and from it are affected. Tristan should see no interruption in the availability of his Clicker interface, and the count will not be lost.

6.3.5 Designing for fault tolerance

In the preceding sections we have discussed fault tolerance analysis from the perspective of examining the ability to provide service restoral and recovery after failures, given an initial implementation of a particular conceptual architecture.

It is perhaps more useful to approach the problem from the other direction: given a conceptual architecture, including a set of available nodes, it is possible to use the Workspace model to design a runtime architecture to meet given fault tolerance requirements.

For example, in the clicker example above, we might imagine that the most important feature of the system is maintaining an accurate count. If we assume that the portable tablets are more failure-prone than the server, a server-based centralized implementation might be most appropriate. Alternately, a replicated implementation will provide a degree of system redundancy.

The key point is that the Workspace Model provides us with an explicit representation of the distributed system implementation, distinct from the desired functionality, which allows us to reason about these issues in a rigorous fashion.

6.4 Application development

In addition to its uses for analysis, design, and reasoning about non-functional runtime attributes, the Workspace Model can also be used directly for application development and the runtime control of system execution, given an appropriate development toolkit and a runtime system that supports the appropriate semantics.

We have implemented one such toolkit and runtime system called fiia.[6] The fiia toolkit is implemented in the Python programming language [134] using the wxPython interface to the wxWidgets widget set [107, 46, 120], and supports application development in Python. The operations and semantics of the model are exactly those defined in the Workspace Model; however, the syntax of evolution and reflection operations has been adapted to make it more natural for Python programmers and convenience functions have been provided. In this section we explain how an application programmer develops software using fiia and provide short code samples. In chapter 7 we provide an overview of the internal implementation of the fiia runtime system.

The Python language binding provided by fiia is just one possible approach to using the Workspace Model in software development. A separate toolkit with a radically different syntax and programming model is currently being implemented by Christopher Wolfe in the Microsoft .NET [108] environment, as part of a related research effort.

6.4.1 The fiia programming model

A programmer using the fiia toolkit has two main tasks: developing fiia-aware components and providing sequences of conceptual-level evolution operations that assemble the desired configurations of these components and their connectors at runtime. The two tasks are not necessarily separate, since the evolution scripts may themselves be embedded in fiia-aware components; however, the evolution scripts

[6] The name fiia is drawn from Ursula K. Le Guin's 1966 novel *Rocannon's World*, in which members of an alien race called the Fiia are described as intelligent, nomadic, communal, telepathic and the willing servants of a human-like race.

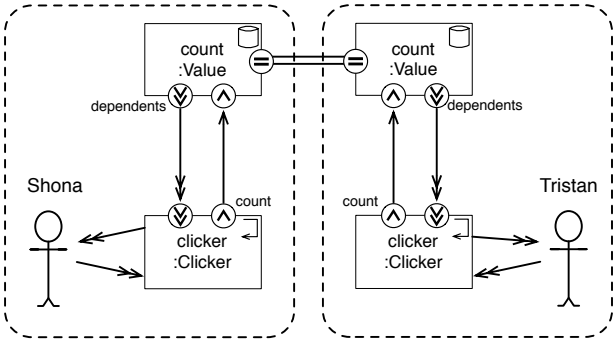


Figure 6.10: A simple “clicker”.

---

may also be independent of the component definitions.

In this section we present component definition and configuration in fiia. The discussion is based around the example presented in chapter 1, which is illustrated in figure 6.10. Once again, the scenario has Shona and Tristan working at two doors of an auditorium, counting the number of people who enter. They are each provided with a “clicker” device that increments the count by one whenever they click its button and also displays the current count. Since the current count is represented by shared stores, Shona and Tristan are updating the same count at the same time.

From a Workspace Model perspective, each of Shona and Tristan have an interface reactor of type Clicker, which provides a single-button interface. Initially, the button’s label is a question mark. When the Clicker is connected to a store of type Value by the connectors shown, its button’s label takes on the value stored, which is expected to be an integer. Each time the button is pressed by either Shona or Tristan, the value of the integer is incremented by one. A change in the integer’s value causes a message to be sent via both Value’s subscription ports; receipt of this message by a Clicker causes it to update its button’s value accordingly.[7] If the Clicker is disconnected from the Value, the button’s label reverts to a question mark.

While obviously very simple, this application serves to illustrate most of the key features of the Workspace Model as implemented in fiia.

In section 6.4.2 we describe the key fiia evolution operations required to config-

[7]This is a classic model-view-controller style interaction [74].

Executed in each workspace:

```
count = fiia.Component(Value)
clicker = fiia.Component(Clicker, title='fiia Clicker')
call = fiia.connect(clicker, 'count', count)
subscr = fiia.connect(count, 'dependents', clicker)
```

Executed once in Tristan's workspace:

```
sync = fiia.synchronize(count, shonaCount)
```

Figure 6.11: The fiia evolution operations required to configure the "clicker" example of figure 6.10.

ure this application. The necessary fiia component definitions are discussed starting in section 6.4.3.

6.4.2 Evolution operations in fiia

In fiia, evolution requests are represented by a set of functions provided in the interface of a Python module called fiia. The functions required to configure the architecture of figure 6.10 are Component, connect and synchronize.[8]

Figure 6.11 shows the conceptual-level evolution requests necessary to create the architecture of the clicker application shown in figure 6.10, assuming appropriate definitions for the Value and Clicker components. The first four lines of code in figure 6.11 would have to be executed once in each of Shona and Tristan's workspaces. The final line would be executed once in Shona's workspace.

The first line of figure 6.11 requests creation of a new conceptual level component of type Value. The kind of component represented by Value (in this case, store) is determined from the component's definition, as are its ports. Since no workspace was specified, the component is created in the current workspace. The return value from the creation request, assigned to the variable clicker, is a Universally Unique

[8] By Python convention [135], class and factory function names start with capital letters and other function names start with lower-case letters; Component is capitalized as it has the semantics of a factory function.

IDentifier (UUID, a Python character string) generated by the fiia runtime and identifying the conceptual level Value component created.

The second line of the figure requests creation of a Clicker component. The fragment `title='fiia Clicker'` is a parameter that will be passed to the constructor of the implementation-level Clicker component, in this case representing the desired title text of the Clicker window.

The third line requests that the count source port of the component identified by the variable clicker be connected to the component identified by the variable count. No target port name need be specified since default call and subscription target ports are automatically provided by the fiia system for each component. The kind of connector required (here, a call connector) is inferred from the kind of the source port.

The connect function is a convenience function which, when used with call ports, is equivalent to a createCall evolution followed by attach evolutions for the target and source ends, in that order. The value returned by connect is the UUID of the created call connector.

The fourth line of the figure connects the dependents subscription source port of the count object to the appropriate default target port on the clicker. Since subscription connectors are multi-point to multi-point, connect is more complex for subscription connectors than for call connectors: if the requested source port is already attached to a subscription connector, that connector is connected to the target port; otherwise a new subscription connector is created and attached first at the source end then at the target end.

Once these four lines have been executed in each workspace, the only remaining step is for Tristan to synchronize the his Value store with Shona. This requires him to have access to the UUID of Shona's store, represented by the shonaCount variable in the last line of figure 6.11.

Tristan could acquire this UUID via reflection on the architecture. The fiia system provides all the reflection operations specified in section 4.4, as well as a fully

general query mechanism based on the SPARQL query language [105].[9] In principle, these could be used to construct a workspace browser tool, which would make component synchronization a "drag and drop" operation. They could also form the basis of a specialized user interface on the Clicker component for locating shared values. Since fiia UUID's are simply strings, Tristan could also acquire the UUID by an "out of band" mechanism such as email [138].

In any event, once Tristan has the UUID of Shona's count store, he can cause the final line of figure 6.11 to be executed. The synchronize function is a convenience function analogous to connect. In this example it is equivalent to a createSyncGroup followed by an attach at Shona's end then an attach at Tristan's end. As explained in section 4.2.3.4, the effect of this is that the current state of Tristan's count is replaced by that of Shona's, and subsequently the two represent the same count.

If Shona's count store had already been attached to a synchronization group prior to this evolution request, Tristan's store would have joined that group. If Tristan's store had already been a member of a synchronization group, the synchronize function would have reported an error.

In addition to the functions discussed here, the fiia module also provides a disconnect function which is the dual of connect; a diverge function which is the dual of synchronize; and functions implementing all the other conceptual-level workspace operations specified in section 4.3.

6.4.3 Component definition in fiia

Components in fiia are instances of normal Python classes [10] that obey a few simple conventions and conform to a few constraints. Component definitions for fiia implementations of the Value and Clicker components are shown in figures 6.12 and 6.13. The interaction between the components has been implemented using a standard MVC pattern [74] in which the Value sends its dependents an update message when it is modified.

[9] The query language and its implementation are discussed in chapter 7.

[10] A fiia component may be an arbitrary network of Python objects, one of which provides a façade [52] for the component. Here we are speaking specifically of the class defining the façade.

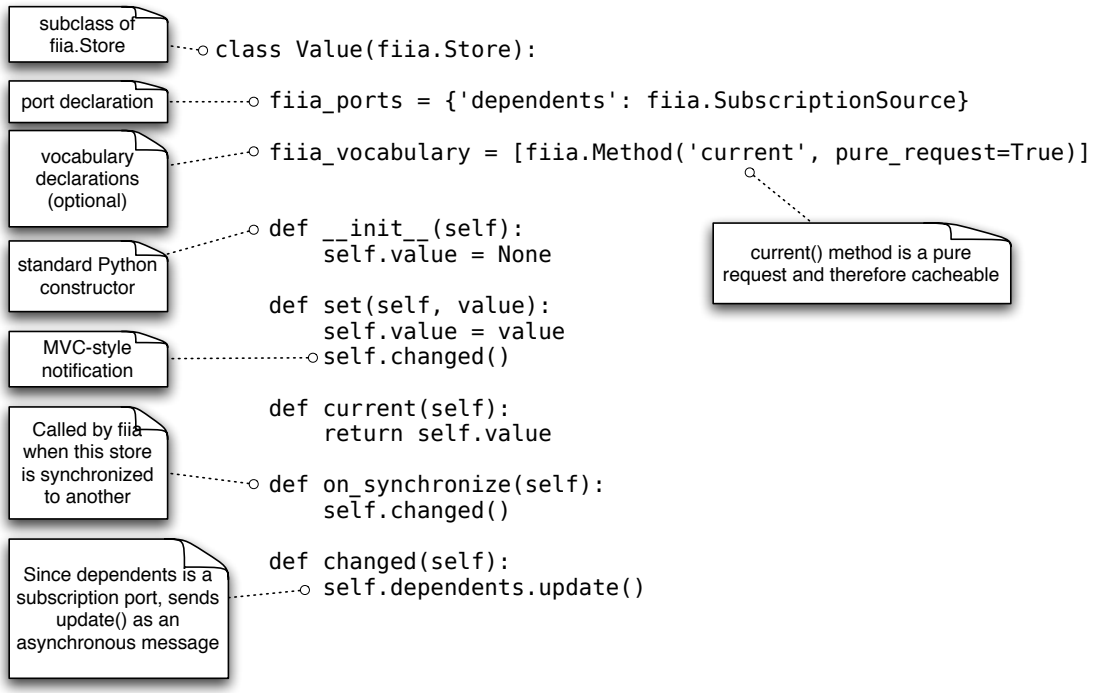


Figure 6.12: Definition of the fiia store Value, as an MVC-style model.

Components in fiia are expected to subclass one of the fiia-provided base classes Store, Reactor or Actor so that the runtime system can automatically determine component kind. In our examples the class Clicker inherits from Reactor and is therefore deemed to be a reactor; the class Value inherits from Store and is deemed to be a store. Following normal Python semantics, inheritance is transitive, so some other class inheriting from Value would also be a store.

In addition to serving as component kind markers, the three fiia base classes are themselves instances of a custom metaclass which provides an appropriate implementation of port and vocabulary inheritance (see below). A subclass may inherit `fiia_ports` attributes from more than one superclass; in this case the port dictionaries are merged with any conflicts being resolved in accordance with Python's standard method resolution order for multiple inheritance. A subclass may also declare its own `fiia_ports` attribute; in this case the new port declarations are merged with the superclass port declarations. Analogous rules are applied for inheritance of the `fiia_vocabulary` attribute.

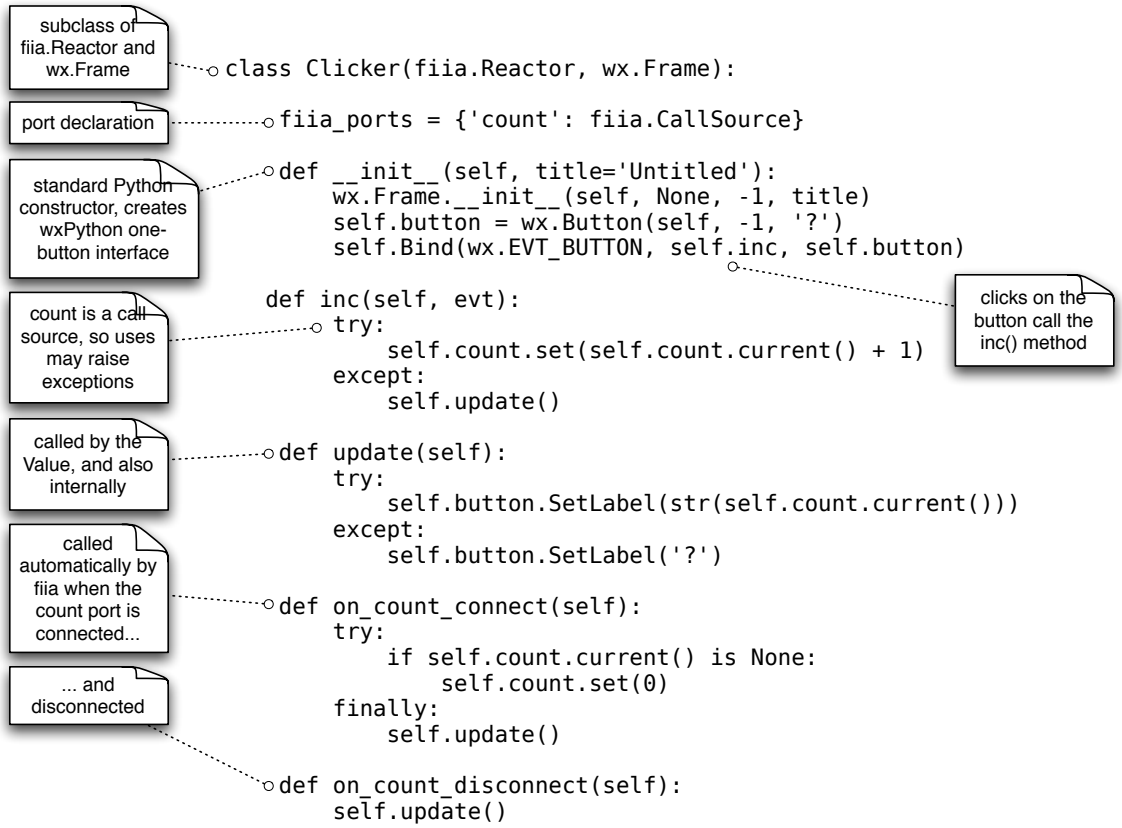


Figure 6.13: Definition of the fiia reactor Clicker, which is also a wxPython Frame.

In order to provide additional flexibility, inheritance from the fiia base classes is not mandatory. Any component that does not inherit from a fiia base class is assumed to be a reactor; also, such components will not support port and vocabulary inheritance.

6.4.4 Port declaration in fiia

In the raw Workspace Model, ports are dynamically added to components after they have been created. The fiia system uses a streamlined approach in which call and subscription target ports are provided automatically; each component has one call target and one subscription target and stores have a synchronization port.

Components may declare any number and combination of call and subscription

source ports by means of a `fiia_ports` class level attribute.[11] Source ports are associated by name with instance variables of the classes that provide them—so for example the Value's "dependents" port declared in the second line of 6.12 is associated with a Valueinstance's self.dependents instance variable. Similarly, in figure 6.13, the class Clicker is declared to have a call source port named "count", which is associated with the instance variable self.count.

6.4.5 Port use in fiia

Since ports are managed by the fiia runtime system, component code should never assign to port variables. If a component wishes to modify one of its ports' connections, it should call the appropriate fiia evolution operation. A component has access to its own UUID through a `fiia_uuid` attribute which is initialized by the runtime system at the time of component creation. Since UUIDs are merely strings, they may be passed from one component to another; this provides a mechanism for bootstrapping arbitrarily complex connections of fiia components using code found within the components themselves.

To make calls on call ports or send messages on subscription ports, a component simply invokes a method on its port variable using the normal Python method invocation syntax. So, for example, the changed method of figure 6.12 is sending the asynchronous update message out the Value component's dependents port. If a subscription connector is attached to the dependents port and that subscription connector has targets, this message will cause the attempted invocation of an update method on each of the target components (see the update method in the Clicker component in figure 6.13). Where a message sent out a port has parameters, these are also delivered with the method invocation; this allows for arbitrarily rich message structures.

Since subscriptions are asynchronous, sending any message out a subscription source port is always legal in fiia. If there is no connector attached to the port,

[11] fiia also provides "multiports", which are essentially lists to which ports may be dynamically added and removed; these are not further discussed here.

or the connector has no targets, or the targets are incapable of processing the received message, any exceptions are silently absorbed by the system (but logged for debugging purposes).

Unlike subscription connectors, call connectors are synchronous; therefore making a call on a call source port may well cause an error to be reported. An error could occur for any one of several reasons: the source port may not have an attached connector; the connector may be implemented across a network link that has failed; the connector may not have an attached target; or an error may occur in the target's processing of the call.

Errors are reported using standard Python exceptions; calls made via call source ports should therefore be protected using exception-handling mechanisms.[12] This is illustrated in the `inc`, `update`, and `_on_count_connect` methods of the `Clicker` class in figure 6.13.

It is often useful for components to perform particular actions when their ports are connected or disconnected. The fiia system provides a simple, lightweight mechanism to support this. If a component provides a method called `on_synchronize` (like the Value component), this will be called whenever that component joins a synchronization group. If a component provides a method with a name of the form `on_`*portname*`_connect` or `disconnect` (like the Clicker component) then the fiia system will automatically invoke this method whenever the port with the indicated name is connected or disconnected, as appropriate.

As specified in section 4.2.3.1, all objects passed on workspace connectors must be passed by value. In fiia this is implemented by having source ports perform a Python deepcopy operation on all parameters and return values for the calls and messages they originate. Additionally, calls, subscriptions and synchronizations may all span process boundaries. This means that any parameter or return value passed out a port may be subject to byte-serialization. Finally, components that may be

[12] Strictly speaking, not all calls need be protected. In particular, a call which will only be invoked in response to another incoming call may simply propagate any errors backwards to the originating call site. However, any calls resulting from user input, from receipt of an asynchronous message, or from the operation of an independent thread in an actor should be protected.

moved from one node to another at runtime may also need to be copied and byte-serialized for transmission. fiia uses Python's standard pickle module for serialization.

6.4.6 Vocabularies in fiia

As discussed in section 4.2.4, a port in the Workspace model may have an associated vocabulary, which defines the calls or messages that it supports.

The fiia system's vocabulary support is strictly optional, and is used to provide information to the runtime system regarding semantic properties of calls on components. This information is used to support certain runtime optimizations, including caching and the avoidance of unnecessary concurrency-control.

fiia vocabulary declarations are made at the component level rather than the port level. Similar to port declarations, vocabulary declarations use the class-level attribute `fiia_vocabulary`. An example of a fiia vocabulary declaration is provided in the third line of the Value component in figure 6.12.

A fiia vocabulary is a list of `fiia.Method` objects, one for each component method of interest. `Method` objects provide two attributes which may be set in the constructor.

The pure_request attribute identifies methods which are guaranteed to make no modifications to component state, which do not access any other components, and (in actors) which will not be modified by any strictly internal processes. The returned values of such methods are referentially transparent, and as such may be cached. This supports the introduction of cache and mirror cache components into call paths, as shown in the refinement of figure 5.9(d), and allows the CCCM component to provide a cache invalidation indication when any non-request call or message arrives at the component. Only methods whose `pure_request` attribute is `True` may be cached. In the Value component, the `current` method is a `pure_request`.

The `no_cc` attribute is used to indicate methods which need not be subjected to a component's normal concurrency control policy. The use of the `no_cc` attribute is motivated by experience from the Clock system [132]. Clock's developers noted that

a frequent logical necessity in groupware systems is for one user to obtain exclusive access to some aspect of a shared object's state prior to modifying it. For example, if one user is moving an object in a drawing program, it is usually nonsensical to allow another user to attempt to move it at the same time. In practice, this means that when a user clicks on an object that user should obtain an exclusive lock on the object's position, and that the user interface component should allow no movement before having obtained such a lock.

Clearly the act of obtaining a lock should be subject to strong concurrency control—if two users simultaneously attempt to click on an object, only one should obtain the lock. However, once the lock has been obtained, the winning user has complete control over the object's position and therefore any methods relating to object movement need not be subject to concurrency control. In practice, over high-latency connections this technique can result in orders of magnitude improvement in perceived feedback times when compared to either a centralized implication or a replicated implementation with naïve concurrency control [132].

6.4.7 Advantages of developing in fiia

In this section we present some of the advantages of programming in fiia.

fiia programs are semi-transparent to distribution. That is, the programmer need not write code to deal with any distributed system issues except those that are inherent in the possibility of partial failure. This is manifested in fiia code as the need to protect calls on call ports with exception handling mechanisms.

This approach closely follows the design principles advocated by Waldo *et al.* in their influential paper on distributed system language design [139], which essentially expresses the rationale underlying the design of the Java Remote Method Invocation (RMI) system.

fiia's design is distinct from RMI in several ways. In Java RMI, method parameters or return types which are serializable are passed by value, while those that implement the Remote interface are passed by reference. Any object that implements Remote is potentially in another process space. In fiia all parameters and

```
    app = wx.PySimpleApp()
    count = Value()
    clicker = Clicker(title='Standalone clicker')
    clicker.Show()
    count.dependents = clicker
    clicker.count = count
    clicker.on_count_connect()
    app.MainLoop()
```

Figure 6.14: Configuring one side of the "clicker" example in stand alone mode.

---

return types traversing ports are passed by value, and access to remote components (attachment of connectors to ports) is arranged by evolution operations.

This provides a fiia component with a clear "inside" and "outside". Any normal object reference is guaranteed to be "inside" (local) and to obey normal Python by-reference semantics. Conversely, objects may be passed out ports without fear of external modification, since they are always passed by value; similarly call and message parameter actuals and return values from calls may be modified with impunity since they are guaranteed to be private copies. We feel that this programming model is simpler and less error prone than the Java RMI model.

One feature of fiia bears repeating: fiia components are simply Python classes that follow a few conventions and obey a few constraints. This means that fiia components may be created using any Python programming environment; exercised outside the fiia runtime system; tested using the standard Python unit testing frameworks; and analysed using standard Python profiling tools. It also means that the user interfaces of fiia components may be constructed using any available Python interface builder. In short, fiia components are full participants in the rich Python open source ecosystem.

As an illustration of this, figure 6.14 shows the Python code necessary to configure and initialize an approximation of one side of the "clicker" example of figure 6.10 in a stand alone mode. It is interesting to compare this with the code in figure 6.11.

Naturally, without the fiia runtime, component synchronization is impossible.

Also, the “subscription” connector created by figure 6.14 has a single target and synchronous, rather than asynchronous semantics. Finally, parameters and return values in a stand alone configuration like this will be passed by reference rather than by value, providing semantic traps for the unwary.

6.5 Conclusion

In this chapter we have illustrated the use of the Workspace Model for scenario based modeling, component design, reasoning about runtime properties like partial failure, as well as its direct use as a programming model.

The Workspace Model’s usefulness in scenario-based modeling for groupware comes directly from the fact that it allows direct representation of key groupware concepts: independent contexts of use, shared information, and communication via shared messaging. These concepts also make the model useful for component level design, in that drawing precise models of groupware systems encourages designers to ask appropriate architectural questions.

The Workspace Model’s usefulness for architectural analysis derives primarily from its clean separation between the conceptual and implementation levels. The conceptual level represents the users’ desired collaborative configurations; the implementation model represents the actual configuration and constrains what is actually possible.

Finally, the programming model offered by the fiia toolkit provides a direct application of the Workspace Model’s conceptual semantics to the problem of groupware development. In the next chapter we describe the internal design of the fiia runtime itself.

Chapter 7

The fiia Runtime System

In the previous chapter we described the application programmer's view of the fiia system. In this chapter we describe our prototype implementation of the fiia runtime system itself.

The fiia runtime prototype was developed mainly to assist with the validation of the refinement rules presented in chapter 5. It also represents a proof-of-concept implementation of the Workspace Model, demonstrating that the model is both implementable and practical.

In developing fiia the emphasis has been on correctness, rather than performance. However, we have implemented several small applications in fiia including the "clicker" example, a distributed slide presentation system, a chat program, and a simple object-based drawing system; all exhibit acceptable interactive performance.

Our prototype supports the full suite of conceptual level elements and evolutions defined in chapter 4, the full suite of implementation level elements and evolutions defined in sections 5.1 through 5.3, and the the full suite of refinements defined in section 5.4. It is capable of incrementally maintaining a correctly-refined implementation level given arbitrary conceptual level evolutions over time. It does not yet support restoral or recovery from implementation level failures.

The fiia runtime prototype is implemented in approximately four thousand physical source lines of Python [134]. It makes extensive use of the Python standard library, as well as the wxPython graphical user interface toolkit [46, 107]. The runtime was itself designed and implemented using Workspace concepts, and the implementation makes extensive internal use of fiia infrastructure components including local ports, CCCMs, message broadcasters, transmitters and receivers. The runtime executes on any platform supporting Python and wxPython, which includes Microsoft Windows, Mac OS X, Linux, and most versions of Unix.

We describe the fiia runtime from the outside in, beginning in section 7.1 with

a brief description of how an application developer can interact with fiia using the node console. This is followed in section 7.2 by a conceptual level description of the fiia runtime's major components and their operation. Section 7.3 provides a discussion of selected implementation level issues. Finally, in section 7.4 we report on our experience using fiiaand present some basic performance measurements.

7.1 Interacting with the fiia prototype

Nodes in fiia are started by executing a Python initialization script. In the current fiia prototype there is always one *main node*, which must be started first and which listens on port 16025 for incoming TCP/IP connections from other nodes. The rest of the nodes in the system are *thin nodes* which are provided with the main node's host name at startup in order to establish a connection. The main node includes the current architecture representation and the machinery for performing evolutions and refinements; thin nodes include only the node console and a local node manager. There may be any number of thin nodes, each of which may be in any workspace. The implementation level and communications architecture of fiia are discussed in more detail in section 7.3.

After initialization, each node provides an interactive Python interpreter, called the node console, which allows for the execution of arbitrary Python programs. Since the node console is "inside" fiia, these programs may include arbitrary calls on the runtime itself. The node console is intended for use by developers. In an end-user system, the node console would be replaced or supplemented by a graphical program launcher and collaborative session manager.

Figure 7.1 shows the node consoles of two nodes that are running the "clicker" example from section 1.4.2, along with the clicker one-button interfaces themselves. In the node consoles, the command prompt is indicated by ">>>" and lines which begin with this are Python statements entered by the user. Other lines are output of the Python interpreter.

The upper node console shows the commands which Shona's node must execute in order to create her clicker. After the first line, which imports the definitions of

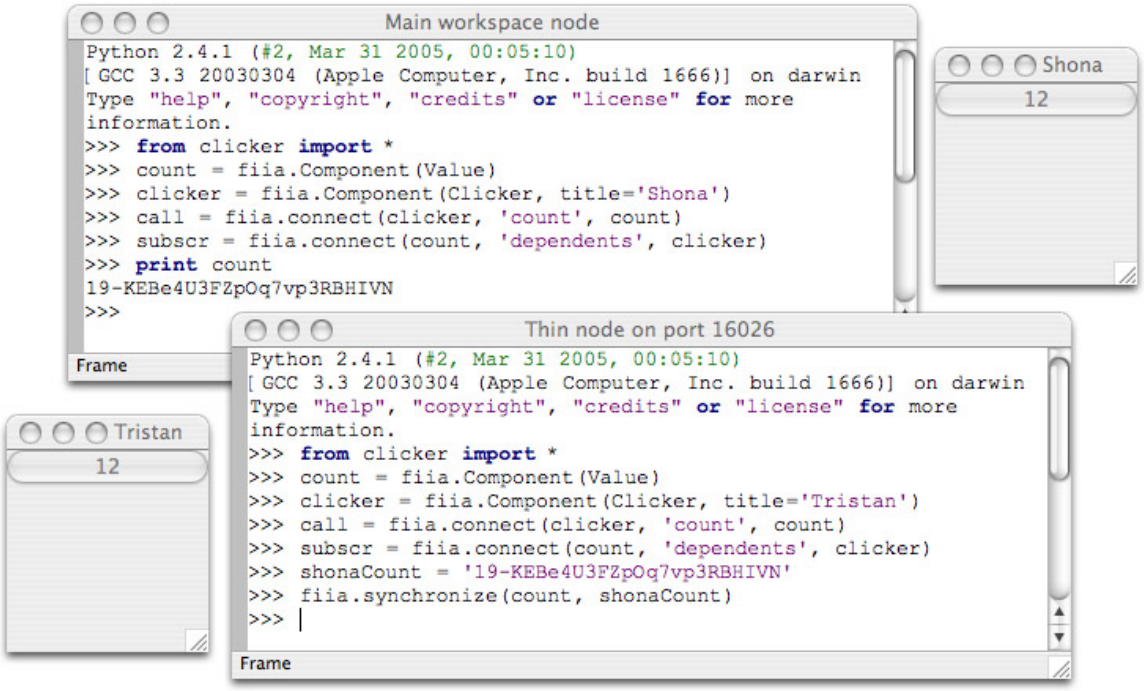


Figure 7.1: The clicker example running in fiia, on two nodes, with both node consoles shown. (The node console is a development tool allowing direct interaction with the fiia runtime.)

---

the Value and Clicker classes from the clicker module, the next four lines are exactly those shown in figure 6.11. The first five user-entered lines in the lower node console (Tristan's) are exactly the same.

In order to allow for the synchronization of Tristan's count to Shona's, Tristan needs the UUID of Shona's count component. Here we have satisfied this requirement by printing the UUID in Shona's node console (it is the long string beginning with "19-") and copying and pasting it into Tristan's node console. The final user-entered line in Tristan's node console effects the synchronization.

The node console allows full access to fiia's commands, including the conceptual level evolutions and the reflection operations. It also allows full introspection of all components implemented on its node, which provides an extremely useful tool for debugging purposes.

This completes our brief overview of the user interface of the fiia prototype. We now discuss the runtime's implementation, at the conceptual level.

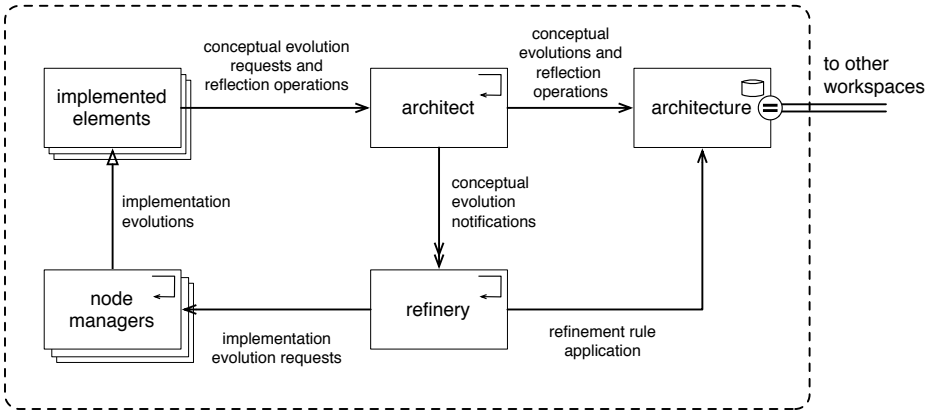


Figure 7.2: A conceptual level view of the fiia runtime.

7.2 Conceptual level view of the fiia runtime

A conceptual level view of the fiia runtime's internal structure is shown in figure 7.2.[1] It consists of:

- at least one architecture store, which maintains a current model of the system's conceptual and implementation architectures and supports conceptual and implementation evolutions, as well as reflection operations, exactly as defined in chapters 4 and 5;
- one architect reactor per architecture, which provides the "Pythonized" syntax discussed in section 6.4, translates it into operations on the architecture, and advises the refinery of any conceptual level changes;
- one refinery reactor per architecture, which applies refinement rules to the architecture in response to notification of conceptual level evolutions, and which then requests the actual realization of implementation level evolutions by the appropriate node manager
- one node manager reactor per node, which creates, destroys, manipulates and maintains direct references to all implemented workspace elements, including user-defined components, infrastructure components, and ports; and
- the implemented components themselves, from which conceptual level evo-

[1] In this and the following diagrams in this chapter, ports are elided to reduce clutter.

lution and reflection requests may originate. The node console is an implemented component.

The fact that notifications of changes to the architecture are sent to the refinery by the architect, rather than by the architecture itself, bears explaining. This approach has been adopted to prevent what we call the "duelling refineries" problem. If the refinery received its change notifications directly from the architecture, then (based on the definition of synchronization) all refineries in the global architecture would be similarly notified. This would result in all refineries attempting to perform refinements simultaneously on the same elements, which is clearly undesirable. By configuring the patterns of communication as we have, we ensure that a single refinery is responsible for refinements resulting from any particular conceptual level change.

It is also worth noting that the connection between a node manager and its implemented elements is a local connector rather than a call connector. In fact, the node manager maintains a Python object reference to each implemented component and port on its node, which it uses to effect implementation level evolutions on them.

7.2.1 Evolution and incremental refinement

Before describing the runtime dynamics of the fiia system we first present fiia's approach to the problem of incremental runtime evolution.

The refinement rules defined in section 5.4 allow us to generate the complete set of implementation architectures for a given conceptual level architecture. However, in a runtime system this is not sufficient, since we need to provide for incremental implementation evolution at runtime. The key problem is this:

- given a current conceptual architecture c, an implementation i satisfying $R(c, i)$ (that is, i is a correct refinement of c), and a conceptual evolution of c resulting in c',
- find a sequence of implementation level evolutions of i that produces i', where $R(c', i')$.

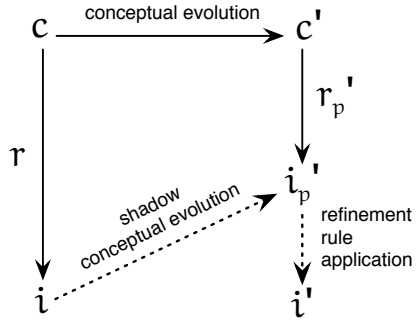


Figure 7.3: The incremental implementation evolution process as implemented in fiia.

Of course, an arbitrary sequence of such implementation level evolutions is unlikely to satisfy the user: rather, we require a sequence of evolutions that is both rapidly implementable and minimally disruptive.

Figure 7.3 illustrates the approach that we have taken in fiia to solving this problem. When a conceptual evolution is applied to c, the runtime records this in the conceptual architecture. It then applies a series of *shadow conceptual evolutions* to i. The effect of these shadow evolutions is to produce a partially refined architecture i'_p that is a correct refinement of c but that may include some conceptual level elements. The fiia runtime then applies its set of refinement rules (exactly the rules defined in section 5.4) to i'_p, producing i' such that $R(c', i')$ and i' is a completely implemented architecture.

This process allows fiia to incrementally respond to arbitrary conceptual level evolutions with only minimal changes to the implementation level architecture.

We envision that a similar system could be employed for making dynamic adjustments to the implementation level (and possibly also conceptual level) architectures in response to changing performance requirements or environmental conditions, including partial failure. This appears to be considerably more complex than supporting incremental conceptual evolution and remains for future work.

In the next section we provide a small example that illustrates the fiia runtime's dynamic response to conceptual level evolution, based on the clicker example presented in section 7.1.

7.2.2 Runtime dynamics

Figure 7.4 illustrates the runtime's dynamic behaviour in response to the conceptual level connect request in the fourth user-entered line of figure 7.1:[2]

```
call = fiia.connect(clicker, 'count', count)
```

This line requests the creation of a connection from the clicker component's "count" port to the default source port on the count component. In the following description, we assume that the clicker and the count are implemented on a single node, node n.

The process begins with the node console requesting that a connection be made from the clicker's "count" port to the count component (1). This request is received by the architect which then requests that the architecture create a new conceptual level call with a freshly generated UUID of k (2), and sends a notification to the refinery that this new call has may require refinement (3). Since the notification is an asynchronous message the refinery's response will occur in a separate thread; possibly in parallel with the subsequent calls from the architect to the architecture. For presentation purposes, here we imagine that the architect's calls on the architecture execute first.

Once the call k has been created, the architect asks the architecture for the UUID p corresponding to the clicker's "count" port (4) and tells the architecture to attach p to the connector k (5). It then notifies the refinery that both p and k have changed and may therefore need refinement (6). This process is repeated for the other port, q (7, 8, 9). At the conceptual level, we have now completed the requested connection.

When the refinery receives the notification of a conceptual level change, it follows the incremental evolution and refinement process described in the previous section. It does this by successively applying each of a list of rules to the architecture attempting to perform a graph match outwards from the element that has changed. Most match attempts will fail. It first applies conceptual shadow evolution rules, then refinement rules.

[2] In figure 7.4, some parameters of the calls and messages shown have been elided to simplify the presentation.

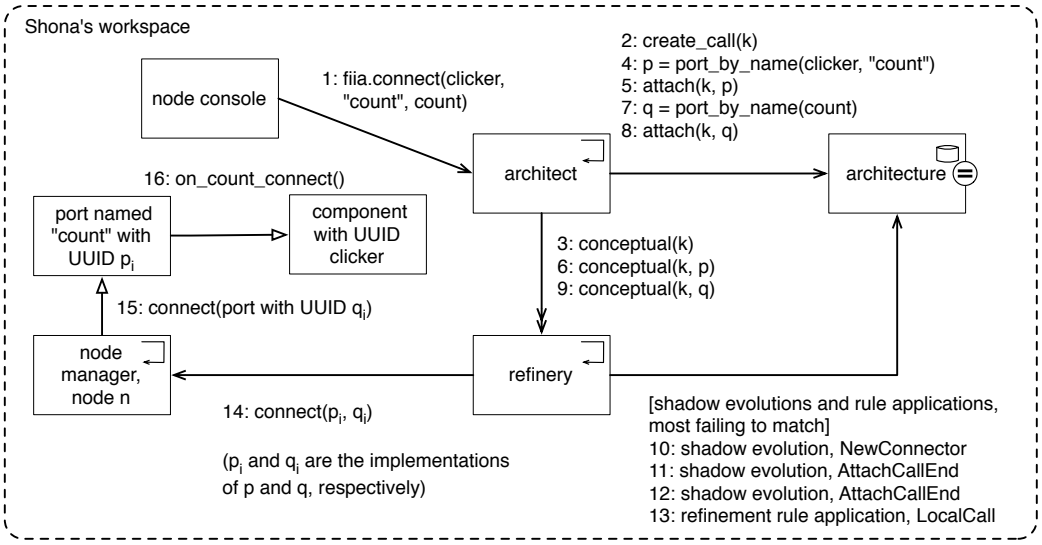


Figure 7.4: Example of the fiia runtime dynamics.

When a rule does match, the rule performs a transformation on the implementation level architecture and reports any changed elements back to the refinery. The refinery queues these changed elements for recursive rule application attempts, since the transformation resulting from one rule's match may allow another to match as well. If the implementation level changes require the actual instantiation, connection, disconnection, destruction, *etc.*, of an implemented element, the refinery requests this of the appropriate node manager. The recursive process of rule application bottoms out when no rule in the rule set causes a transformation of the architecture.

The effects of the rules which match in this case (10-13) are illustrated in figure 7.5. The left side of the figure shows the conceptual architecture at the completion of (9). The right side of the figure shows the corresponding implementation level. Each box on the right side of the figure represents a different point in time, with time increasing downwards. The conceptual level representation is unaffected by rule application.

The arrival of message (4) will cause the rule NewConnector to match (10). NewConnector is a shadow evolution rule, which matches when a conceptual level connector has no implementation at all. It provides an initial implementation which is

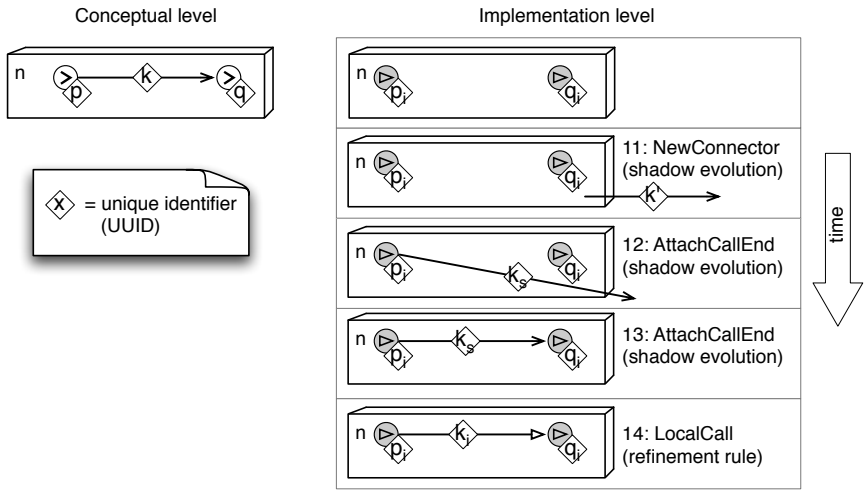


Figure 7.5: Rule application for the rules matched in figure 7.4

simply a copy of the conceptual level connector, added to the implementation architecture. Here, it matches the conceptual level call connector k and responds by creating an implementation level shadow, k_s.

The next rule to match is AttachCallEnd, which is another shadow rule. AttachCallEnd matches conceptual level calls which are connected at either the source or target end, but whose current implementations not connected. It transforms the implementation level by attaching the current (shadow) implementation of the call connector to the implementation of the element to which the conceptual call is attached. Here, on its first match it connects the shadow call k_s to the implementation of p, which is a LocalSource port p_i (11); on its second match it does the same thing at the target end of k_s with LocalTarget q_s (12). At this point we have a partially refined architecture that is a correct refinement of the current conceptual architecture.

The final rule to match is LocalCall, which is exactly the refinement specified in figure 5.9(a) on page 111. LocalCall has the effect of replacing the shadow call connector k_s with a local connector k_i from port p_i to port q_i.

Since local connectors are real implementation level constructs, at this point the refinery sends a message to the node manager for node n, requesting that it establish a local connection from port p_i to port q_i (see (14) in figure 7.4).

The node manager maintains a mapping from UUIDs to Python objects for all elements implemented on its node, including components and ports. It finds the port objects corresponding to the UUIDs p_i and q_i, and instructs the LocalSource implementation with UUID p_i (which will be the port named "count" on the clicker component) to connect itself to the port with UUID q_i (which will be the default call target port on the count component) (15).

This creates the initially-requested connection. The final step, as discussed in section 6.4.5, is that the "count" port on the clicker component invokes the clicker component's `on_count_connect` method so that the clicker can take any appropriate action (16).

This completes our discussion of the dynamics of the fiia runtime system. In the two subsections that follow we provide more detailed descriptions of the core components of the fiia runtime: the architecture, the refinery, and the refinement rules.

7.2.3 The architecture component

The core of the fiia runtime is the architecture component, which maintains the current model of the system's conceptual and implementation architectures. The architecture is stored as a triple graph using a modified version of the World Wide Web Consortium (W3C) Resource Description Framework (RDF) [85] format. Each triple is of the form (subject, predicate, object), where a subject is a workspace element, an object is a workspace element or an attribute, and a predicate indicates a relation between a subject and an object. For example, to represent the fact that a store count is anchored to a node n in a workspace w, we would use the triples shown in figure 7.6.

In the figure, `w`, `n` and `count` are UUIDs, `Workspace`, `Node` and `Store` are pre-defined types, and `is_a`, `is_in` and `anchored_to` are pre-defined relations. The architecture provides all of the types and relations defined in chapters 4 and 5, plus a few additional ones used for internal purposes. The types are organized in a type lattice which is also stored in the triple store; for example:

```
(w, is_a, Workspace)
(n, is_a, Node)
(n, is_in, w)
(count, is_a, Store)
(count, is_in, w)
(count, anchored_to, n)
```

Figure 7.6: Triples representing the fact that a conceptual store count is anchored to a node n in a workspace w.

```
(Store, subtype_of, ConceptualComponent).
```

is an entry in the triple store specifying that the type `Store` is a subtype of the type `ConceptualComponent`. The type lattice is used to simplify the expression of some of the refinement rules.

Maintaining all architectural elements and their relationships and attributes in a triple store allows for arbitrary queries to be performed on the architecture, in a uniform manner. The architecture supports queries using a slightly modified version of the SPARQL graph query language [105]; an example query is discussed in section 7.2.4.

The implementations of the triple store and the SPARQL query engine are heavily modified versions of the RDFLib Python library for RDF processing [75].

7.2.4 The refinery and refinement rules

As discussed above, the refinery iteratively applies a list of refinement rules to the architecture in response to change notifications provided by the architect. When a rule matches and transforms the architecture, this may lead to recursive rule application on the transformed elements. The recursion bottoms out when a round of rule application results in no further modifications the architecture.

The refinery itself has no knowledge of refinements all. Rather, each refinement rule is coded in its own class supporting a match-transform-realise protocol and the refinery has a list of such classes. To apply a rule to the architecture, the refinery

Creating and applying a rule in the refinery (simplified):

```
rule = architecture.refine_with(LocalCall(uuid))
if rule.matched:
    rule.realise(self)
```

The architecture's `refine_with` method:

```
def refine_with(self, rule):
    rule.match(self)
    if rule.matched:
        rule.transform(self)
    return rule
```

Figure 7.7: Application of refinement rules in the refinery and the architecture.

creates a new rule instance, passing the UUID of the changed element as a parameter to the rule's constructor (this is called `self.start` inside the rule object). It then calls the architecture's `refine_with` method with the rule instance as the parameter. The architecture attempts to match the rule and returns the rule object for further processing by the refinery. This is illustrated in the first line of figure 7.7.[3]

The complete body of the architecture's `refine_with` method is shown in the lower part of figure 7.7. The architecture calls the rule's `match` method, passing itself as the parameter. In the `match` method the rule attempts to find a match for its precondition in the current architecture's graph. If it finds a match it stores the UUIDs of the matching elements as instance attributes and sets its `matched` property to `True`. (An example of a rule implementation is provided in figure 7.8 and discussed below.)

If the rule did match, the architecture will then call the rule's `transform` method, again passing itself to the rule as the parameter. The `transform` method will modify the architecture such that the rule's postcondition is satisfied. Finally, the architecture returns the rule to the calling context, in this case the refinery.

When the rule object is returned to the refinery its `matched` property is checked.

[3]The top part of the figure is slightly simplified for presentation purposes; in the real refinery implementation the rule creation and call to `refine_with` are actually embedded in a loop that traverses all the refinement rules and provides exception handling.

If the value of matched is True, the refinery calls the rule's realise method, in which the rule may notify the refinery of further elements against which rules should be applied, or may request that the refinery communicate with one or more node managers to realize an actual implementation level change.

A slightly simplified version of the LocalCall refinement rule is illustrated in figure 7.8, along with the graphical specification of the refinement it implements, which is copied from figure 5.9(a).

The match method determines whether or not this rule matches in the current architecture, starting from the UUID (self.start) that was passed to it on rule creation.

The heart of the match method is a SPARQL query, which is defined in the graph.Pattern object called where. In the pattern, all strings beginning with a question mark in are unbound match variables; self.start is the element from which the match begins; and is_a, source_of, target_of and instantiated_on are relations with the obvious meanings. This pattern will match if self.start is a Call, and has a ?source and a ?target, both of which are instantiated_on a common ?node.[4] The correspondence between the query pattern's variables and the graphical specification is indicated at the top of the figure.

The pattern is applied against the architecture using the query_object method, which returns a result object. If the result object's ?source entry is non-null (*i.e.*, has been bound by the query), then the pattern found a match. In this case the LocalCall stores the UUIDs of the matching source, target and node, and sets its matched attribute to True.

As explained above, if there is a match then the transform method will subsequently be called with the architecture as a parameter. In the LocalCall rule the required transformation is simple: the source is connected to the target with a local call; the fact that the new local call is the implementation of the original call's conceptual counterpart is recorded, and the call is erased from the architecture.

[4] Only implementation level entities may be subjects of the instantiated_on relation, so we need not explicitly check that the source and target are implementation level elements.

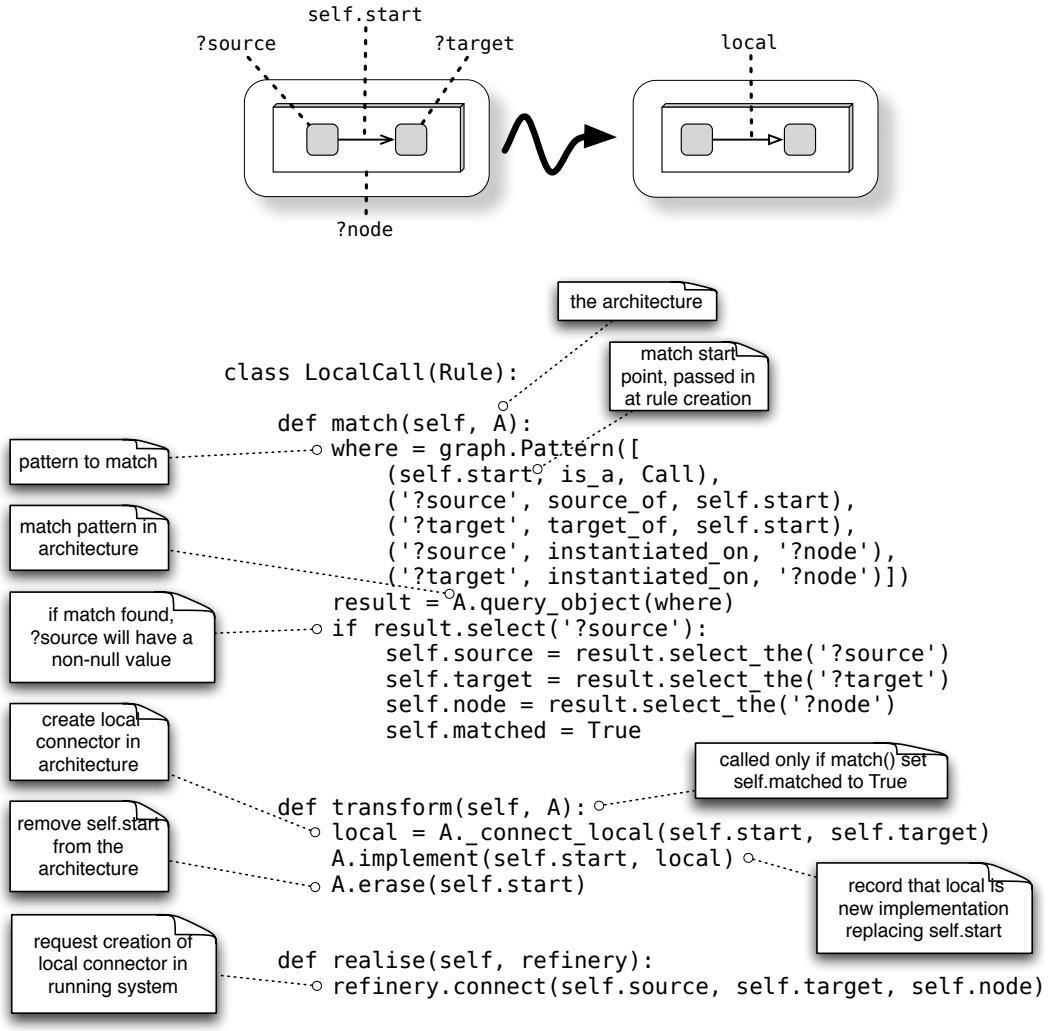


Figure 7.8: The graphical specification and Python implementation of the LocalCall refinement rule, from figure 5.9(a).

Finally, on return to the refinery the rule's `realise` method is called. This simply requests that the refinery have the appropriate node manager implement the required connection. Here there are no modified or newly introduced partially-refined entities on which rules might be applied. If there had been, a call like

```
refinery.implementation_change(self.component)
```

would be included in the `realise` method to notify the refinery that rule matches should be attempted starting from the `self.component` element.

7.3 Implementation level view of the fiia runtime

We turn now to an implementation level view of the fiia prototype, along with a discussion of selected lower-level implementation issues.

As mentioned in section 7.1, our prototype version of the fiia runtime provides a single architecture component which is instantiated on a distinguished node called the *main node*.[5] An arbitrary number of *thin nodes* may be connected to the main node via TCP/IP. Each node is an independent Python interpreter running in its own operating system level process. Each node is a member of some workspace; a thin node may belong to the same workspace as the main node or a different one; a node's workspace is determined during its initialization.

An implementation-level view of the fiia prototype is shown in figure 7.9. At the top of the figure is the main node, which includes the architecture and its associated architect and refinery components. One thin node is shown at the bottom of the figure; other thin nodes would be connected to the main node in the same fashion.

Each node includes a node manager and a node console, plus all infrastructure components necessary for correct implementation of the workspace semantics. In addition to the connections shown, each node manager can communicate with all other node managers; this is necessary for such things as moving a component from one node to another.

[5] The eventual intent is that there would be one or more main nodes per workspace, with a mechanism allowing workspaces to discover one another or become aware of one another via out-of-band signalling; however, our prototype has not yet reached this stage.

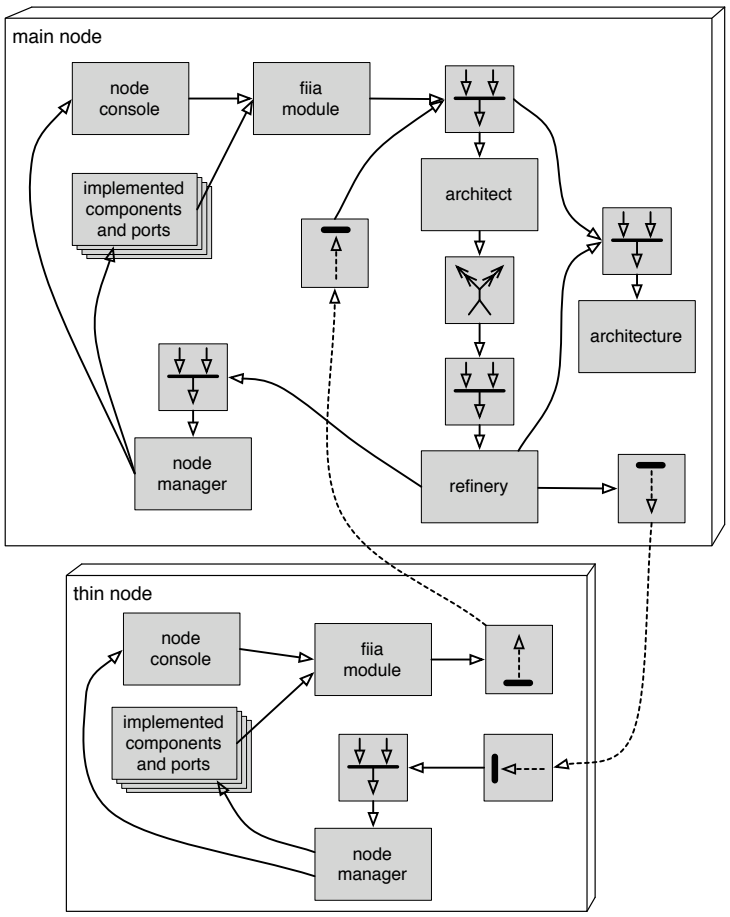


Figure 7.9: The implementation level architecture of the fiia runtime system, showing the main node and one thin node.

One component not discussed in section 7.2 but appearing in the implementation of each node is the fiia module. This is a normal Python module that is accessible to any component or script by means of a normal Python module import (it is automatically available in the node console). The fiia module provides uniform global access to fiia's classes and evolution operations via its connection to the architect component. This connection is established at node initialization, and is direct in the case of the fiia module on the main node, and remoted for the fiia modules on thin nodes.

The fiia module also maintains a record of its own node and workspace UUIDs, which are provided as over-ridable defaults wherever these are required in evolution

operations.

The fiia module also automatically adds a `preferred_node` attribute to any call to `fiia.Component`. Except in the case of components that have been explicitly anchored, this attribute is used by the refinery's component instantiation rules as a hint regarding the node on which a component should be instantiated.

7.3.1 Communications architecture

A naïve implementation of the Workspace model might use one TCP/IP stream for each transmitter-to-receiver connection. For any reasonably complicated architecture this approach would use an inordinate amount of operating system resources, especially since most of the streams would be idle most of the time.

The fiia prototype instead establishes one bidirectional TCP/IP stream between each pair of nodes. This is managed by a communications subsystem that is not represented in figure 7.9—in effect, this represents the implementation of the implementation of fiia interprocess communication.

Python's standard `ThreadingTCPServer` mechanism is used to listen for incoming connection requests from other nodes. On the receiving end, the actual connection is managed by a `NodeRequestHandler`, which is effectively an actor component with its own thread. On the sending end there is a `NodeClient`, which is a reactor that transmits messages on request. The `NodeClient` and `NodeRequestHandler` communicate using a custom request-reply protocol.

The actual transmitter and receiver objects provided by the fiia infrastructure are therefore quite simple. A transmitter knows the UUID of its associated receiver and has a direct connection to the `NodeClient` connected to that receiver's node. When it receives a message destined for that receiver it serializes the message's contents, adds the receiver's UUID, and passes it to the `NodeClient` for transmission.

The receiving `NodeRequestHandler` uses the receiver UUID included with the message to determine message routing and forwards the message to the correct receiver. The receiver de-serializes the message and passes it on to its connected component. Return values (if any) are then serialized and passed back to the Node-

`RequestHandler`, which transmits them back to the originating node. There they are returned to the originating transmitter, de-serialized, and passed back along the chain to the originating component.

Calls arriving at transmitters block awaiting the returned values. Subscription messages are transmitted non-blocking, consistent with the asynchronous semantics of subscription connectors.

7.3.2 Threading architecture

The Workspace model is inherently multi-threaded, and so is fiia. As indicated above, each node has a minimum of one thread for its `ThreadingTCPServer` and one for each `NodeRequestHandler` connected to another node. In addition, each node has a thread for its wxPython event loop, which allows for user interaction. Finally, every message broadcaster component also provides its own thread.

At first blush, the number of threads in a fiia runtime might seem to make development of the fiia toolkit and fiia applications complex in the same way that all multi-threaded programming is complex. However, the presence of the CCCM components dramatically simplifies things: application components (including, *e.g.* fiia's architect, refinery, architecture, and node manager components) are all programmed using a single-threaded model. Only infrastructure components need make special provision for multi-threading.

The wxPython event loop thread presents special challenges. As is common in modern user interface toolkits, any modifications to the state of user interface components is required to take place in the wxPython event loop. However, a quick examination of, *e.g.* figure 6.6 on page 136 shows that the Clicker component, which is a wxPython `wx.Frame` subclass, is being called from an event broadcaster's internal thread.

The solution adopted is to provide a specialized CCCM for wxPython user interface components. This blocks any incoming method calls that aren't already in the wxPython event loop thread on a synchronized queue, then posts a message to the event thread using wxPython's `wx.CallAfter` mechanism such that the return value

is put into the queue when available. This allows the caller to unblock and return appropriately, while still satisfying wxPython's own threading model.

In our current prototype implementation of fiia there is only one thread per `NodeRequestHandler`. This means that an in-progress call from node a to node b effectively blocks any other messages from a to b until it completes. This represents a potentially serious performance bottleneck, since a slow computation can block others arbitrarily. It is also a possible source of deadlock. If a call from a to b causes a call from b to a, which in turn causes another call from a to b, there will be no thread available to handle this last call.

At the same time, one thread per message broadcaster is somewhat wasteful, in that most message broadcasters are expected to be idle most of the time. Given that Python can only generate a finite number of threads before thrashing, a complicated architecture with many message broadcasters could prove problematic.

A planned future enhancement to fiia will address these two issues by moving to a thread-pool approach where message broadcasters and messages received at `NodeRequestHandlers` will be handled by a common pool of threads. While this thread pool will necessarily be finite, it should be possible to have a large enough thread pool to handle most reasonable architectures without running out of threads.

7.4 Experience using fiia

As indicated in the introduction to this chapter, the fiia toolkit and runtime system was created mainly to aid in the development and validation of the refinement rules presented in section 5.4. This proved to be an invaluable exercise—it is easy to be sloppy when drawing diagrams; much harder to lie to running code.

The fiia runtime is not yet a production quality groupware toolkit. That said, we have used the fiia prototype to produce a number of small groupware systems. Applications developed to date include the clicker implementation presented in section 6.4, a chat application, a simple object-based drawing program, and an application for presenting PowerPoint-style presentations. This last was used by the author to present [104] at the DSV-IS 2005 conference.

The fiia system does introduce some performance overheads when compared to single-user systems or hand coded groupware applications. These come primarily from the indirection of inter-component communication through multiple infrastructure elements (local ports, CCCM's, *etc.*) and from the semantic requirement to pass all parameters and return values by value.

However, in use, the performance of fiia-based applications appears to be subjectively acceptable. In the DSV-IS 2005 presentation the author had a slide controller unit and the slides themselves on a first generation tablet PC and the presentation display on a laptop connected to a projector; the two computers communicated by means of an ad-hoc wireless network. Changing from one slide to another was virtually instantaneous, despite the requirement to transmit the contents of each slide (approximately 200 kB) from the tablet PC to the laptop as each slide changed.[6]

In addition to interactive performance, it is also interesting to consider the performance of the refinement computation itself. Figure 7.10 shows the time required to request, compute and implement refinements as the number of elements and relations in the architecture increases.

The graph shows one thousand data points gathered in one execution of a benchmark program in a single-node topology. Each data point represents the time required for refinement and implementation of a small conceptual architecture consisting of one component with one call source port and one call target port, connected by a call connector to another (previously created) component. The conceptual level fiia script to request a single instance of this architecture is:

```
current = fiia.Component(TimeTest)
fiia.connect(current, 'neighbour', prev)
prev = current
```

The timer was started just prior to issuing the component creation request and stopped when the `on_neighbour_connect` method in the newly instantiated component was called, signifying completion of the connection. Times were measured on an IBM ThinkPad T40 laptop with a 1.6 GHz Intel P4-M processor and 1.5 GB of

[6] This inefficient topology was deliberately chosen to demonstrate the system's performance.

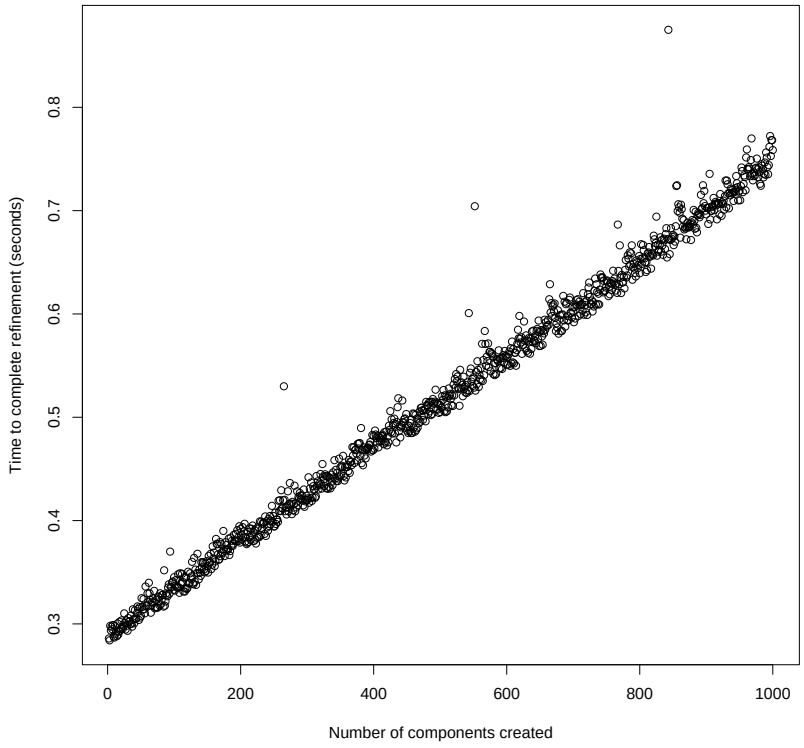


Figure 7.10: Refinement performance of fiia as the number of Workspace elements grows.

memory, running Windows XP Professional Service Pack 2, Python 2.4.2, and wx-Python 2.6.1.

As is evident from the graph, fiia's time to compute and implement refinements increases linearly with the number of elements and relations in the architecture. Profiling of the fiia runtime indicates that the majority of the system's time in this benchmark is spent executing the SPARQL queries in the rule implementations and that it is this time component that grows as the number of elements increases.

Clearly, if large numbers of conceptual elements need to be created quickly, the current fiia system's refinement performance unlikely to be adequate. However, for the kinds of systems we have built so far, which consist of small numbers of large grained components, refinement performance is acceptable. As indicated in the beginning of this chapter, thus far fiia has been implemented with a focus on correctness rather than performance; many optimizations to the system are clearly

possible.

7.5 Conclusion

The fiia prototype represents an initial, somewhat primitive implementation of the Workspace Model's semantics and dynamics. It has been extremely useful in validating and polishing the Workspace Model's refinement rules, and serves as a proof-of-concept that the Workspace Model is implementable in practice. Implementing groupware in fiia has proven to be extremely easy (the chat application was written in less than thirty minutes, most of which was spent tinkering with the user interface) and initial performance of the runtime is better than expected.

Clearly there is much work to be done before fiia will be usable as a production quality groupware system. In addition to the threading and performance enhancements mentioned above there are many other directions that could be pursued in fiia's ongoing development; some of these are discussed as future work in chapter 9.

Chapter 8

Evaluation

In section 3.1 we presented a desiderata for a model of groupware architecture, from the point of view of groupware system users, application programmers, and toolkit implementors. The Workspace Model, presented in chapters 3 through 7, has been designed to meet exactly the requirements identified in that desiderata.

In this chapter we return to the desiderata requirements (shown as boxed text in the sections below) and evaluate the Workspace Model against them. Our evaluation includes reference to examples presented in the preceding chapters, reports of our implementation experience, and proof sketches based on the formal definition of the Workspace Model from chapters 4 and 5. In some cases, particularly where we address the software engineering properties of the Workspace Model, our evaluation is necessarily tentative; these cases are indicated.

In the sections that follow we present our evaluations in the order of the desiderata, dealing first with with user requirements, then with application programmer requirements, and finally with toolkit implementor requirements.

8.1 User requirements

Users want groupware systems that allow for fluid collaboration, that are acceptably efficient, and that don't provide unnecessary surprises in use.

8.1.1 Evolution

> *The model must provide an explicit representation of dynamic change at both the conceptual and implementation levels. This must allow for user-driven reconfiguration in response to changing needs, as well as system driven reconfiguration in response to changes in the runtime environment, including partial failure.*

The conceptual and implementation level evolution calculus specified in sections 4.3 and 5.3 provide the Workspace Model's mechanism for explicitly representing runtime change.

At the conceptual level, the evolution calculus allows users to create workspaces, place nodes in workspaces and move nodes from one workspace to another, create components, anchor them to nodes if desired, create connectors, and attach connectors to components. Most important for groupware systems, it provides an explicit representation of state sharing, in the form of the synchronization relation. Naturally, all these concepts would be represented in other terms in application user interfaces; however,

As illustrated in section 6.3, the Workspace Model also provides for system driven reconfiguration in response to changes in the runtime environment including partial failure. This allows Workspace-based systems to provide for restoral or recovery as appropriate.

Our current fiia toolkit supports user-driven reconfiguration but does not yet support system-driven reconfiguration. Computing correct and efficient implementation reconfigurations in response to changing system capabilities or partial failure is a challenging issue that is discussed further in chapter 9.

8.1.2 Implementation efficiency

> *The model must allow for reasonably efficient implementations of groupware systems. To meet this, the model should not introduce unnecessary overheads and should include performance enhancing features such as caching, as well as support for a range of replica consistency maintenance approaches.*

The Workspace Model's implementation level has been designed to be as efficient as possible, given the constraints imposed by distribution transparency and the requirement to have a uniform semantics for all calls and messages. However, the model does introduce several types of implementation overhead, including:

- indirections embedded in inter-component call implementations;

- the semantic requirement to pass all parameters and return values by value;
- the possibility of unnecessary network communication (as opposed to network communication that is logically required for the functioning of the system).

It is important to note that the distinction between the conceptual and implementation levels, and the refinement process between them, is not a significant source of inefficiency. As shown in section 7.4 and discussed in section 8.3.3.3, our experience with the fiia toolkit is that actual computation of refinements is acceptably efficient; this despite the fact that fiia was not designed for efficiency. Further, once a conceptual architecture has been refined to an implementation, actual calls and messages proceed across implemented architectures without reference to the refinement rules.

The implementation level does provide support for caching and for a range of replica consistency maintenance approaches, including high-performance approaches such as operational transforms [126]. As discussed in section 7.4, experience with our fiia system suggests that the overheads introduced by the Workspace Model are not significant in at least some real applications.

8.2 Application programmer requirements

As motivated in section 1.1.2, groupware application programmers need an appropriate set of high-level abstractions that simplify the construction of groupware systems and that avoid premature commitment to a particular distributed implementation.

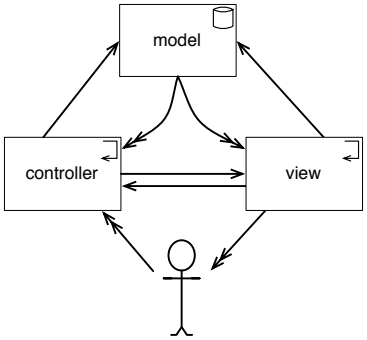
8.2.1 Conceptual expressiveness

> *The model must provide support for a range of design approaches and must be capable of modelling a range of interesting groupware systems. As a minimum, we suggest that the conceptual level should support the core architectural features identified in sections 2.1.2 through 2.1.7: separation of user interface from state, provision of intermediate layers, tree structure, a notification mechanism, collaboration through shared state, and asynchronous messaging.*

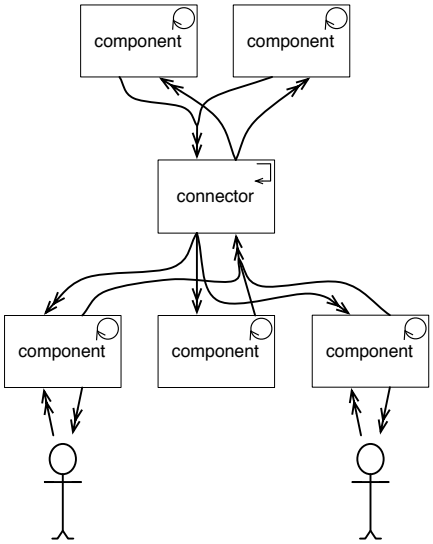
The conceptual level of the Workspace Model supports all the features identified in sections 2.1.2 through 2.1.7 of this document. Separation of user interface from application state is supported by the division of the system into components, and in particular by the location of state in store components. Intermediate layers and tree structures are possible through the attachment of components to one another in appropriate topologies (some examples are shown in figure 8.1, discussed below). The subscription connector provides the required notification mechanism as well as the capacity for collaboration through asynchronous messaging. Finally, the synchronization connector and its replicated and centralized implementations provide the collaboration through shared state.

Models of several groupware systems are been presented elsewhere in this paper using the Workspace Model, ranging from our very simple “clicker” application to the more complicated CASE tool scenario. The Workspace Model was also used to illustrate a variety of distributed system implementations of groupware systems in section 2.2. The model has been used by others in the design of an immersive exercise-based game, called “Life is a Village” [56]. Finally, as discussed in chapter 7, the Workspace model has been used to design and implement the fiia system, as well as a number of small groupware programs within the fiia system. All of these uses speak for the expressiveness of the model from an application developer’s perspective.

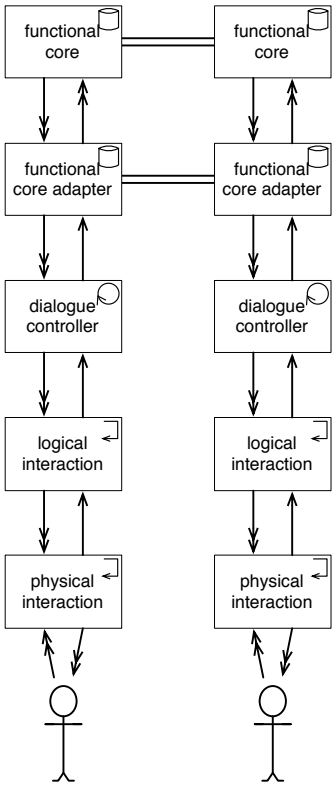
As further evidence that the Workspace Model allows designers to represent a wide range of conceptual architectures, figure 8.1 illustrates several well-known



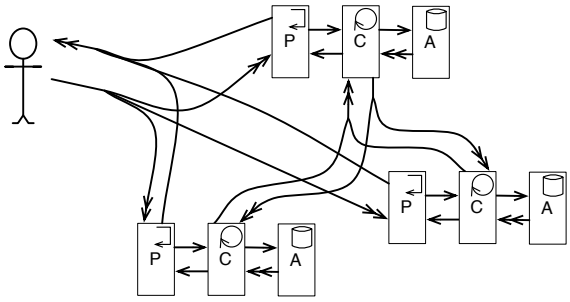
(a) model-view-controller (MVC)



(b) groupware C2



(c) groupware Arch



(d) presentation-abstraction-control (PAC)

Figure 8.1: Workspace depictions of well-known architectural styles.

architectural styles, re-expressed as Workspace conceptual architectures.

Figure 8.1(a) shows the MVC architectural style, per Krasner and Pope's original formulation [74]. The controller and view are both modelled as reactors, each able to directly call the other and the model; the model is a store that provides change notifications as asynchronous messages on its outgoing subscription connector.

Figure 8.1(b) shows a groupware example in the C2 layered style, with two layers of components and one connector. C2 components are actors with distinct "top" and "bottom" sides, which communicate via asynchronous messaging. The C2 connector is component-like [128]; here the connector is modelled by a stateless reactor that routes messages from its top to its bottom and vice versa, connected to the components above and below it by subscriptions.

Figure 8.1(c) shows an Arch-style five-layer architecture [130] that has been "unzipped" as suggested in the style suggested by Dewan's generic model of groupware architecture [37]. In this particular architecture, the top two layers are shared, while the bottom three layers are separate for each user.

Finally, figure 8.1(d) shows a tree-structured PAC architecture. The PAC style does not explicitly specify the thread model of agent facets, nor how intra-agent and inter-agent communication is to be accomplished. Here we have somewhat arbitrarily designated the presentation as a reactor, the control as an actor, and the abstraction as a store. Communication between the three facets of an agent is by calls, except in the case of notifications from the abstraction; communication between levels of the hierarchy is via asynchronous messages.

As with implementation level expressiveness, we acknowledge that we have applied the Workspace Model's conceptual level to the design of a relatively limited number of systems. Further experience, particularly with larger and more complicated systems, will be required to determine whether it is truly expressive enough to adequately model a wide range of applications.

8.2.2 Distribution transparency

> *It must be possible to represent design time components and their runtime architectural configurations without unnecessary reference to how those components or configurations are to be implemented in a distributed system.*

A designer using the Workspace Model operates at the level of component and port definitions, and of sequences of conceptual level evolution operations that compose components and their ports into systems using connectors. All of these features are distribution transparent, except where exposing distributed system issues to support user visible system features.

As illustrated in section 6.4, a programmer designing a Workspace conceptual component is concerned with the component's internal data structures and algorithms and with the component's communication with other components via its ports. The designer does not know, and need not know, whether any other components in the system will be local, remote, or even mobile. The pass-by-value semantics of Workspace calls and messages ensure that local and remote components will always behave identically. Since the system does not guarantee that there will necessarily be a connector attached to any call port, the designer is required to provide exception handling in the event that a call on a call port fails. However, this is independent of whether the failure results from an incompletely configured conceptual level or a distributed system partial failure [139].

All conceptual level evolution operations are also distribution transparent except for the anchor and float operations, which explicitly refer to nodes. An earlier version of the Workspace Model lacked anchor and float for exactly this reason. However, it appears evident that at runtime users sometimes really do care on what computer particular software components are instantiated. Examples include the desire to control the display device on which a user interface component appears, and the desire to control the location of data on portable computers prior to a planned disconnection from a network. The anchor and float operations need only be used in

cases like this.

In sum, the conceptual level is almost, but not entirely, distribution transparent. We argue that the few places where the distributed system level has been allowed to appear at the conceptual level are both well-justified and appropriate.

8.3 Toolkit implementor requirements

Toolkit implementors need to provide programming interfaces and runtime systems that support the application programmer and user needs listed above. In order to do this, they need a formal representation of the syntax and semantics of the systems they are implementing, an appropriate range of distributed system implementations to choose among, and a mechanism that allows for the efficient incremental computation of distributed implementations in response to changing user needs and changing system environments.

8.3.1 Formal representation

> *The architectural representations at both the conceptual and implementation levels must have well-defined syntax and semantics, in order to allow precise representation of evolution, at either level, and of the mapping between levels.*

The syntax of the Workspace Model consists of its workspaces, nodes, components, ports, connectors, and the allowed relationships between them, which are specified in chapter 4 for the conceptual level and chapter 5 for the implementation level. The allowed relationships at each level are defined by the evolution calculus. If we start with an empty architecture at either level and systematically apply evolution operations to it, we can (in principle) generate the entire space of syntactically correct architectures.

The semantics of the Workspace Model's elements are defined precisely but informally in chapters 4 and 5. The semantics at the conceptual level were chosen to match the important core concepts for the design of groupware that were described

in chapter 2 (see section 8.2.1 for further discussion of this point). The semantics of the implementation level are deliberately very simple and were chosen to be directly implementable in most common programming languages. A fully formal semantic model would be possible using a process algebra or similar technique; however, the value of such a formulation at the architectural level is not clear.

To the best of our knowledge, the Workspace Model represents the only formally-specified architecture for synchronous groupware that includes separate conceptual and implementation levels, as well as a formal mapping between them.

8.3.2 Implementation expressiveness

> *At the implementation level, the model must allow for replicated, centralized and hybrid approaches to managing shared state and must provide an appropriate range of implementations for any given conceptual architecture. The implementation level must be represented in a way that supports reasoning about performance, security, fault-tolerance, and other key distributed system attributes.*

The Workspace Model supports a full range of architectures for implementing shared state, including replicated, centralized and hybrid approaches. A hybrid architecture for the CASE tool example is illustrated in figure 3.4 on page 56. A range of different centralized and replicated architectures for the "clicker" example are illustrated in figures 1.4, 6.6, 6.7, and 6.8.

Support for reasoning about distributed system attributes is provided by several implementation level features. These include:

- components being located on identified nodes;
- differentiation between local and remote connectors;
- indication of where parameters and return values are copied for passage by value (local ports) and serialized for network transmission (transmitters and receivers);
- indication of the threading model for delivery of asynchronous messages (message broadcasters);

- indication of the locations where concurrency control is applied and replica consistency maintenance algorithms enacted (CCCMs); and
- indication of how multi-cast message transmission is achieved (either using message broadcasters or channel implementations).

Fully understanding attributes like performance requires an understanding of such things as the replica consistency maintenance algorithms enacted by CCCMs, the message ordering policies provided by channels, and the cache invalidation and refresh policies implemented by caches. These are deliberately represented outside the model; however, an understanding of distribution topologies plus those attributes is sufficient to support effective performance analysis.

Section 6.3 illustrates the Workspace Model's support for analysis of fault tolerance. As demonstrated in that section, the model's separation of conceptual from implementation levels, plus the refinement relation between the two levels, allows for precise reasoning about recovery and restoral from system failure, and the relationship between a current implementation and the users' intent.

While we are satisfied with the expressiveness of the implementation level, we acknowledge that we have applied it to the development of only one runtime system (fiia) and a relatively small suite of applications. Further use and experimentation will be required to determine whether it is adequate for a broader set of applications or whether it will need to be extended.

8.3.3 Refinement

> *The model must provide a formal relation (refinement) between the two architectural levels such that implementation architectures may be derived automatically from conceptual architectures. In order to be useful and implementable, the refinement relation must refine all possible conceptual level architectures to fully refined architectures, and the computation of a refined architecture must be tractable for large architectures.*

The formal definition of the refinement relation from the conceptual to the implementation level is provided in section 5.4 and its use to generate implementation-level architectures is illustrated in sections 1.4.2, 3.4, and 6.3.

For refinement to be useful, it must always be possible to refine a syntactically correct conceptual architecture to a fully refined implementation architecture by repeated application of the refinement rules. This property will hold if:

- refinement terminates (termination); and
- at termination, the architecture is fully refined (completeness).

In sections 8.3.3.1 and 8.3.3.2 we provide proof sketches of the termination and completeness properties. We follow this in section 8.3.3.3 with a proof that the computation of a single refinement (as opposed to the computation of the full refinement relation) is tractable.

In addition to these proofs, our implementation of the refinement mechanism in our fiia toolkit, illustrated in section 7.2.4, gives us further confidence that the refinement relation has been appropriately specified for use in a toolkit.

8.3.3.1 Refinement termination

We sketch the proof of termination as follows:

For a finite conceptual architecture c, the complete refinement relation may be computed as follows. Call the set of refinement rules Q. For each rule $r \in Q$, find all distinct subgraphs in c where the left hand side of r matches. At each match, apply r to c producing a partially refined architecture c'; let C_1 be the set of all c' so produced. Repeat the process recursively for all $c' \in C_1$ and all $r \in Q$, producing a tree of architectures. The process ends for a particular architecture c_n when no rule $r \in Q$ matches in c_n.

This algorithm will terminate if two conditions hold:

Finite Branching. At each level n in each branch of the tree, the set C_n of partially refined architectures is finite; and

Branch Termination. For all branches in the refinement tree, we ultimately reach an architecture where no rule $r \in Q$ matches.

We consider first the Finite Branching condition. Since the left hand side of each each refinement rule r contains a finite but non-zero number of architectural elements in a finite number of relations to other elements, and the initial architecture c is finite, any r can match in c only a finite number of times. And, since Q is finite, C_1 must also be finite. The same argument applies for each $c' \in C_1$ and thus for all subsequent levels of refinement C_n. Therefore the Finite Branching condition holds.

The Branch Termination condition may be proved by structural induction over the set of refinement rules Q.

The minimal structures in partially refined architectures are patterns of architectural elements that never appear in the left-hand side of a refinement rule. An architecture that consists only of minimal structures requires no refinement; therefore, its refinement trivially terminates.

The rules in Q may be grouped into several categories.

Removal rules. Some rules remove non-minimal structures from the architecture and are therefore monotonic towards minimal structures. These are rules 5.7(c) and (d),[1] 5.8(e) and (f), 5.9(e) and (f), 5.10(a), (e) and (f), 5.11(a), (d), and (e), 5.12 (b), and 5.13(e).

Non-minimal transformations. Some rules transform non-minimal structures into minimal structures. These are rules 5.7(b), 5.8(a) through (c), 5.9(a) through (d), and 5.12(a).

Addition of minimal structures. Rules 5.13(a) and (b) add new minimal structures (channel endpoints) to the architecture and also reroute call connectors from channels to the channel endpoints. Since these rules only match where a call connector is attached to a channel, and each rule application removes such an attachment, the rules may only be applied a finite number of times. And,

[1] For convenience, in this section we refer to rules directly by their figure numbers.

since removing call connectors from channels eventually allows the removal rule 5.13(e) to match, these rules are monotonic towards minimal structures.

Relation modification. Some rules change relations.

- Rule 5.7(a) anchors floating components. Since only anchored components may be implemented (an implemented component is a minimal structure), and there is no other rule which floats anchored components, this rule is monotonic towards minimal structures.
- Rule 5.13(c) and (d) reroute call connectors from channels to channel endpoints. Since there is no rule that reroutes call connectors from channel endpoints to channels, channel endpoints are minimal structures, call connectors attached to channel endpoints are refined to minimal structures by rules 5.9, and removing call connectors from channels eventually allows the removal rule 5.13(e) to match, these rules are monotonic towards minimal structures.

Addition of non-minimal structures. Some rules add new non-minimal structures to the architecture.

- Rule 5.8(d) replaces a subscription source port with its implementation. However, the only rule that matches a subscription source implementation is the removal rule 5.8(e), so this rule is monotonic towards minimal structures.
- Rules 5.10(b) through 5.10(d) add new call connectors to the architecture. Since each call connector added removes an attachment between an implemented component and a subscription connector, these rules can only match a finite number of times. And, since removing attachments from subscription connectors eventually allows removal rule 5.10(e) to match, these rules are monotonic towards minimal structures.
- Rules 5.10(b) and 5.11(b) add new channels to the architecture. However, channels may only be added to subscription connectors and synchronization connectors which do not have channels attached to them already, so

these rules may only be applied a finite number of times.

- Rule 5.11(c) replaces a connection between a CCCM and a channel with two call connectors and two local ports. Since this replacement may only be done once for each CCCM to channel connector, this rule may only be applied a finite number of times.

It is also important that the rules in category 5, which introduce new non-minimal structures into the architecture, are not mutually recursive. (If they were, it would be possible for the refinement algorithm to be trapped in an endless loop.) By inspection, we see that these rules add only new call connectors, new channels and new ports, and that in no case do these elements replace another of the elements on this list, so there is no mutual recursion.

In summary, each refinement rule either is monotonic towards minimal structures or may be applied only a finite number of times in a given architecture. So, repeated application of refinement rules to any branch of the refinement relation will result in only minimal structures. Therefore the Branch Termination condition holds.

Since both the Finite Branching and Branch Termination conditions hold, there is a provably terminating algorithm for computing the refinement relation.

8.3.3.2 Refinement completeness

Refinement completeness requires that all architectures c_n for which no $r \in Q$ matches are fully refined (or *ground*) architectures. A ground architecture is one that contains only workspaces, nodes, and implementation level components, connectors and ports.[2]

We sketch a proof of refinement completeness as follows:

The refinement relation is complete if the following two conditions hold:

Initiation. All non-ground syntactic forms that can appear in a conceptual level architecture are subject to one or more refinement rules.

[2]Recall that people and subscription connectors between people and components are allowed in Workspace diagrams but are not part of the formal notation and not subject to refinement.

Progress. All refinement rules either generate ground architectures, or generate partially refined architectures all of whose non-ground elements are subject to further refinement.

To prove the Initiation condition, we must consider all syntactically correct conceptual level architectures. Syntactically correct conceptual level architectures are defined by the conceptual level evolution calculus of section 4.3. Examining this calculus, we identify the following categories of non-ground syntactic forms:

Conceptual components, which may provide ports, which are always contained within workspaces, and which may be either floating or anchored to nodes.

- Components in workspaces that contain no nodes cannot be directly instantiated. The are removed from the architecture by rules 5.7(c) and (d). The one exception are stores that are members of synchronization groups containing other stores; these are refined by rule 5.11(a).
- Components that are floating in a workspace that contains nodes may be anchored to any of the nodes by rule 5.7(a).
- Components that are anchored to a node will be instantiated by rule 5.7(b).

Ports, which are always on the surface of conceptual components. As shown above, conceptual components are always either instantiated on nodes (in which case their ports are moved to the instantiated components), or removed along with their ports.

- Attached and unattached call ports and subscription ports on instantiated components are refined by rules 5.8(a) through (d). Call ports and subscription target ports become local ports. Subscription target ports become local ports connected to message broadcasters, with any outgoing subscriptions attached to the message broadcaster.
- Synchronization ports that are attached to synchronization connectors are refined to a pair of local ports by rule 5.11(b).
- Unattached synchronization ports are removed from the architecture by rule 5.8(f).

Call connectors, which may be attached at their source end, target end, or both, to call ports. As shown above, call ports are always ultimately refined to local ports.

- Call connectors lacking either a source or a target are removed from the architecture by rules 5.9(e) or (f).
- Call connectors having both a source and a target are refined by rules 5.9(a) through (d).

Subscription connectors, which may have zero or more sources and zero or more targets, all of which will be subscription ports. As shown above, subscription source ports are refined to message broadcasters (from the perspective of the connector) and target ports to local ports.

- Subscription connectors with no sources are removed from the architecture by rule 5.10(f).
- Subscription connectors with at least one source are refined by rule 5.10(b).

Synchronization connectors, which may have zero or more attached subscription ports.

- Synchronization connectors with no attached synchronization ports are removed from the architecture by rule 5.11(e).
- Synchronization connectors with at least one connected synchronization port are refined by rule 5.11(e).

Since all syntactic forms that may be found in conceptual level architectures have refinements, the Initiation condition holds.

To prove the Progress condition, we consider each of the refinement rules in turn. For the progress condition to hold, either the right hand side of the rule must be a ground architecture, or it must consist only of sub-architectures that are guaranteed to be subject to further refinement.

Table 8.1 summarises the refinement rules and the rules that match their right-hand sides. In some cases the way in which the right-hand sides are matched in by

Rule	Rules matching right-hand-side (or ground)
5.7(a) 5.7(b) 5.7(c) and (d)	5.7(b) 5.8(a), (b), (c), (d), and (f), and 5.11(a), (b), and (c) ground
5.8(a) 5.8(b) 5.8(c) 5.8(d) and (e) 5.8(f)	5.9(a) through (e) 5.9(a) through (d) and (f) 5.10(d) 5.10(a), (b) and (c) ground
5.9(a) through (f)	ground
5.10(a) 5.10(b), (c) and (d) 5.10(e) 5.10(f)	5.10(b), (c), (d), and (e) 5.9(a) through (f) and 5.10(c), (d), and (e) 5.13(e) ground
5.11(a) 5.11(b) 5.11(c) 5.11(d) 5.11(e)	5.8(a), (b), (c), and (d), and 5.11(b) 5.11(c) 5.11(c) and (d) 5.13(e) ground
5.12(a) and (b)	5.9(a) through (f)
5.13(a) and (b) 5.13(c) and (d) 5.13(e)	5.9(a) and (c), 5.10(c), (d), and (e), and 5.11(c) and (d) 5.9(a) through (d), 5.10(c), (d), and (e), and 5.11(c) and (d) ground

Table 8.1: Summary of the Progress property of refinement.

other rules is straightforward; however, other cases require more careful consideration. These are addressed below.

Conceptual level components and ports. As shown in the discussion of the Initiation condition, (starting on page 191), conceptual components are always refined to implementation components and conceptual ports are always refined to local ports, plus message broadcasters in the case of subscription source ports.

Generic connectors. In rules 5.8 (a) through (e), the connectors shown on the right-hand side are generic connectors, which may be either implementation-level (in which case they are ground and require no further consideration) or conceptual level. These latter are addressed below.

“Dangling” conceptual connectors. In rules 5.8(a) through (e), 5.10(a) through (d), 5.11(a) through (c) and 5.12(a) and (b), conceptual connectors are shown with “dangling” ends, which may or may not be attached to (unseen) components

or ports. For the Progress condition to hold, it is essential that all such connectors be subject to further refinement. We consider the three connector types in turn.

Call connectors. If a call connector lacks either a source or a target it will be removed from the architecture by rules 5.9(e) and (f). If it has both a source and a target, then these may be either conceptual or implementation level. If either end is conceptual level, it must be a component or a port; as proven above, conceptual components and ports are eventually refined to implementation level. If both ends are implementation level, the call connector is refined by rules 5.9(a) through (d).

Subscription connectors. Subscription connectors with no targets are refined at their source ends only. If a subscription connector has one or more conceptual level targets, these must be subscription target ports, attached by the evolution in figure 4.12(d). As shown above, such ports are always refined to local target ports, thereby allowing rule 5.10(d) to match. If a subscription connector has no sources it will be removed from the architecture by rule 5.10(f). If it has conceptual level sources, these must be one of the following:

- Subscription source ports, attached by the evolution in figure 4.12(c)). As shown above, these are always refined to message broadcasters (from the perspective of the subscription connector), which allows rules 5.10(b) and (c) to match.
- Channels, attached by rule 5.10(b). In this case, rule 5.10(c), (d) or (e) will match.

Synchronization connectors. Synchronization connectors may be attached to subscription source ports by the evolutions in figures 4.12(e) and (f). In this case, they will be matched by rules 5.7(d) or 5.11(a), (b), (c) or (d). Synchronization connectors may also be attached to channels, by rule 5.11(b). In this case they will be matched by rules 5.11(c) or (d).

The remainder of cases are straightforward. It is therefore clear that all refinement rules either generate ground architectures, or generate partially refined architectures all of whose non-ground elements are subject to further refinement. The Progress condition is therefore satisfied. In conjunction with the Initiation condition, this proves that the refinement relation is complete: that is, that all architectures in which no further rules match are ground architectures.

8.3.3.3 Tractable computation of refinement

In addition to termination and completeness, the use of the refinement relation in a toolkit requires that computation of refinement be tractable. However, computing the *entire* refinement relation using the algorithm sketched in the proof of termination would in fact require a calculation exponential in the number of workspace elements and relations and would therefore be intractable for large architectures.

Fortunately, in practice it is never necessary to compute all possible refinements of a large architecture. In many cases, a single refinement will be adequate. In others, such as in the case of determining an appropriate strategy for a failure recovery or restoral, it is necessary to compute only a small set of candidate refinements for a bounded sub-graph of the complete conceptual architecture.

Computing a single refinement is linear in the number of elements and relations in the architecture. Consider a conceptual architecture c. Application of a single rule r to c will result in either a match failure, in which c is unchanged, or a match, which will result in a partially-refined architecture c'. A simple approach to generating a fully-refined architecture is therefore the following, which computes a single refinement branch:

1. Begin with the conceptual architecture c.
2. Iteratively apply each of the individual rules to c until some rule r matches, generating c'.
3. Repeat from step 1, replacing c with c'.
4. When no rule r matches in the current architecture, refinement is complete.

As argued above, all refinements either are monotonic towards minimal structures or may be applied only a finite number of times, where that number is defined by a number of relations. Therefore computation of a single refinement is linear in the number of elements and relations in the conceptual architecture.

Our implementation of refinement in the fiia runtime, discussed in section 7.2.4, uses exactly the approach explained above above. It considers only bounded subgraphs, defined by the scope of a single conceptual level change. Our experience with fiia shows that this approach is both effective and efficient.

8.4 Conclusion

In sum, the Workspace Model meets all of the significant requirements identified in the desiderata of section 3.1 and introduces a number of new ideas that should be considered for inclusion in other modeling languages or systems. In the next and final chapter we identify areas for future enhancement of the model, its implementation, and its use for the construction of interesting groupware systems.

Chapter 9

Conclusion and Future Directions

We have presented the Workspace Model, a formally-defined architectural model for groupware that provides a clean separation between conceptual structure and distributed implementation. The Workspace Model includes an evolution calculus that allows the formal description of architectural change at runtime. It supports both user-driven conceptual change and system-driven implementation change. The two levels of architectural description are linked by a refinement relation, which allows the automatic generation of the set of implementation architectures corresponding to any given conceptual architecture.

As demonstrated in chapter 8, the Workspace Model possesses the essential properties required of a dynamic architectural model for synchronous groupware. However, this is not to say that the model is in any sense a perfect or ultimate model of groupware architectures.

In this chapter we present suggestions for future work that arise out of the model in its current form. We divide the presentation into three sections: validation of the model as a software engineering tool; enhancements to the model itself; alternate programming approaches; issues arising from implementation of the model; and groupware interface issues arising from the model's semantics.

9.1 The Workspace Model as a software engineering tool

One of our intents in developing the Workspace Model was that it would provide an appropriate set of abstractions for the development of groupware systems. The modelling examples presented in chapter 6, the completed applications discussed in chapter 7, and the design of the fiia runtime system itself all suggest that we have succeeded in this intent. Also suggestive are the use of the Workspace model to design an immersive, three dimensional, exercise based game [56] and the fact

that the Workspace Model has been used successfully in an undergraduate software architecture course at Queen's University.

However, thoroughly validating the Workspace Model from a software engineering perspective will require considerably more development experience with significant groupware applications. Some of the specific open questions are:

- Does use of the Workspace Model by real groupware designers actually provoke appropriate questions at appropriate points in the development life cycle, thereby leading to more appropriate designs?
- Does the Workspace Model scale well as a design and implementation representation, both in terms of the development of large systems, and in terms of its use by large numbers of developers?
- Is groupware designed using the Workspace Model easier and less expensive to implement than groupware developed using other approaches?
- Is groupware implemented using the Workspace Model easier and less expensive to enhance and maintain than groupware developed using alternate approaches?

As with many issues in software engineering, actually answering these questions poses considerable practical difficulties.

9.2 Enhancements to the Workspace Model

There are many possible extensions and enhancements to the Workspace Model. Here we highlight a few of these.

Nested components. In its current form the Workspace Model supports a flat component model that does not allow components to be defined as nested structures of lower level components. This is a useful feature that is found in many other architectural models. An earlier version of the Workspace Model did include nested components; however, their presence significantly complicated the formal definition of evolution and refinement rules and this feature was

temporarily abandoned. Now that the flat version of the model has been fully defined and validated, both through proof and prototype, it would be interesting to re-open this area of investigation.

More detailed architectural representations. The Workspace Model currently provides no mechanism for explicitly representing replica consistency maintenance algorithms, caching algorithms, or channel ordering policies. A formal mechanism for incorporating this information into the model would be useful.

Further distributed implementations. The Workspace Model does not currently include refinements that automatically generate such features as clusters, federations, partial replication, and "hot-standby" servers. Since these features are widely used in commercial groupware systems, their inclusion in the model should be investigated.

Representation of internal component state. In the current model an implemented component that has acquired interesting internal state is indistinguishable from a freshly-created component. This means that some combinations of evolution and refinement that are structurally correct in fact discard state that might be interesting to the user. Ensuring that this doesn't happen (as we do in the fiia runtime) currently requires this fact to be represented outside the model.

Dynamic adaptation to preserve quality attributes. The intelligent selection of an implementation architecture for a conceptual architecture may require balancing such quality-related concerns as device and network capabilities, performance requirements, security and privacy concerns, and required levels of fault tolerance in different failure scenarios. All of these concerns (and others) may change dynamically at runtime. A completely adaptive Workspace Model would provide explicit representation of these concerns as well as a mechanism for reasoning from them to a choice of implementations.

Constraints. As indicated in chapter 2, several other groupware systems support the direct use of dynamically maintained declarative constraints. The addition of these to the Workspace Model should be investigated.

9.3 Security

As defined in this document, the Workspace Model makes no provision for security: in principle, any user or software component can invoke any operation in any workspace. Clearly any production quality implementation of the Workspace Model would need to take security and privacy issues into account.

Prior work based on an early version of the Workspace Model suggests that the model can provide an appropriate base for secure collaborative systems, with the addition of appropriate security provisions [129]. In particular, the execution of the conceptual level evolution calculus operations provides a "hook" for the operation of a security reference monitor.

Further investigation of security and privacy issues, including an update of [129] to account for the current version of the Workspace Model, would be appropriate.

9.4 Alternate programming approaches

Our fiia prototype offers programmers only minimal syntactic sugar over a a relatively literal interpretation of the Workspace Model's conceptual level component and connector architecture and evolution calculus. Many other programming approaches are possible, each with their own particular characteristics. An investigation of the space of such programming approaches would be useful.

One such investigation is already ongoing, in follow-on research by Christopher Wolfe. This is looking at the use of the Workspace Model to provide fully distributed groupware semantics semi-automatically to programs written in the C# language for Microsoft's .NET runtime.

9.5 Implementation issues

The development and initial use of the fiia prototype exposed a number of implementation level issues which are either outside the scope of the current model or only indirectly hinted at. These include:

Debugging support. Debugging in the fiia toolkit can sometimes be unnecessarily difficult. In our experience, the problems arise from the temporal separation between conceptual level requests and their effects at the implementation level, as well as from well-known issues with debugging multi-point asynchronous message communications (as implemented by subscription connectors). Significant research is required into effective debugging representations of these issues.

Exposing architectural knowledge inside components. It would sometimes be useful to make knowledge available in the architecture directly available to component implementations. For example, consider a distributed representation of a card table as in [66]. When a card is in a player's "hand" its value is visible to that player but not concealed from other players. We can imagine that the state of the table is maintained in a store component and that players' components may query the store. In this case, it would be useful to be able to respond differently depending which player is making the request. Naturally, the requesting player's identity could be encoded with the request itself (with appropriate cryptographic authentication, if necessary). However, this information is already implicitly represented in the architecture in the structure of components and connectors.

This kind of scenario appears to be common enough that it would be useful to make the answer to "who made this request or sent this message?" easily available at the component level.

9.6 Groupware interfaces

There are a large number of well-known user interface issues that would have to be addressed in any production quality groupware system developed using the Workspace Model. These include such things as inter-user awareness at many levels of detail, coordination mechanisms for facilitating group action, mechanisms for distinguishing between private and shared information and for moving information

between these categories, mechanisms for acquiring exclusive access to shared objects and for distinguishing who has such access, and so on.

In addition, the Workspace Model presents a number of new or extended concerns which would also have to be addressed in any real implemented system. These include:

Fine-grained session management. Most commercial groupware systems include some form of session management tool allowing users to contact one another and join into shared sessions. In the Workspace Model, sharing is at the level of synchronized stores; however, this is a programming construct that should almost certainly not be exposed directly to users—users need only care that they are working on or with "the same thing". Since sharing is possible at the individual store level, "sessions" can be relatively fine grained and fluid. Appropriate user interfaces for managing such fine grained and fluid collaboration do not yet exist, so fine-grained session management interfaces present an interesting area for further study.

Reporting partial failure. In most groupware systems, partial failure is manifested as the disappearance of one or more users from a collaborative session, or as an inability for the session to continue. However, the Workspace Model's support for fine-grained collaboration provides the possibility of relatively fine-grained partial failure. How to make such failures visible to the user, at either a system-wide or application specific level, is an interesting user interface research problem.

9.7 Summary

The Workspace Model provides a formal multi-level approach to the specification, development and runtime implementation of distributed groupware systems. Like many such approaches, it raises as many questions as it answers. We look forward to addressing these questions in the coming years.

References

[1] H. Abdel-Wahab and M.A. Feit. XTV: A framework for sharing X window clients in remote synchronous collaboration. In *Proceedings of the IEEE Conference on Communication Software: Communications for Distributed Applications and Systems* (Tricomm '91, Chapel Hill, NC, USA, April), pages 159–167, 1991.

[2] H. Abdel-Wahab, O. Kim, P. Kabore, and J.P. Favreau. Java-based multimedia collaboration and application sharing environment. In *Colloque Francophone sur L'Ingenierie des Protocoles (CFIP'99), Nancy, France*, pages 451–463, April 26–29 1999.

[3] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks* (FTCS-30, DCCA-8, New York, NY), pages 327–336, June 2000. Also available from `spread.org`.

[4] G.E. Anderson, T.C.N. Graham, and T.N. Wright. Dragonfly: Linking conceptual and implementation architectures of multiuser interactive systems. In *Proceedings of the 22nd International Conference on Software Engineering* (ICSE '00, Limerick, Ireland, June 4–9), pages 252–261, 2000.

[5] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, second edition edition, 1997. ISBN 0-201-31006-6.

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

[7] S.A. Baset and H. Shulzrinne. An analysis of the Skype peer-to-peer internet telephony protocol. In *IEEE Infocom*, 2006. To appear.

[8] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 1998. ISBN 0-201-19930-0.

[9] J. Begole, M.B. Rosson, and C.A. Shaffer. Supporting worker independence in collaboration transparency. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '98), pages 133–142. ACM Press, 1998.

[10] J. Begole, C.A. Struble, C.A. Shaffer, and R.B. Smith. Transparent sharing of Java applets: A replicated approach. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '97, Banff, Alberta, Canada, Oct. 14–17), pages 55–64. ACM Press, 1997.

[11] R. Bentley, T. Rodden, P. Sawyer, and I. Sommerville. An architecture for tailoring cooperative multi-user displays. In J. Turner and R. Kraut, editors, *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4), pages 187–194. ACM Press, 1992.

[12] T. Berlage and A. Genau. A framework for shared applications with replicated architecture. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '93, Atlanta, GA, USA, Nov. 3–5), pages 249–257. ACM Press, 1993.

[13] Y.W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, 2001. Available from `www.gamasutra.com`.

[14] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Game Developers Conference*, 2001. Available from `www.gamasutra.com`.

[15] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53,103, December 1993.

[16] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. ACM Press/Addison-Wesley, 1999. ISBN 0-201-57168-4.

[17] J. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, Queen's University, March 2004. Available from `www.cs.queensu.ca`.

[18] Mark Brockington. Client-side movement prediction. In Thor Alexander, editor, *Massively Multiplayer Game Development*, pages 293–303. Charles River Media, 2003.

[19] R. Burridge. *Java Shared Data Toolkit User Guide version 2.0*. Sun Microsystems, JavaSoft Division, 1999. Available from `http://java.sun.com`.

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons Ltd., 1996. ISBN 0-471-95869-7.

[21] Chris Butcher and Bart House. Recreating the LAN party online: The networking and social infrastructure of Halo 2. In *Game Developers Conference*, 2005. Available from `www.cmpevents.com/GD05`.

[22] G. Calvary, J. Coutaz, and L. Nigay. From single-user architectural design to PAC*: A generic software architecture model for CSCW. In *Human Factors in Computing Systems: CHI '97 Conference Proceedings* (USA), pages 242–249. ACM Press/Addison-Wesley, 1997.

[23] J.M. Carroll. *Making Use: Scenario-Based Design of Human-Computer Interactions*. MIT Press, 2000. ISBN 0-26203-279-1.

[24] A. Chabert, E. Grossman, L. Jackson, S. Pietrowicz, and C. Seguin. Java object sharing in Habanero. *Communications of the ACM*, 41(6):69–76, June 1998.

[25] G. Chung and P. Dewan. A mechanism for supporting client migration in a shared window system. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '96, Seattle, WA, USA, Nov. 6–8), pages 11–20. ACM Press, 1996.

[26] G. Chung and P. Dewan. Towards dynamic collaboration architectures. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 1–10, New York, NY, USA, 2004. ACM Press.

[27] G. Chung, P. Dewan, and S. Rajaram. Generic and composable latecomer accommodation service for centralized shared systems. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18), pages 129–147, 1998.

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

[29] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Addison Wesley, 2003. ISBN: 0-201-70372.

[30] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2000. ISBN 0201-619-180.

[31] J. Coutaz. PAC, an object oriented model for dialog design. In *Proceedings of the IFIP Conference on Human Computer Interaction* (INTERACT '87), pages 431–436. Elsevier Science Publishers B. V. (North-Holland), 1987.

[32] J. Coutaz. PAC-ing the architecture of your user interface. In *Proceedings of the Fourth Eurographics Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '97), pages 15–32. Springer Verlag, 1997.

[33] J. Coutaz, F. Bérard, E. Carraux, and J. Crowley. Early experience with the mediaspace CoMedi. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18), pages 57–72. Kluwer Academic Publishers, 1998.

[34] M. Day. What synchronous groupware needs: Notification services. 6th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VI), 1997.

[35] M. Day, J.F. Patterson, J. Kucan, W.M. Chee, and D. Mitchell. Notification service transfer protocol version 1.0. Technical Report 96-08, Lotus Workgroup Technologies, November 15 1996. Available from `http://research.lotus.com`.

[36] V.C. de Paula, G.R.R. Justo, and P.R.F. Cunha. Specifying and verifying reconfigurable software architectures. In *PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 21, Washington, DC, USA, 2000. IEEE Computer Society.

[37] P. Dewan. Architectures for collaborative applications. In M. Beaudouin-Lafon, editor, *Computer Supported Co-operative Work*, pages 169–193. John Wiley & Sons Ltd., January 1999. ISBN 0-471-96736-X.

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

[39] P. Dewan and R. Choudhary. Coupling the user interfaces of a multiuser program. *ACM Transactions on Computer-Human Interaction*, 2(1):1–39, March 1995.

[40] A. Dix, D. Ramduny, and J. Wilkinson. Interaction in the large. *Interacting with Computers*, 11(1):9–32, December 1998.

[41] P. Dourish. Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '96, Boston, MA, USA, Nov. 16–20), pages 268–277. ACM Press, 1996.

[42] P. Dourish. Using meta-level techniques in a flexible toolkit for CSCW applications. *ACM Transactions on Computer-Human Interaction*, 5(2):1–39, March 1998.

[43] E. Dubois, L. Nigay, and J. Troccaz. Consistency in augmented reality systems. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '01,* Toronto, Canada, May), Published as Lecture Notes in Computer Science vol. 2254, pages 117–130. Springer-Verlag, 2001.

[44] E. Dubois, L. Nigay, J. Troccaz, O. Chavanon, and L. Carrat. Classification space for augmented surgery, an augmented reality case study. In A. Sasse and C. Johnson, editors, *Proceedings of the 7th IFIP Conference on Human Computer Interaction* (INTERACT '99, Edinburgh, UK), pages 353–359. IOS Press, 1999.

[45] D.A. Duce, J.R. Gallop, I.J. Johnson, K. Robinson, C.D. Seelig, and C.S. Cooper. Distributed collaborative visualization – the MANICORAL approach. In *Proc. Eurographics UK Conference*, pages 69–85, March 1998.

[46] R. Dunn et al. The wxPython interface to the wxWidgets user interface toolkit. `wxpython.org`.

[47] T. Duval and L. Nigay. Implémentation d'une application de simulation selon le modèle PAC-Amodeus. In *Proceedings of the 11th Conférence francophone Interaction Homme-Machine* (IHM '99, Monpellier, France), pages 86–93, 1999.

[48] W.K. Edwards, E.D. Mynatt, K. Petersen, M.J. Spreitzer, D.B. Terry, and M.M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '97, Banff, Alberta, Canada, Oct. 14–17), pages 119–128. ACM Press, 1997.

[49] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM Conference on the Management of Data* (SIGMOD '89, Seattle, WA, USA, May 2–4), pages 399–407. ACM Press, 1989.

[50] U. Gall and F.J. Hauck. Promondia: A Java-based framework for real-time group communication on the Web. In *Proceedings of the 6th World Wide Web Conference,* (Santa Clara, CA. April 7–11). Elsevier Science Publishers B. V. (North-Holland), 1997.

[51] Gamespy. `www.gamespy.com`.

[52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[53] GoToMeeting. `www.gotomeeting.com`.

[54] T.C.N. Graham. *Declarative Development of Interactive Systems*, volume 243 of *Berichte der GMD*. R. Oldenbourg Verlag, July 1995.

[55] T.C.N. Graham, C.A. Morton, and T. Urnes. ClockWorks: Visual programming of component-based software architectures. *Journal of Visual Languages & Computing*, 7(2):175–196, June 1996.

[56] T.C.N. Graham and W. Roberts. Toward quality-driven development of 3D computer games. In *Proceedings of the Thirteenth International Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '06), July 2006. To appear.

[57] T.C.N. Graham and T. Urnes. Relational views as a model for automatic distributed implementation of multi-user applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4), pages 59–66. ACM Press, 1992.

[58] T.C.N. Graham and T. Urnes. Linguistic support for the evolutionary design of software architectures. In *Proceedings of the 18th International Conference on Software Engineering* (ICSE 18, Berlin, Germany, Mar. 25–29), pages 418–427. IEEE Computer Society Press, 1996.

[59] T.C.N. Graham and T. Urnes. Integrating Support for Temporal Media into an Architecture for Graphical User Interfaces. In *Proceedings of the 19th International Conference on Software Engineering* (ICSE '97, Boston, MA, USA, May 19–23). IEEE Computer Society Press, 1997.

[60] T.C.N. Graham, T. Urnes, and R. Nejabi. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST '96, Seattle, WA, USA, Nov. 6–8), pages 1–10. ACM Press, 1996.

[61] S. Greenberg. Toolkits and interface creativity. *Journal of Multimedia Tools and Applications*, in press. Invited submission to the Special Issue on Groupware.

[62] S. Greenberg and D. Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '94, Chapel Hill, NC, USA, Oct. 22–26), pages 207–217. ACM Press, 1994.

[63] J. Greer and Z.B. Simpson. Minimizing latency in real-time strategy games. In D. Treglia, editor, *Game Programming Gems 3*, pages 488–495. Charles River Media, 2002.

[64] Groove. `groove.net`.

[65] J.C. Grundy. Engineering component-based, user-configurable collaborative editing systems. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18), September 1998.

[66] 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 Transactions on Computer-Human Interaction*, 1(2):81–125, June 1994.

[67] C. I. Johnson. Interactive graphics in data processing: Principles of interactive systems. *IBM Systems Journal*, 7(3/4):147–173, 1968.

[68] S. Junuzovic, G. Chung, and P. Dewan. Formally analyzing two-user centralized and replicated architectures. In *Proceedings of the European Conference on Computer Supported Cooperative Work* (ECSCW '05, Paris, Sept. 18–22), pages 83–102, 2005.

[69] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proc. 13th International Conference on Distributed Computing Systems (ICDCS)*, pages 195–202, 1993.

[70] A. Karsenty, C. Tronche, and M. Beaudouin-Lafon. GroupDesign: Shared editing in a heterogeneous environment. *Usenix Journal of Computing Systems*, 6(2):167–195, 1993.

[71] T. Kindberg, G. Coulouris, J. Dollimore, and J. Heikkinen. Sharing objects over the Internet: The Mushroom approach. In *Proceedings of IEEE Global Internet '96* (Mini-conference at GLOBECOM '96, London, England, Nov. 20–21). IEEE ComSoc, 1996.

[72] M. Kobayashi, M. Shinozaki, T. Sakairi, M. Touma, and S. Daijavad. Collaborative customer services using synchronous Web browser sharing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '98, Seattle, WA, USA), pages 99–108, 1998.

[73] M. Kolon and W.J. Goralski. *IP Telephony*. McGraw Hill, September 1999.

[74] G.E. Krasner and S.T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[75] D. Krech et al. The RDFLib Python library for RDF processing. `rdflib.net`.

[76] P. Kruchten. Architectural blueprints—The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[77] Y. Laurillau and L. Nigay. Clover architecture for groupware. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '02, New Orleans, LA, USA), pages 236–245. ACM Press, 2002.

[78] Jay Lee. Considerations for movement and physics in MMP games. In Thor Alexander, editor, *Massively Multiplayer Game Development*, pages 275–289. Charles River Media, 2003.

[79] D. Li and R. Muntz. COCA: Collaborative objects coordination architecture. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '98, Seattle, WA, USA), pages 179–188, 1998.

[80] J.C.R. Licklider. The computer as a communication device. *Science and Technology*, April 1968. Reprinted in Digital Systems Research Center Tech Note 61, August 7, 1990, available from `www.hpl.hp.com`.

[81] R. Litiu and A. Prakash. Developing adaptive groupware applications using a mobile component framework. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '00, Philadelphia, PA, USA), pages 107–116. ACM Press, 2000.

[82] R. Litiu and A. Prakash. DACIA: A mobile component framework for building adaptive distributed applications. *SIGOPS Operating System Review*, 35(2):31–42, 2001.

[83] S. Lukosch. Customizable data distribution for synchronous groupware. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS 2004)*, pages 70–77, 2004.

[84] J. Magee and J. Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[85] F. Manola and E. Miller. *Resource Description Framework (RDF) Primer*. World Wide Web Constorium (W3C), February 2004. `w3.org/TR/-rdf-primer/`.

[86] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 44–53, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[87] N. Medvidovic, R.N. Taylor, and Jr. E.J. Whitehead. Formal modeling of software architectures at multiple levels of abstraction. In *Proceedings of the California Software Symposium 1996* (CSS '96, Los Angeles, CA), pages 28–40, April 1996.

[88] M. Mintz. MSN Messenger protocol. `http://www.hypothetic.org/docs/msn`.

[89] A. Mitchell, I.R. Posner, and R.M. Baecker. Learning to write together using groupware. In *Human Factors in Computing Systems: CHI '95 Conference Proceedings* (Denver, CO, USA, May 7–11), pages 288–295, 1995.

[90] D. Mitchell. A component approach to embedding awareness and conversation. In *WETICE '98: Proceedings of the 7th Workshop on Enabling Technologies*, pages 82–89, Washington, DC, USA, 1998. IEEE Computer Society.

[91] NATO Mutilateral Interoperability Programme. *Multilateral Interoperability Programme Technical Interface Design Plan*, version 2.5, December 2005. Available from `mip-site.org`.

[92] R.E. Newman-Wolfe, M.L. Webb, and M. Montes. Implicit locking in the Ensemble concurrent object-oriented graphics editor. In J. Turner and R. Kraut, editors, *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4), pages 265–272. ACM Press, 1992.

[93] D.A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the ACM*

Symposium on User Interface Software and Technology (UIST '95, Pittsburgh, PA, USA, Nov. 14–17), pages 111–120. ACM Press, 1995.

[94] L. Nigay and J. Coutaz. Building user interfaces: Organizing software agents. In *Proc. ESPRIT'91 Conference*, pages 707–719, 1991.

[95] T. O'Grady. Flexible data sharing in a groupware toolkit. Master's thesis, University of Calgary, Calgary, Alberta, Canada, November 1996.

[96] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering* (ICSE '98, Kyoto, Japan), pages 177–186. IEEE Computer Society, 1998.

[97] J.F. Patterson. A taxonomy of architectures for synchronous groupware applications. *ACM SIGOIS Bulletin Special Issue: Papers of the CSCW'94 Workshops*, 15(3):27–29, April 1995.

[98] J.F. Patterson, M. Day, and J. Kucan. Notification servers for synchronous groupware. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '96, Boston, MA, USA, Nov. 16–20), pages 122–129. ACM Press, 1996.

[99] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[100] G.E. Pfaff et al. *User Interface Management Systems*. Eurographics Seminars. Springer Verlag, 1985.

[101] W.G. Phillips. Architectures for synchronous groupware. Technical Report 1999-425, Queen's University, Kingston, Ontario, Canada, May 1999. Available from `www.cs.queensu.ca` and `symbiosis.rmc.ca`.

[102] W.G. Phillips and T.C.N. Graham. Workspaces: A multi-level architectural style for synchronous groupware. In *Proceedings of the Tenth International Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '03), number 2844 in LNCS, pages 92–106. Springer-Verlag, 2003. ISBN 3-540-20159-9.

[103] W.G. Phillips and T.C.N. Graham. Workspace model specification, version 1.0. Technical Report 2005-493, Queen's University, Kingston, Ontario, Canada, March 2005. Available from `www.cs.queensu.ca` and `symbiosis.rmc.ca`.

[104] W.G. Phillips, T.C.N. Graham, and C. Wolfe. A calculus for the refinement and evolution of multi-user mobile applications. In *Proceedings of the Twelfth International Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '05), LNCS, pages 137–148. Springer-Verlag, 2005.

[105] E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF*. World Wide Web Constorium, `w3.org`, candidate recommendation, 6 April 2006.

[106] J. Randall. Scaling multiplayer servers. In D. Treglia, editor, *Game Programming Gems 3*, pages 520–533. Charles River Media, 2002.

[107] N. Rappin and R. Dunn. *wxPython in Action*. Manning, 2006. ISBN 1-932394-62-1.

[108] J. Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002. ISBN 0-735-61422-9.

[109] M. Roseman. *GroupKit 5.0 Documentation*. University of Calgary GroupLab, June 1998. Available from `http://www.cpsc.ucalgary.ca`.

[110] M. Roseman and S. Greenberg. GroupKit: A groupware toolkit for building real-time conferencing applications. In J. Turner and R. Kraut, editors, *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '92, Toronto, Canada, Oct. 31–Nov. 4), pages 43–50. ACM Press, 1992.

[111] M. Roseman and S. Greenberg. Building flexible groupware through open protocols. In *Proceedings of the Conference on Organizational Computing Systems* (ACM COOCS '93, Milpitas, CA, USA, November). ACM Press, 1993.

[112] M. Roseman and S. Greenberg. Building real time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction*, 3(1):66–106, March 1996.

[113] M.B. Rosson and J.M. Carroll. *Usability Engineering: Scenario-based Development of Human Computer Interaction*. Morgan-Kauffmann, 2002. ISBN 1-55860-712-9.

[114] J. Roth and C. Unger:. DreamTeam - a platform for synchronous collaborative applications. In Th. Herrmann and K. Just-Hahn, editors, *Groupware und organisatorische Innovation (D-CSCW'98)*, pages 153–165. B.G. Teubner Stuttgart, Leipzig, 1998.

[115] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998. ISBN 0-201-30998-X.

[116] C. Schuckmann. Private electronic mail message, June 15, 1998.

[117] C. Schuckmann, L. Kirchner, J. Schummer, and J.M. Haake. Designing object-oriented synchronous groupware with COAST. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '96, Boston, MA, USA, Nov. 16–20). ACM Press, 1996.

[118] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996. ISBN 0-13-182957-2.

[119] ShowEQ open-source project. `/www.showeq.net`.

[120] J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets*. Bruce Perens' Open Source. Prentice Hall, 2005. ISBN 0-131-47381-6.

[121] J.M. Spivey. *The Z Notation: A Reference Manual*. Internation Series in Computer Science. Prentice Hall, second edition, 1992. Also available from `spivey.oriel.ox.ac.uk`.

[122] M. Stefik, D.G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Transactions on Office Information Systems* (also in [6]), 5(2):147–167, 1987.

[123] N. Stephenson. *Snow Crash*. Bantam Spectra, 1993. ISBN 0-553-56261-4.

[124] N.A. Streitz, J. Geißler, J.M. Haake, and J. Hol. DOLPHIN: Integrated meeting support across liveboards, local and remote desktop environments. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '94, Chapel Hill, NC, USA, Oct. 22–26), pages 345–358. ACM Press/Addison-Wesley, 1994.

[125] SubEthaEdit. Available from `www.codingmonkeys.de`.

[126] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievments. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW '98, Seattle, WA, USA), pages 59–68. ACM Press, 1998.

[127] F. Tarpin-Bernard, B. David, and P. Primet. Frameworks and patterns for synchronous groupware: AMF-C approach. In *Proceedings of the IFIP 2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI '98,* Herkalion, Crete, September 14–18), September 1998.

[128] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[129] L.J. Terpstra. A security model for the Workspace groupware architecture. Master's thesis, Royal Military College of Canada, Kingston, Ontario, May 2002.

[130] UIMS Tool Developers' Workshop. A metamodel for the runtime architecture of an interactive system. *ACM SIGCHI Bulletin*, 24(1):32–37, 1992.

[131] T. Urnes. *Efficiently Implementing Synchronous Groupware*. PhD thesis, York University, Toronto, Ontario, Canada, 1998.

[132] T. Urnes and T.C.N. Graham. Flexibly mapping synchronous groupware architectures to distributed implementations. In *Proceedings of the Sixth Eurographics Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '99), pages 133–148, 1999.

[133] R. van Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

[134] G. van Rossum et al. The Python programming language. `python.org`.

[135] G. van Rossum and B. Warsaw. *Python Enhancement Proposal 8: Style Guide for Python Code*, version 43264, 23 March 2006. Available from `python.org`.

[136] B.T. Vander Zanden, R. Halterman, B.A. Myers, R. McDaniel, R. Miller, P. Szekely, D.A. Giuse, and D. Kosbie. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *ACM Transactions on Programming Languages and Systems*, 23(6):776–796, 2001.

[137] F.B. Viegas and J.S. Donath. Chat circles. In *Human Factors in Computing Systems: CHI '99 Conference Proceedings* (Pittsburgh, PA, USA, May 15–20), pages 9–16, 1999.

[138] D. Waitzman. A standard for the transmission of IP datagrams on avian carriers. Internet Engineering Task Force Request for Comments 1149, April 1990. Available from `ietf.org`.

[139] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories, Inc., November 1994. Available from `research.sun.com`.

[140] K. Walrath and M. Campione. *The JFC Swing Components: A Tutorial Guide to Constructing GUIs*. The Java Series. Addison-Wesley, 1999. ISBN 0-201-32577-2.

[141] Webarrow. `webarrow.com`.

[142] Webex. `webex.com`.

[143] M. Wermelinger and J.L. Fiadeiro. Algebraic software architecture reconfiguration. In *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 393–409, London, UK, 1999. Springer-Verlag.

[144] World of Warcraft. `worldofwarcraft.com`.

[145] J. Wu, T.C.N. Graham, and P. Smith. A study of collaboration in software design. In *2003 International Symposium on Empirical Software Engineering* (ISESE 2003) Rome, Italy. IEEE Computer Society, 2003.

[146] Xbox Live. `www.xbox.com`.

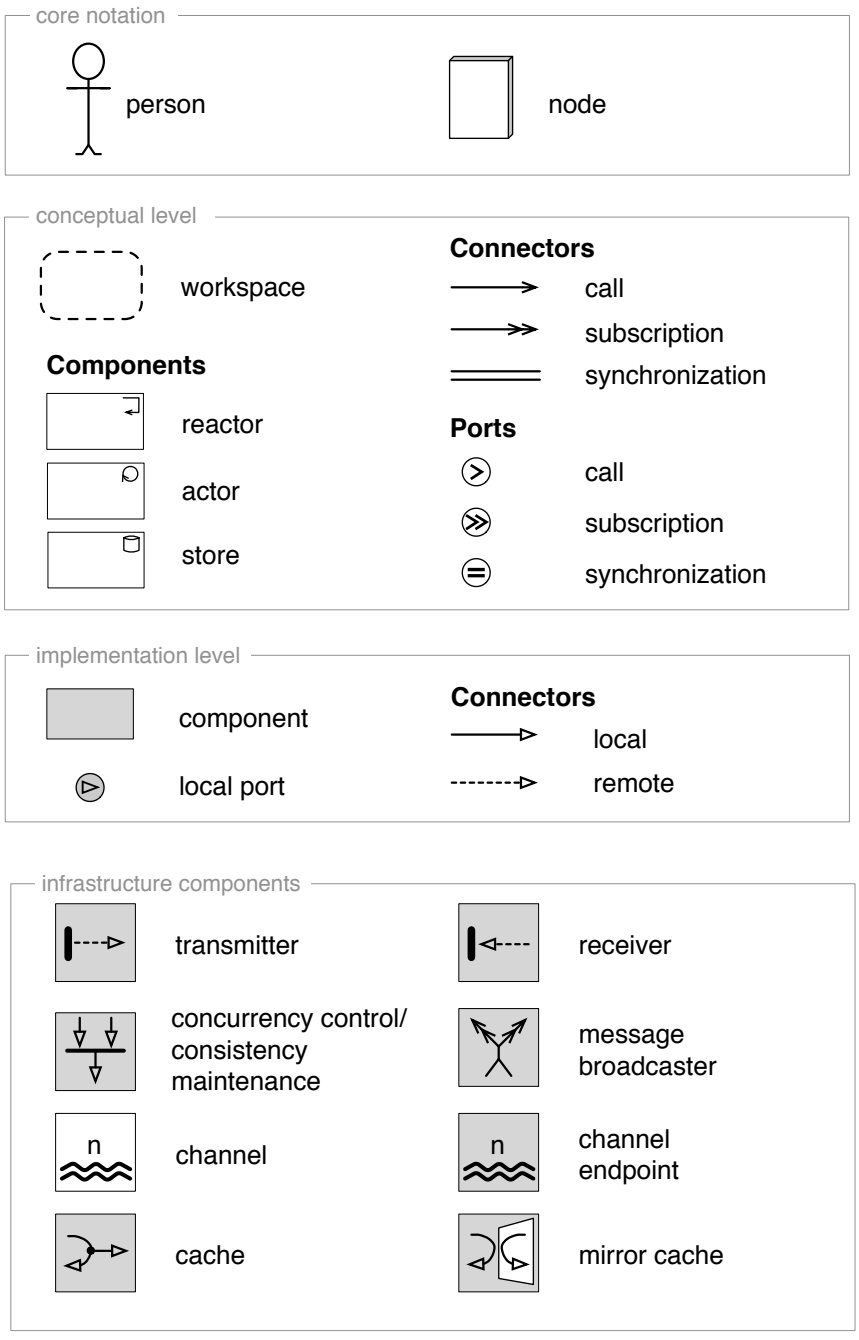
[147] Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 162–171, New York, NY, USA, 2004. ACM Press.

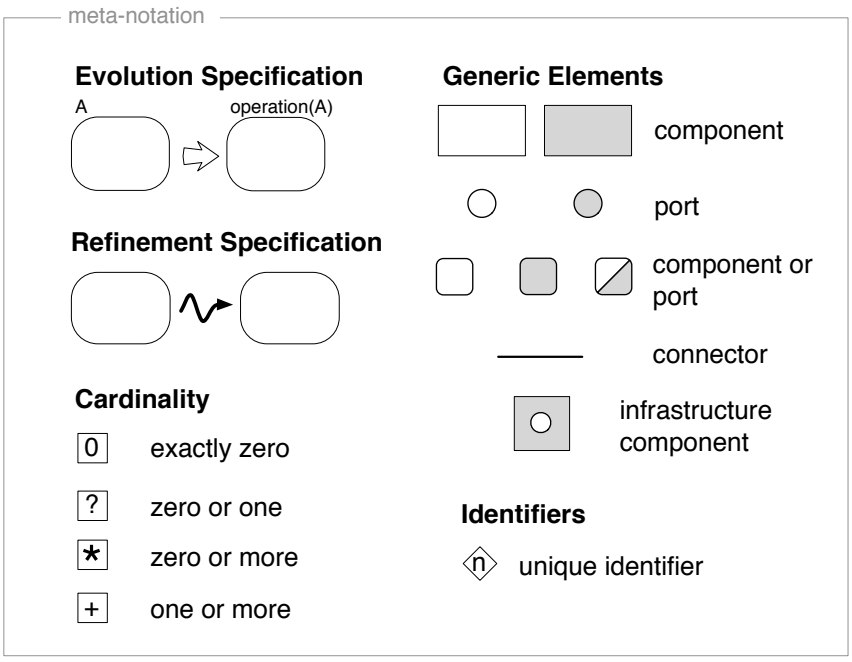
[148] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[149] D.A. Young. *X Window System: Programming and Applications with Xt, OSF/Motif Edition*. Prentice Hall, 1990. ISBN 0-13-49704-8.

Appendix A

Notation Summary for the Workspace Model




meta-notation
Evolution Specification
A
operation(A)
Refinement Specification
Cardinality
0
exactly zero
?
zero or one
*
zero or more
+
one or more
Generic Elements
component
port
component or port
connector
infrastructure component
Identifiers
n
unique identifier

Appendix B

Definitions

Side-effect free. A call or message a on a component C is side-effect free *iff* for any parameters p^*, request d, parameters q^* and deterministic component D, C.a(p); y=D.d(q) and y=D.d(q) result in y taking on the same value (assuming the system is otherwise quiescent). The intuition is that C.a(p) only returns values; it does not change any state in the system either directly or indirectly.

Side-effect free calls are referred to as *requests*. Side-effect bearing calls with no return value are referred to as *updates*. Other calls are referred to as *request-updates*. Side-effect free calls are by definition also referentially transparent.

Passive/active. A component C is passive *iff* whenever the system is otherwise quiescent, C does not initiate communication; *i.e.,* C does not make any calls (section 4.2.3.2) and C does not send any messages. The intuition is that C only reacts to communication performed by others; C does not initiate communication; C does not have its own thread of control. Components that are not passive are called active.

Deterministic/non-deterministic. A component C is deterministic *iff* whenever C is in state *s*, and some call or message $a(p^*)$ is applied to C, there is some unique r returned by a and the component enters some unique state s'. The intuition is that the component does not consult any external sources of information, such as random number generators, files whose values are non-deterministically determined, the system clock, *etc.* Components that are not deterministic are called non-deterministic. Active components are always non-deterministic.

Consistent. Two components are consistent at some time t if they meet some application-specific definition of consistency. The strongest form of consistency (called

strong consistency) is observational equivalence, meaning that any method called on either component would give the same result. (Consistency is therefore a notion that can be sensibly applied to deterministic components only.) Weaker notions of consistency can be applied; *e.g.*, weakly FIFO queues may be considered to be consistent even if they do not contain the same contents. Some definitions of consistency may therefore require knowledge of the past.

Two message streams are consistent if they meet some application-specific definition of consistency. Strong consistency over message streams means that the message stream traces contain identical messages in the same order. Weaker forms of consistency may involve (for example) one message stream collapsing messages into a shorter, semantically equivalent sequence, or message streams differing in the order in which messages are announced.