Workspace Model Specification

Version 1.0

W. Greg Phillips and T.C. Nicholas Graham
greg.phillips@rmc.ca graham@cs.queensu.ca

Technical Report 2005-493

School of Computing, Queen's University
Kingston, Ontario K7L 3N6

Copyright © 2005, W.G. Phillips and T.C.N. Graham

Document prepared March 31, 2005

Abstract

This specification defines the Workspace Model, a multi-level architectural style that supports describing, reasoning about and implementing synchronous groupware systems.

The top level of the model is the *conceptual level* which makes visible such features as users' workspaces, available computational devices, active and passive components (both real and virtual), calls, messages, and information shared between users. It is defined in terms of *workspaces*, *people*, *computational nodes*, *components*, *ports* and *connectors*. An *evolution calculus* defines the available operations for manipulating these workspace entities.

A conceptual level architecture and its changes over time can be used to describe a particular collaborative scenario. When instantiated in a software system, the conceptual level architecture represents a distribution-independent design description which can be mapped to any one of a number of possible implementations.

The implementation level of the model makes visible such concepts as process boundaries, method invocations, asynchronous and request-reply messaging protocols, concurrency control, caching, and replica consistency maintenance. The implementation level has its own evolution calculus.

Allowable refinements from the conceptual level to the implementation level are specified using a graph grammar notation. In effect, a refinement of a conceptual diagram results in an implementation. The refinements have been designed such that the implementation of any given conceptual level architecture will be both complete and correct with respect to the conceptual level design.

This specification defines the syntax and semantics of the conceptual and implementation levels, evolution at both levels, and refinement between levels, within a unified semi-formal framework.

Contents

1 Introduction and Rationale 5
1.1 This is a Specification . . . 5
1.2 A Note About Terminology . . . 6
1.3 Organisation . . . 6

2 Core Model Elements 6
2.1 Person . . . 6
2.2 Computational Node . . . 6

3 Conceptual Level Model Elements 7
3.1 Workspace . . . 7
3.2 Components . . . 8
3.3 Connectors . . . 8
3.4 Ports . . . 11
3.5 Components Redux . . . 12
3.6 Attributes . . . 14
3.7 Relations Among Conceptual Level Elements . . . 14

4 Conceptual Level Evolution Calculus 14
4.1 Meta-Notation and Pattern Matches . . . 15
4.2 Workspaces . . . 17
4.3 Nodes . . . 17
4.4 Components . . . 17
4.5 Ports . . . 18
4.6 Components and Nodes . . . 18
4.7 Connectors . . . 19
4.8 Connectors and Ports . . . 21
4.9 People . . . 22
4.10 Attribute Modification . . . 23
4.11 Conceptual Level Reflection Operations . . . 23

5 Implementation Level Model Elements 25
5.1 Implementation Components . . . 25
5.2 Connectors . . . 26
5.3 Ports . . . 26
5.4 Infrastructure Components . . . 27
5.5 Relations Among Implementation Level Elements . . . 29

6 Implementation Level Evolution Calculus 30
6.1 Components . . . 30
6.2 Ports . . . 30
6.3 Connect and Disconnect . . . 32
6.4 Implementation Level Reflection Operations . . . 32

7 Refinements from the Conceptual to the Implementation Level **33**
7.1 Components . . . 33
7.2 Ports . . . 34
7.3 Calls . . . 35
7.4 Subscriptions . . . 37
7.5 Synchronization . . . 39
7.6 Channels . . . 39
7.7 Inter-level Reflection Operations . . . 41

References **41**

A Definitions **43**

B Implementation Considerations **44**
B.1 Component Implementations . . . 44
B.2 Unique Identifiers . . . 44
B.3 Failure Reports . . . 44
B.4 Custom Implementations of Infrastructure Components . . . 44
B.5 Attributes as Implementation Hints . . . 44
B.6 Convenience Operations . . . 45
B.7 Port Creation and Use . . . 45
B.8 Channels . . . 45
B.9 Implementing Refinement . . . 45
B.10 Concurrency Control Issues . . . 46

List of Figures

1 Core notation. 7
2 Conceptual level notation. 7
3 Semantically equivalent synchronization group depictions, canonical description in the centre. 11
4 Meta-notation used in evolution calculus diagrams. 16
5 Conceptual level operations on workspaces. 17
6 Conceptual level operations on nodes. 18
7 Conceptual level operations on components. 19
8 Port creation and destruction. 20
9 Anchoring and floating components. 20
10 Conceptual level operations on connectors. 21
11 Attaching connectors to ports. 22
12 Detaching connectors from ports. 23
13 Implementation level notation. 25
14 Implementation level infrastructure components. 27
15 Operations on implementation level components. 31
16 Operations on implementation level ports. 31
17 Implementation level port connection and disconnection. 32
18 The schema used for refinement rules. 33
19 Refinements of components. 34
20 Refinements of ports 35
21 Refinements for call connectors. 36
22 Refinements for subscriptions. 37
23 Further refinements for subscriptions. 38
24 Refinements for synchronizations. 39
25 Refinements for channels. 40

1 Introduction and Rationale

This specification introduces and defines the Workspace Model, a multi level architectural model for groupware which fills two roles not well served by other such models [7].

First, the Workspace Model provides a set of notations that allow us to coherently describe and reason about a wide range of existing and desired collaboration modes and the software that supports them. These notations allow the state of a collaboration to be represented at any point in time, and also support description of the system dynamics required for changing collaboration modes and configurations of devices, networks, applications, data, and collaboration participants [8].

Second, the Workspace Model decouples necessarily-user-visible system issues from issues related to distributed implementation. User-visible issues include input and output displays, collaborative groups, and the ability to modify data or receive messages from message sources. Implementation issues include process boundaries, concurrency control, replicated versus centralised implementations of shared data, and performance optimisations. A mechanism is provided within the Model to rigorously reason about correspondences between user-visible system descriptions and implementation descriptions.

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 specification presents the core, the conceptual level, the implementation level, the representable changes or *evolutions* at each level and the mapping between levels (called *refinement*).

1.1 This is a Specification

It is worth emphasising that this document is a specification of a particular architectural model. It defines the model elements, the allowed evolutions on architectures, the required relationships between architectural levels, and *nothing else*. The syntax and semantics presented are deliberately abstract and independent of any particular programming language, application programming interface or implementation. The specification is not intended to guarantee interoperability of different implementations of the Workspace Model.

A language, toolkit or runtime system supporting the Workspace Model would be said to conform to this specification if it provides a semantically correct implementation of each of the key elements defined herein, such that all defined constraints are met.

The operations in the evolution calculus are deliberately primitive in order to support rigorous reasoning. We anticipate that implementations of the model would provide higher-level "convenience operations". This, and other implementation issues, are discussed further in appendix B.

1.2 A Note About Terminology

The terms side-effect free, request, update, request-update, passive, active, deterministic, non-deterministic, and consistent are used in specific technical senses in this specification. For definitions, see appendix A on page 43.

1.3 Organisation

The specification is organised as follows. The next section presents the core model elements which are visible at both the conceptual and implementation levels. Section 3 introduces the key model elements of the conceptual level, their properties and their relations to one another. This is followed by a description of the conceptual level *evolution calculus*, which is used to express changes to the conceptual level architecture over time. Sections 5 and 6 similarly introduce the implementation level and its evolution calculus. Finally, section 7 defines how conceptual level architectures are refined to implementation level architectures which may ultimately be realized in hardware and software.

2 Core Model Elements

The notation for the Workspace Model's core elements is shown in figure 1. The two core elements are *person*, *computational 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.

2.1 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 3.5 for more details.

2.2 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.

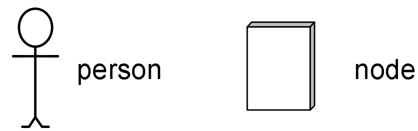

Figure 1: Core notation.

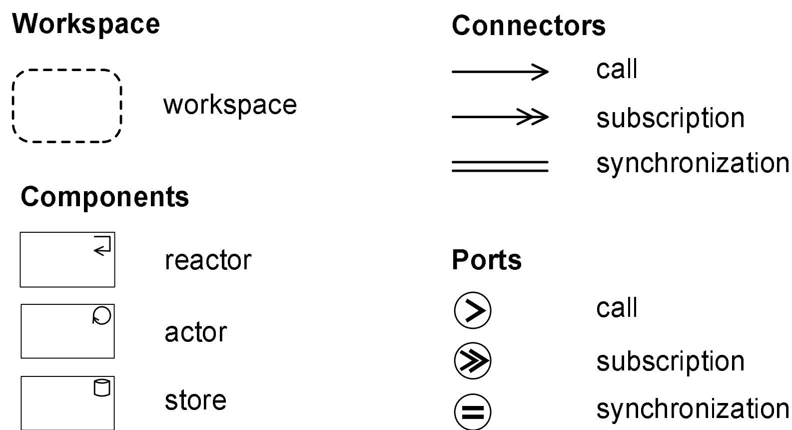

Figure 2: Conceptual level notation.

3 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 [8].

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 7.

The conceptual level notation is summarised in figure 2. In addition to the two core constructs already presented in figure 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.

3.1 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 3.6).

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

3.2 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 bridges 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 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 3.5. However, in the interim an approximate intuition is:

- A reactor is a software or hardware component which is passive (inert until acted upon, see appendix A). Once acted upon it may send messages to, or directly operate on, other components to which it is connected.
- An actor is like a reactor except that it is active; that is, it has the ability to independently initiate activity within the workspace.
- A store is like a reactor except that it may not directly operate on other components. In addition, stores may represent information that is shared between workspaces.

Components may have names and types, which 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 3.6).

3.3 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 3.3.4 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.

3.3.1 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 compo-

nent'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.

3.3.2 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 A).
- 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.
- 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.

3.3.3 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.
- All sources and targets of a given subscription connector must be within the same workspace.

[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.

- 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.

3.3.4 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]

A group of mutually synchronized stores is referred to as a *synchronization group*. Within a synchronization group components converge to consistency (see Annex A) 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 $t_2 \geq t_1$, 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 3 illustrates semantically equivalent five 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 g, it is s that is modified to bring about consistency.

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

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

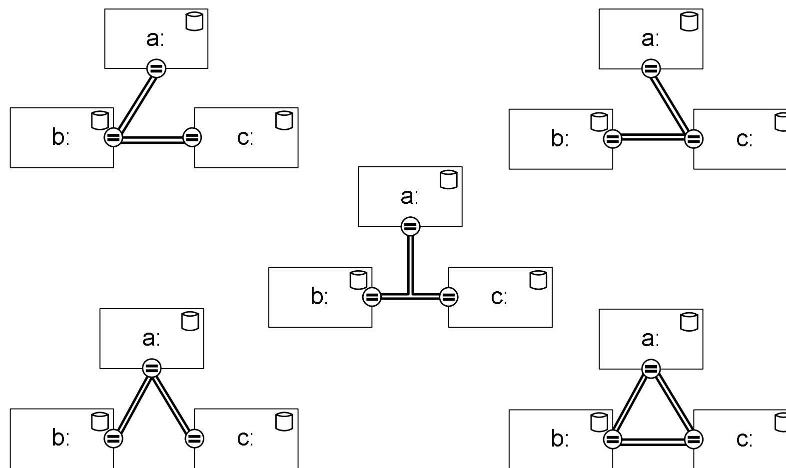

Figure 3: Semantically equivalent synchronization group depictions, canonical description in the centre.

3.4 Ports

Components may provide any number of *ports*, which represent attachment points for connectors. Ports are created dynamically on component surfaces. If "static" 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 3.5 for details. Further, it is possible to create a port on a component that serves no useful purpose; it is up to the component designer and run-time user to ensure that port creations are sensible.

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.

3.4.1 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, run-time errors (reported as exceptions or by some other means) may result.

3.4.2 Subscription Ports

A subscription source port may be attached to at most one outgoing connector. Since subscription connectors are many-to-many, we expect that a single outgoing subscription connector will suffice for many purposes. However, 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.

3.4.3 Synchronization Ports

A synchronization port may be attached to at most one subscription connector. The vocabulary of a subscription 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.

3.5 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.

3.5.1 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.

3.5.2 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 A (*e.g.*, clocks) are often modelled as actors. Software components with internal threads of control (*e.g.*, servers) are likewise modelled as actors.

3.5.3 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 which 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 [6] or the Abstraction component in the PAC architecture [3]. A telephone call might be partially modelled as two synchronized stores, one in the workspace of each call participant.

3.5.4 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, able to create messages (*e.g.*, by physical movement) and 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.

3.6 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 a has a corresponding observed attribute a'. 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 runtime system. For example, a synchronization connector might be given an attribute suggesting that a centralised implementation would be most appropriate. See section B.5 for further discussion.

3.7 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 one exception to this rule is synchronization connectors, which are not subject to the containment relation.

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 frequently omit ports in workspace diagrams. If a connector k is shown touching the boundary of a component c, then k is attached to some (unseen) port p provided by c.

Anchorage. 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.

4 Conceptual Level Evolution Calculus

The complete configuration of conceptual level workspace elements at any point in time is referred to as a *run time 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 6.

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 1 and 2, as well as a meta-notation which is introduced in figure 4 and described in section 4.1.

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). The left-hand side of the diagram represents a pattern that must be matched in A for the operation to succeed. Essentially, it 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 3.7. 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 6 (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 8). 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, $\omega(A, p^*) = A$.

4.1 Meta-Notation and Pattern Matches

The meta-notation used in evolution calculus definitions is shown in figure 4. The meta-notation includes a template for evolution calculus specifications, a set of generic workspace element symbols, a means of identifying particular workspace elements, symbols for cardinality constraints, and symbols representing component state.

As discussed in the previous section, an operation $\omega(A, p^*)$ is specified by one or more diagrams, which take the form of the evolution specification of figure 4. The left hand side of the diagram (labelled A) represents a precondition which must be matched in A for the operation to succeed.

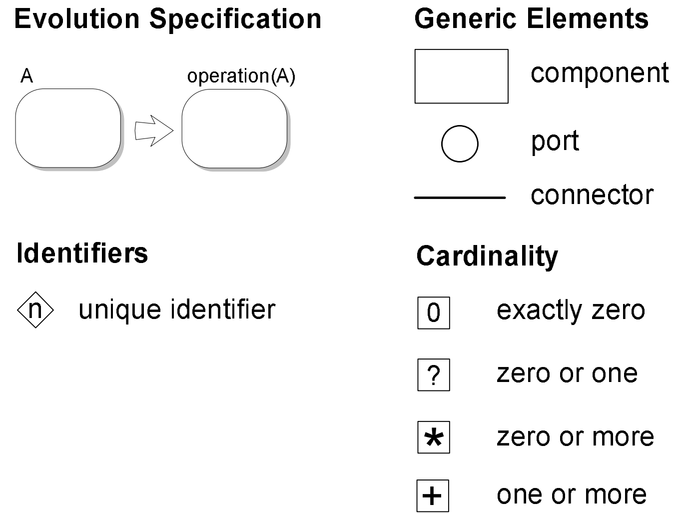

Figure 4: Meta-notation used in evolution calculus diagrams.

For a pattern to match, all elements shown in the left-hand side of the diagram must be present in A, with the identifiers and cardinalities given, and taking part in any relations depicted in the diagram (see section 3.7).

Each workspace element in an architecture has a unique identifier. 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 operations signature and the identifier must not be in use in A. This ensures that an identifier identifies at most one workspace element.

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 5(a) will match if A includes a workspace w which contains exactly zero components, zero nodes, zero call connectors and zero subscription connectors.

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

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.

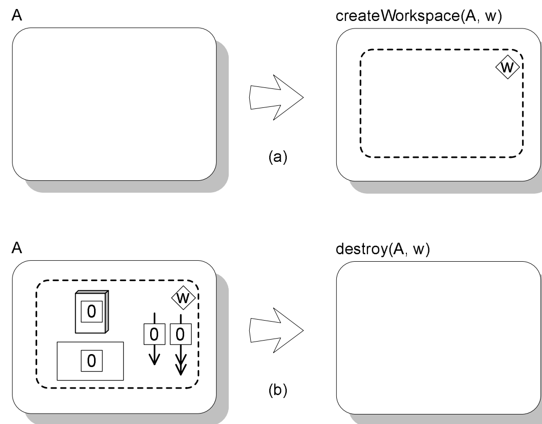

Figure 5: Conceptual level operations on workspaces.

4.2 Workspaces

Figure 5 (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 implied 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 5 (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.)

4.3 Nodes

Figure 6 (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 4.6.
Figure 6 (b). A node with no anchored components may be destroyed.
Figure 6 (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.

4.4 Components

Figure 7 (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.

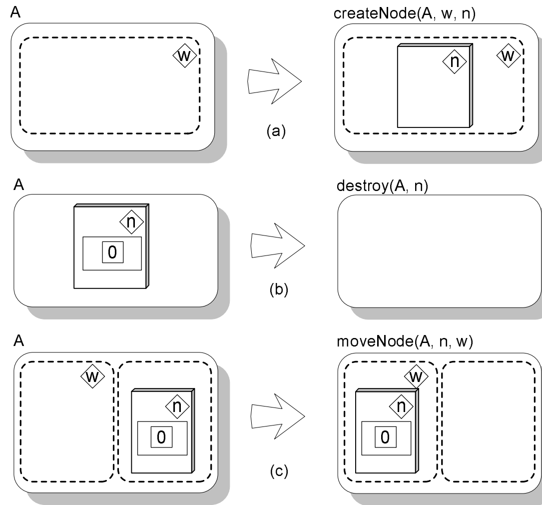

Figure 6: Conceptual level operations on nodes.

Figure 7 (d). A component may be destroyed, provided none of its ports are attached to any connectors (see section 4.7). Destroying a component also destroys all ports present on its interface (see section 4.5).
Figure 7 (e). A component may be moved from one workspace to another, provided none of its ports are attached to call or subscription connectors. Such a move leaves synchronization connectors attached and has no effect on ports. For clarity, ports are not shown in this figure.

4.5 Ports

Ports may be created on component boundaries. The kind of the component constrains the kinds of ports that may be created on it. Ports are always unattached when created. Attachment is defined in section 4.8.
Figure 8 (a) and (b). Call source ports may be created only on actors and reactors.
Figure 8 (c), (d) and (e). Call targets ports and subscription source and target ports may be created on any component.
Figure 8 (f). Synchronization source ports may be created only on stores. A store may provide a maximum of one synchronization port.
Figure 8 (g). A port may be destroyed only if it is not attached to any connector.

4.6 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

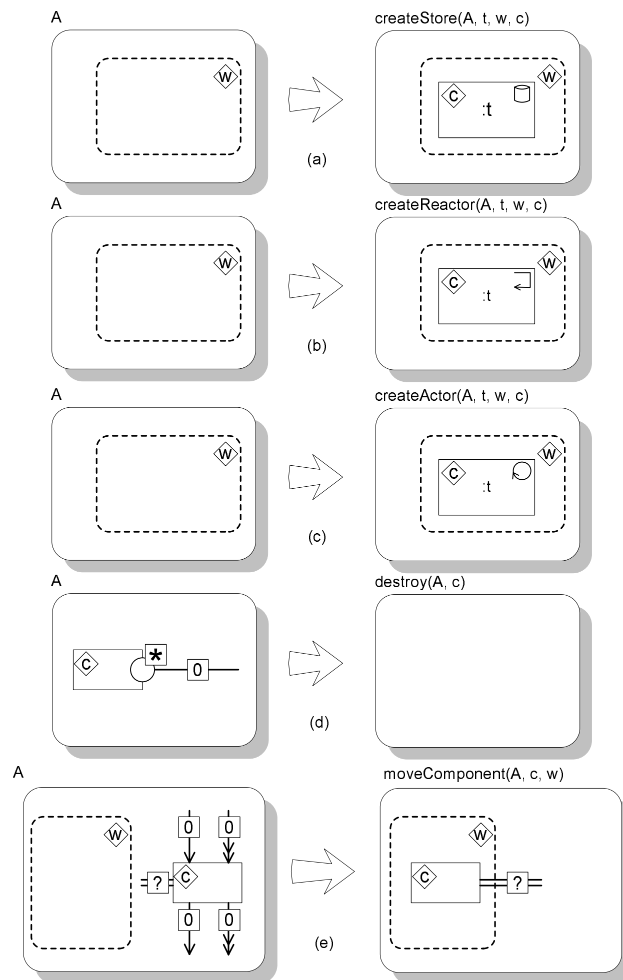

Figure 7: Conceptual level operations on components.

certain information is present on a laptop that is about to be disconnected from a network.
Figure 9 (a). A components 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 9 (b). A component that is anchored to a node may be floated off it. Floating a component leaves it in the same workspace. Floating a component has no effect on ports or connectors.

4.7 Connectors

Figure 10 (a) and (b). Call and subscription connectors may be created within workspaces. Newly created connectors are not initially attached to ports.
Figure 10 (c). Synchronization connectors are not subject to workspace containment

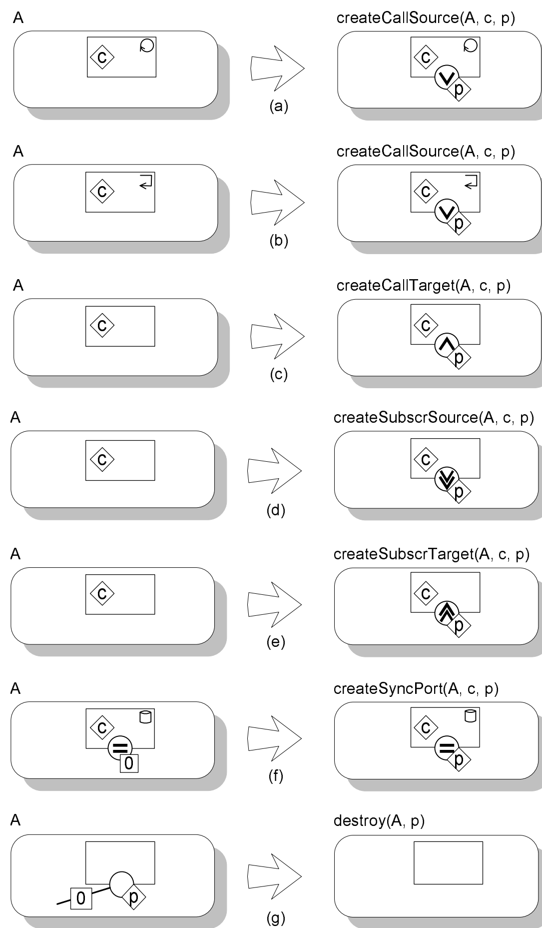

Figure 8: Port creation and destruction.

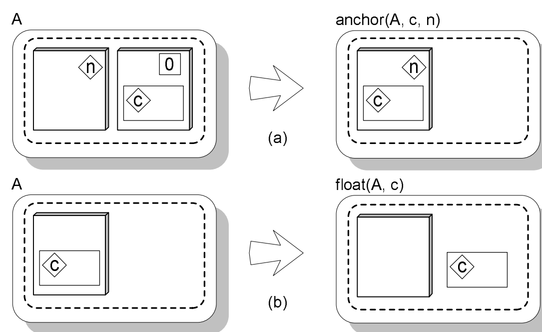

Figure 9: Anchoring and floating components.

and may simply be created.

Figure 10 (d), (e) and (f). Connectors which are not attached to ports may be destroyed. See section 4.8 for the definitions of attachment and detachment.

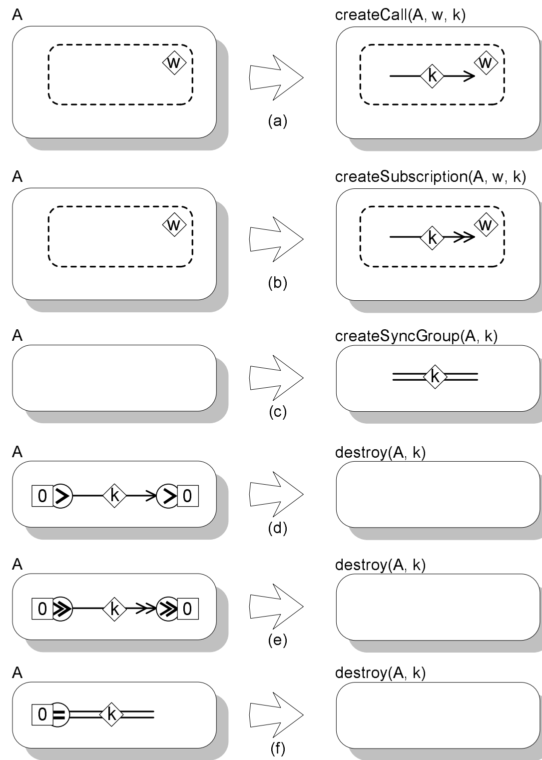

Figure 10: Conceptual level operations on connectors.

4.8 Connectors and Ports

Connectors may be attached to, and detached from, corresponding ports. Call and subscription connectors may be attached only to ports in the same workspace; 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 11 (a). A call connector with an unattached source end may be attached to a call source port that is also unattached.

Figure 11 (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 11 (c). A subscription connector may be attached to a subscription source port that is not attached to any other subscription connector. This allows a subscription connector to be attached to multiple source ports.

Figure 11 (d). A subscription connector may be attached to a subscription target port, regardless of any other connectors attached to that port. This allows a subscription connector to be attached to multiple target ports, and a subscription target port to accept multiple incoming connectors.

Figure 11 (e). A synchronization connector that is not attached to any ports may

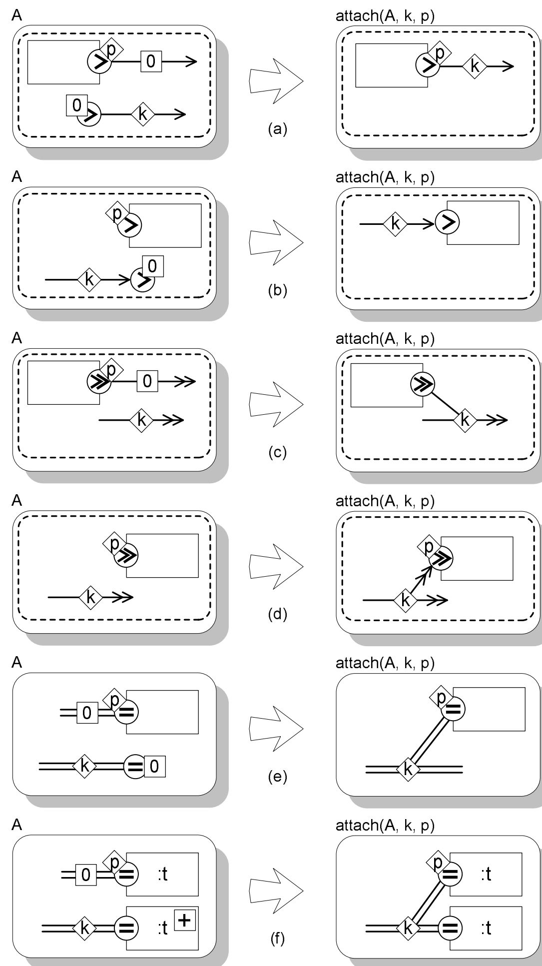

Figure 11: Attaching connectors to ports.

be attached to an unattached synchronization port.
Figure 11 (f). A synchronization connector that is already attached to one or more 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 12 (a) through (e). Any attached connector/port pair may be detached.

4.9 People

As discussed in section 3.5.4, 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.

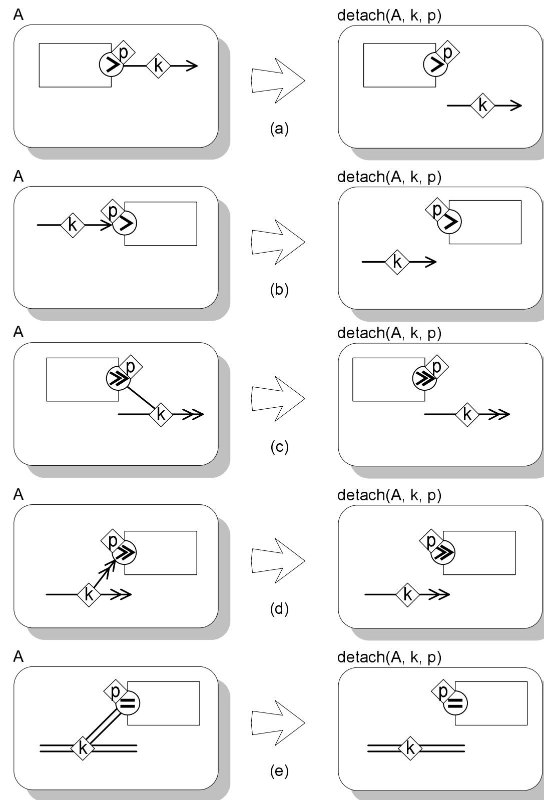

Figure 12: Detaching connectors from ports.

4.10 Attribute Modification

As discussed in section 3.6, any workspace element may have arbitrary attributes associated with it. The attributes are not normally shown in the architectural diagrams in this specification. The list of these attributes and values of the intent attributes may be modified at run time.

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:

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

4.11 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 opera-

tion's signature (*e.g.*, component or connector identifiers). It is therefore a practical 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.

Strictly speaking, the reflection operations do not form part of the evolution calculus, since they do not result in evolution. However, since reflection operations appear to be a practical necessity for the effective use of evolution operations, a minimal list of the required reflection operations is given here.

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.
  $getPorts : A \times c \rightarrow \mathbb{P}(p)$

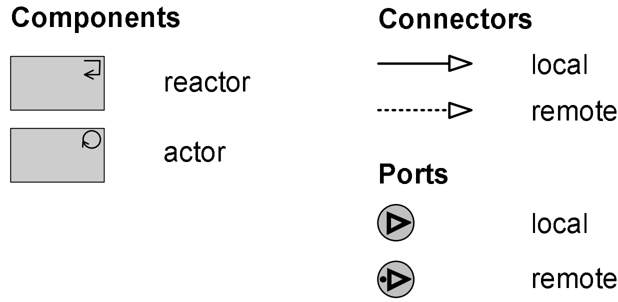

Figure 13: Implementation level notation.

- 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 6.4 and 7.7, respectively.

5 Implementation Level Model Elements

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.

In this section we introduce the components, connectors and ports that make up the implementation level. The evolution calculus operations for the implementation level are defined in section 6. In section 7 we present the refinements that map from the conceptual level to the implementation level.

The implementation level notation is summarised in figure 13. It includes two kinds of components, analogous to the reactor and actor components of the conceptual level. It also includes local connectors, corresponding to in-process procedure calls or method invocations, and remote connectors, corresponding to inter-process messages. As at the conceptual level, there are source and target ports supporting each kind of connector.

5.1 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. At the implementation level there are no stores. The two implementation level component types are:

Implementation Reactor. Implementation level reactors must be passive and deterministic. An implementation reactor may provide local and remote source and target ports. The implementation of a conceptual level reactor or store will include an implementation level reactor plus other components and connectors.

Implementation Actor. As at the conceptual level, implementation level actors may be active and non-deterministic. An implementation actor may provide local and remote source and target ports. The implementation of a conceptual level actor will include an implementation level actor plus other components and connectors.

5.2 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 request 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. The message sender does not block on a reply. Where a request-reply protocol is required (*e.g.*, for implementing remote calls), or where guaranteed message delivery is required, this must be implemented by the components at either end of the connector (see the transceiver component in section 5.4.1). A remote connector may be used from one node to itself.

A request or request update call will require two remote connectors, one to make the request and the other to return the result.

5.3 Ports

The implementation level provides two kinds of ports, corresponding to its two kinds of connectors. Each kind of port has source and target ends. 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.

A source port will normally correspond to a syntactic construct, visible within the containing component's code, that allows procedure calls or method invocations to be made on it. This will typically be a named variable or an element of a list or array.

A target port will normally correspond to a callable interface on the containing component.

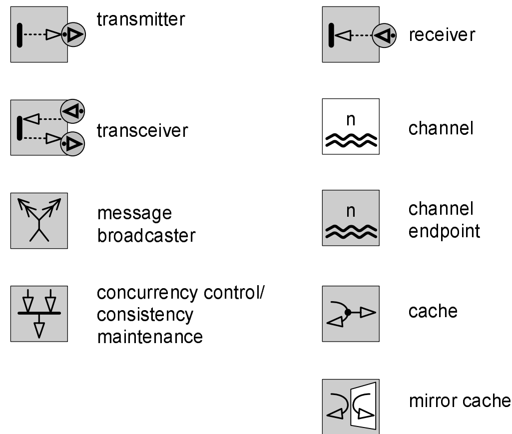

Figure 14: Implementation level infrastructure components.

5.4 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 14 and defined in the remainder of this section.

5.4.1 Transmitters, Receivers and Transceivers

Transmitter-receiver pairs and transceiver 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.

Transmitter-receiver pairs are used to implement one-way reliable messaging, as would be needed to implement pure update call connectors and subscription connectors. A transmitter provides one remote source port and a receiver provides one remote target port.

Transceiver pairs are used to implement reliable request-reply protocols, as would be needed for request and request-update call connectors. A transceiver provides a remote source port and a remote target port.

5.4.2 Channel

In implementing groupware it is frequently necessary for a group of endpoint components to communicate with one another. While it is possible to connect n components using $n(n-1)$ transceiver pairs, it is often simpler to consider these communication paths as channels [2]. Messages sent into a channel at any one of its endpoints are received at all other endpoints.

Figure 14 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 7 and are not intended to be used directly at the conceptual level. They are ultimately refined into channel endpoints.

Channels and channel endpoints are identified by a channel designator, shown as n in figure 14. In workspace diagrams, any two channel endpoints with the same name represent endpoints on the same channel.

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 preferred. Where this is the case, it may be necessary to specify a particular channel implementation. See appendix B.

Message broadcasters are used in the implementation of subscription connectors. They accept messages as inputs and broadcast them to one or more subscribers.

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. The implementation of message broadcasters may make use of a multicast messaging layer or of shared channels.

5.4.3 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 5.4.4).

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 operation transform [4], Oreste [5], *etc.*)

5.4.4 Cache and Mirror Cache

Where calls with return values (requests) are implemented using transceivers and synchronous messages, caches can be used 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. In addition to incoming and outgoing call ports, caches also provide an incoming subscription port to allow cache invalidation and update messages to be received and acted on.

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 provides an incoming local port which is used by the connected CCCM to provide cache invalidation messages when the state of the target changes. It also provides an outgoing (conceptual level) update port which is used pass cache invalidations on to its cache component, 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). Invalidations and updates are transmitted on the outgoing update port.

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. 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 7.3.

5.5 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. 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. 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.

6 Implementation Level Evolution Calculus

Just as for the conceptual level, implementation level operations are defined by an evolution calculus. The semantics of the implementation level calculus operations are specified in this section, using the meta-notation introduced in figure 4 on page 16.

6.1 Components

The allowed operations on implementation level components are defined in figure 15.
Figure 15 (a). A component may be instantiated on a node. Newly instantiated components are not connected to other components.
Figure 15 (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 5.2 for further discussion.
Figure 15 (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 15 (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.

6.2 Ports

Remote ports are found only on transmitters, receivers and transceivers (see section 5.4.1 and figure 14) and are considered integral to those components; hence, there are no operations to create them or destroy them.
Figure 16 (a) and (b). Local source and target ports may be created on component boundaries. Created ports are initially unconnected.
Figure 16 (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.

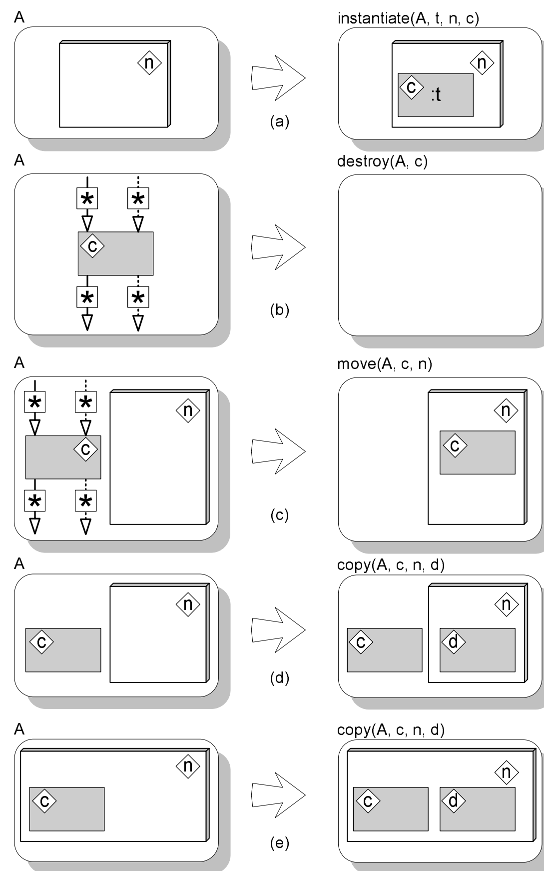

Figure 15: Operations on implementation level components.

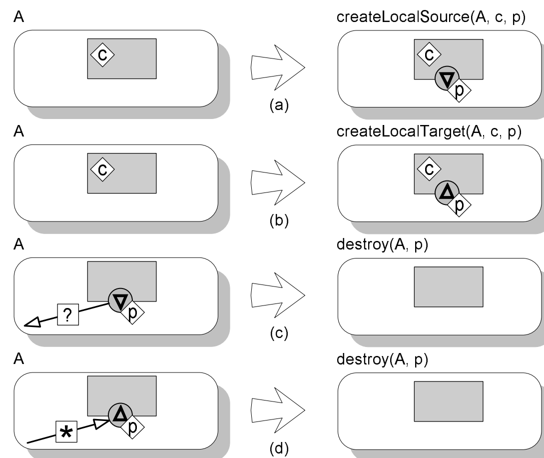

Figure 16: Operations on implementation level ports.

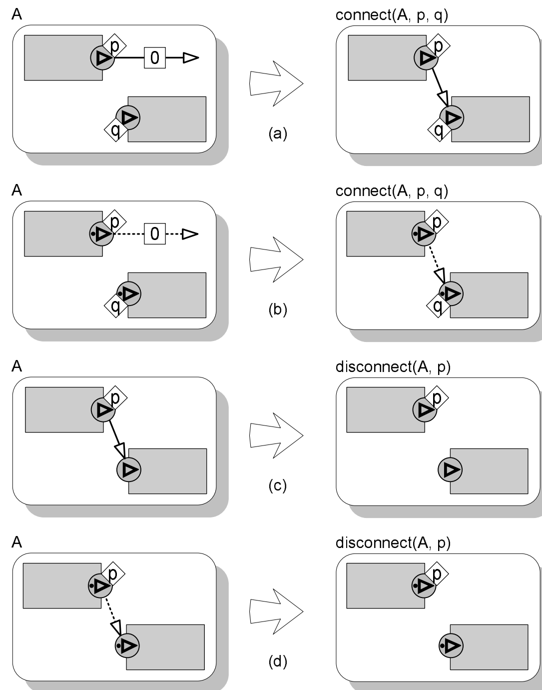

Figure 17: Implementation level port connection and disconnection.

6.3 Connect and Disconnect

Figure 17 (a) and (b). An unconnected local source port may be connected to a local target port. An unconnected remote source port may be connected to a remote target port. Connecting two ports implicitly creates a connector of the appropriate type.
Figure 17 (c) and (d). Two connected ports may be disconnected. This action implicitly destroys the connector. Note that since source ports may have only one outgoing connector we can specify the connector to remove by means of the source port's identity.

6.4 Implementation Level Reflection Operations

The implementation level requires a set of reflection operations similar to those for the conceptual level introduced in section 4.11 on page 23. As in that section, we here describe a minimum set.

As noted in section 3.7, 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 $getProvidingComponent$ defined in section 4.11 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:

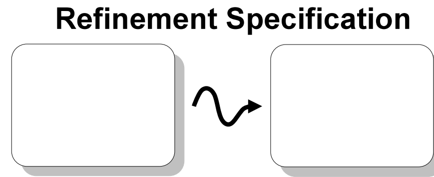

Figure 18: The schema used for refinement rules.

- 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 port, return all connected ports.
  $getConnectedPorts : A \times P \rightarrow \mathbb{P}(P)$

7 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 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 conceptual level elements remain in the architecture.

The refinement rules use the meta-notation and pattern match rules introduced in section 4.1 on page 15. Refinements rules are written in the refinement schema illustrated in figure 18. The squiggly arrow means “may be refined to” or “may be implemented as”.

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.

7.1 Components

The main refinements for components are illustrated in figure 19. Stores which are part of synchronization groups may also be implemented according to the refinements presented in section 7.5.

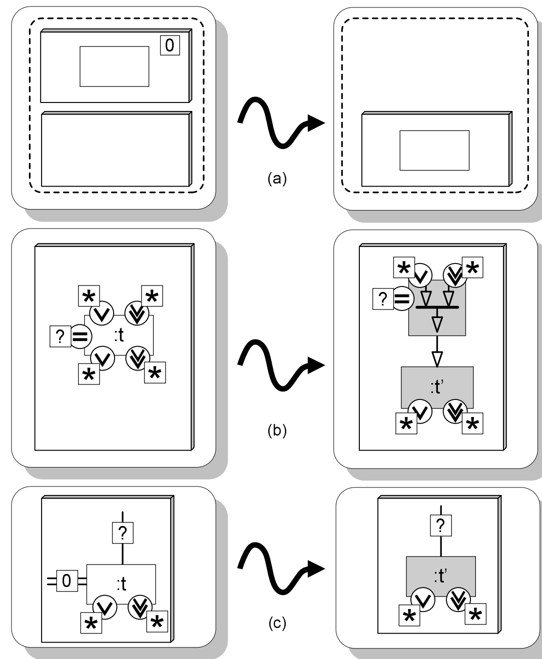

Figure 19: Refinements of components.

Figure 19(a). A conceptual level component which is not anchored to any node may be implemented on any node in its workspace.
Figure 19(b). A conceptual level component of type t which is anchored to a node may be implemented on that node. The implementation will include a component of corresponding type t' plus a concurrency-control consistency maintenance component. Any synchronization port and all incoming call and subscription ports appear on the CCCM component and any outgoing connectors are routed from the component of type t'. Ports are not yet refined to the implementation level.
Figure 19(c). This is a simplification of case (b) for a conceptual level component that is not a member of a synchronization group and that has not more than one incoming connector (call or subscription). In this case the CCCM component is not required, since replica consistency maintenance is not an issue and there will never be more than one thread entering the component.

7.2 Ports

The refinements for subscriptions and synchronizations ultimately convert ports attached to these connector types to call ports (see sections 7.4 and 7.5), so only refinements for call ports and unconnected ports are necessary.
Figure 20(a) and (b). Call ports refine to single local source ports, either source or target as appropriate.
Figure 20(c). Any port which is not attached to a connector need not be implemented.

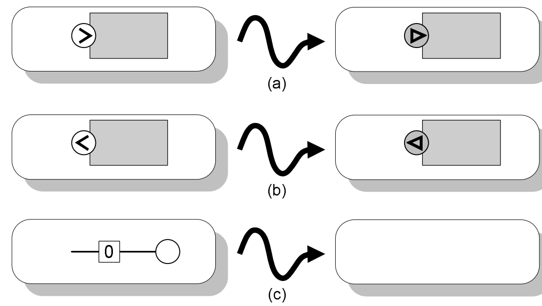

Figure 20: Refinements of ports

7.3 Calls

Figure 21(a). Calls between ports on the same node may be implemented directly by local connectors.[4]
Figure 21(b). Calls between any two ports, whether or not on the same node, may be implemented by inserting transceivers in the call path to mediate the inter-nodal communication.The fact that the connectors from the ports to the transceivers are local connectors implies that each transceiver must be on the same node as its associated port. There may be an unnecessary performance penalty incurred by this approach where the ports are in fact on the same node.
Figure 21(c). Pure updates may be implemented using a transmitter-receiver pair rather than transceivers, since no return value is required.
Figure 21(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 transceivers, 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.

[4]In this figure, and in several that follow, the components on which the illustrated ports would appear have been suppressed; the directionality of the ports may be inferred from the attached connectors. Similarly, the port depictions for the infrastructure components have been suppressed to reduce visual clutter, without loss of information.

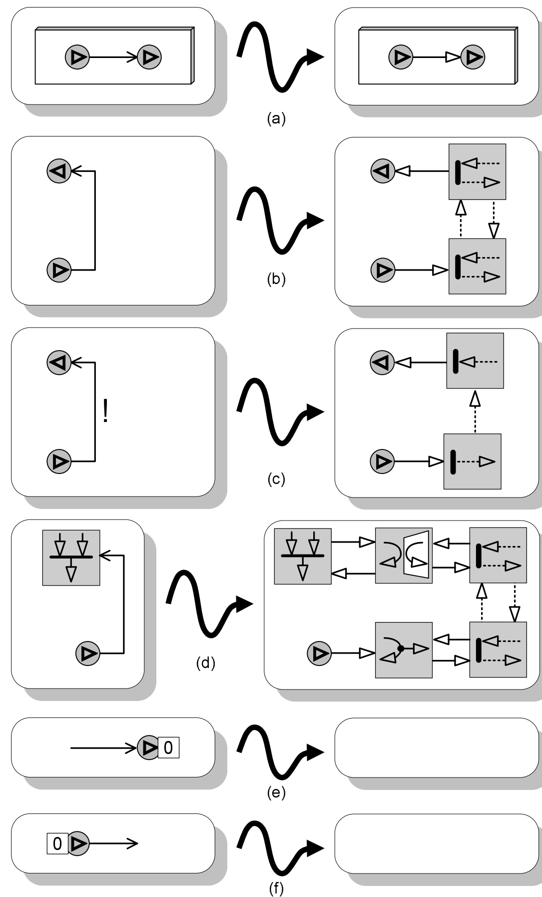

Figure 21: Refinements for call connectors.

- The CCCM calls the mirror cache via the 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.

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 safely be cached because of actors' inherent non-determinism.

Figure 21(e) and (f). Call connectors that are unattached at either end are not implemented.

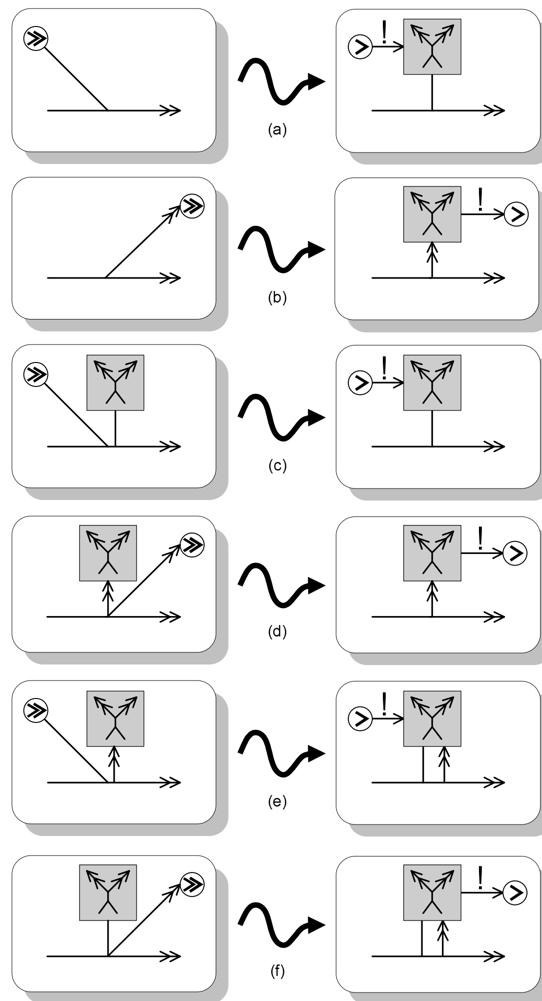

Figure 22: Refinements for subscriptions.

7.4 Subscriptions

7.4.1 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.

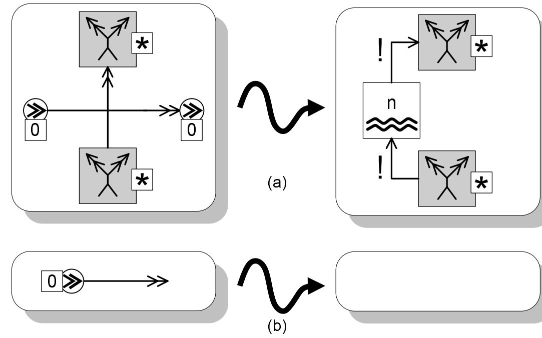

Figure 23: Further refinements for subscriptions.

7.4.2 Refinable Subscription Connections

Since subscriptions have an asynchronous semantics, at least one message broadcaster is required in any implementation of a synchronization connector with one or more attached source ports.

The refinements in figure 22 allow such connectors to be implemented using anywhere from a single message broadcaster to an independent message broadcaster for each connected port.

Figure 22(a). A subscription source port attached to a subscription connector may be refined to a call port attached to a message broadcaster via an update connector. The message broadcaster then becomes a source for the subscription connector. Any other ports attached to the subscription connector are unaffected.

Figure 22(b). This is the analogue of (a) for a target port.

Figure 22(c). A subscription source port attached to a subscription connector which already has a message broadcaster as a source may be refined to a call port attached to that message broadcaster by an update connector.

Figure 22(d). This is the analogue of (c) for a target port.

Figure 22(e). A subscription source port attached to a subscription connector which already has a message broadcaster as a target may be refined to a call port attached to that message broadcaster by an update connector. The message broadcaster must then be attached to the subscription connector as a source.

Figure 23(f). This is the analogue of (e) for a target port.

Figure 23(a). Where a set of message broadcasters is attached to a subscription connector and that connector has no other sources and targets, the connector may be implemented using a channel (with some channel designator n which is not currently used in A) connected to the message broadcasters by update connectors. The implementation of channels is defined in section 7.6.

Note that an individual message broadcaster may be both a source and target in this refinement.

Figure 22(b). A subscription connector with no sources need not be implemented.

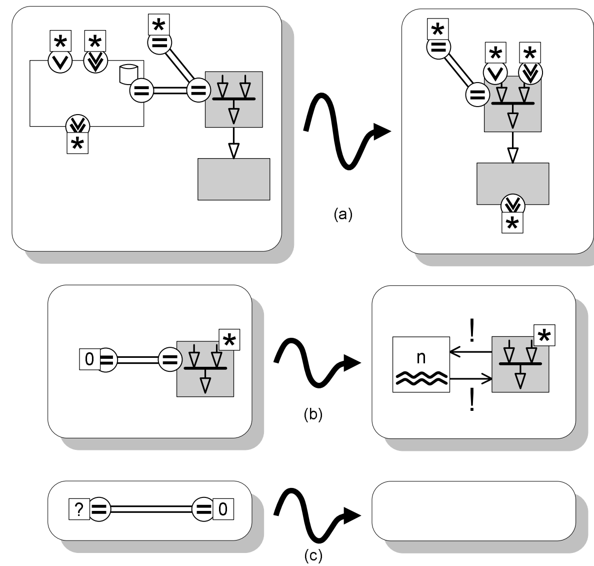

Figure 24: Refinements for synchronizations.

7.5 Synchronization

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

Figure 24(a). A store which is synchronized with an already-implemented component (see section 7.1) 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 outgoing call ports.)

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

Figure 24(b). A group of store implementations which are mutually synchronized (here represented by their CCCM components) may be implemented by attaching each of them to a new channel with some new channel designator n.

In effect, this is a replicated implementation, wherein the CCCM components are responsible for enacting replication protocols adequate to ensure store consistency.

Figure 24(c). A synchronization group with zero or one connected ports need not be implemented. If it had one connected port, that port need not be implemented either.

7.6 Channels

In figures 23 and 24, conceptual level channels were introduced to handle the n-way communication required between subscription ports and between CCCM components. This section presents the refinements that transform conceptual level chan-

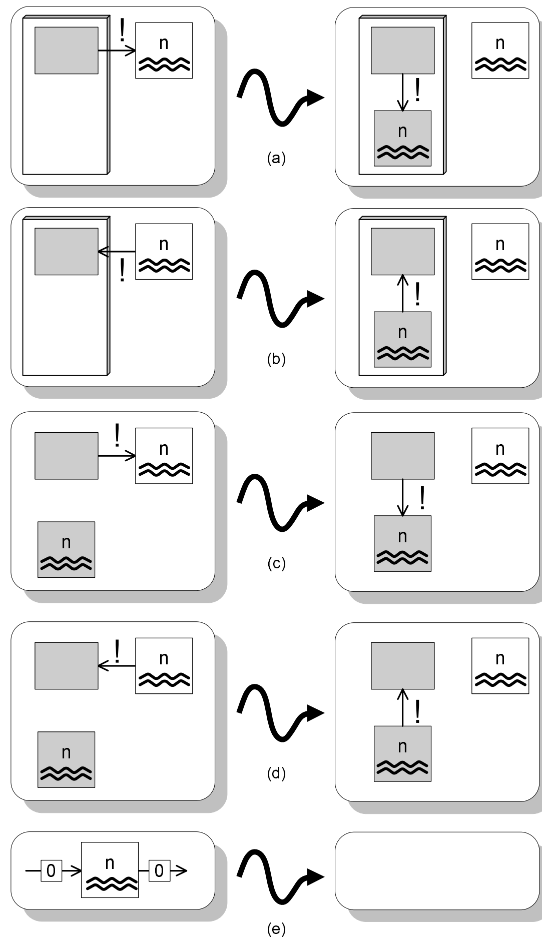

Figure 25: Refinements for channels.

nels to channel implementations.

Figure 25(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 on the same node as c. The conceptual level channel is otherwise unaffected.

Figure 25(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.

Figure 25(e). A conceptual level channel with no attached connectors is not implemented.

These refinements allow connected channels to be implemented using 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.

7.7 Inter-level Reflection Operations

Supporting dynamic evolution at run-time 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 for which it forms part of the implementation.
  $getConceptual : A \times E \rightarrow \mathbb{P}(e)$

Acknowledgements

Len Bass, Larry Constantine, Emannuel Dubois, Phil Gray, Rick Kazman, Laurence Nigay, Len Terpstra, Leon Watts and James Wu all provided valuable suggestions at various points during the Model's development. Later stages of the development were done in close collaboration with Chris Wolfe. Of course, any deficiencies in this specification are the fault of the authors alone.

References

[1] 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), June 2000. Also available from `www.spread.org`.

[2] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53,103, December 1993.

[3] J. Coutaz. PAC, an object oriented model for dialog design. In *Proc. INTERACT'87*, pages 431–436. Elsevier Science Publishers B. V. (North-Holland), 1987.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

A 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 quiescent, C does not initiate communication; *i.e.*, C does not make any calls (section 3.3.2) 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.

B Implementation Considerations

This appendix documents a number of implementation considerations that were identified during the creation of the specification. It does not form part of the specification proper.

B.1 Component Implementations

For components, the Workspace client programmer is responsible for providing implementation definitions (*e.g.*, class definitions with the desired semantics) for part of the required implementation. For connectors and ports, the Workspace runtime system provides one or more default implementations.

B.2 Unique Identifiers

In the definition of the evolution calculus in sections 4 and 6, unique identifiers were provided externally with each operation. In implementations of the Workspace Model we expect that globally unique identifiers would be provided automatically by the run-time system.

B.3 Failure Reports

Operation requests in the evolution calculus may fail, resulting in an identity on the architecture (see section 4). In this case, a failure report will be provided to the operation's requester. The exact form of the failure report is system-specific, however failure reports should be in the form of a exceptions in systems that support them.

B.4 Custom Implementations of Infrastructure Components

Implementations of the Workspace Model should also allow client programmers to provide their own infrastructure component implementations (see section 5.4) in order to provide specialised services or to improve performance by making use of application-specific semantics.

B.5 Attributes as Implementation Hints

A developer may wish to provide implementation hints to the run-time system. For example, the developer may desire that a particular synchronization group be implemented by replication and that another be implemented by locking.

This information may be communicated to the run time system via attributes set on conceptual level workspace elements. In the example above, the synchronization group might have a desiredImplementation attribute, the value of which was a reference to a specialised CCCM component that implemented the desired synchronization strategy. Implementation hints would also be used to request the use of custom implementations of infrastructure components, as discussed in the previous section.

Details of this approach will be determined based on implementation experience.

B.6 Convenience Operations

The conceptual and implementation level evolution calculi may be thought of as "assembly languages" for operations on architectures. We expect that actual implementations will define convenience operations that are compositions of evolution calculus operations. For instance, at the conceptual level it might be reasonable to have a convenience operation $connect(port_a, port_b)$ which implicitly creates a connector of the appropriate kind and attaches it to the named ports. In our own implementation we intend to let experience guide the selection of such operations.

Likewise, the reflection operations described in sections 4.11, 6 and 7.7 constitute only a minimal set and would likely be cumbersome to use. We anticipate that any implementation of the workspace architecture would provide convenience reflection operators, or perhaps a general query facility.

B.7 Port Creation and Use

The object requesting the creation of a port is responsible for binding the port to appropriate code (*e.g.*, a syntactic variable) within the component. Ports not bound to code are effectively useless. Any calls or messages originating from the bound code point must be within the defined vocabulary for that port, otherwise a run-time exception or equivalent will be raised. Likewise, at the target end, if a component does not support a method the vocabulary of an attached call port, that method may also result in a run-time exception or equivalent.

B.8 Channels

The channel construct discussed in section 5.4.2 may be implemented using a centralised message router, a distributed group communication service such as Spread [1], or by any other appropriate means.

Implementation using a group communication service is an appropriate design strategy since this allows the Workspace toolkit developer to abstract the network, provide for efficient message delivery, provide ordering and reliability guarantees for the messages transmitted, and enforce appropriate distributed system semantics for group joining and leaving, and response to partial failure.

Where channels are expected to provide a particular message ordering policy (local FIFO, causal, total, *etc.*) this would be specified via an implementation hint (see section B.5) on the conceptual-level subscription or synchronization connector.

B.9 Implementing Refinement

The refinements presented in section 7 define a space of allowable implementations for any conceptual architecture but give no guidance as to which implementation would be most appropriate for a given situation.

In practice we expect run-time systems to derive implementations dynamically and incrementally, in response to changes in either the conceptual level architecture or the available implementation resources. The derived implementation architectures must correspond to allowed refinements in accordance with the rules specified in this section.

As a long-term goal, the Workspace run-time system is intended to offer fully automatic, dynamic adjustment of the implementation level architecture in response to changing user needs, user mobility, and the characteristics of available input, output, computational and storage devices and their interconnections. In the short term, the run-time system will be based on simple heuristics and may require manual intervention in order to achieve optimal performance (for some definition of "optimal").

B.10 Concurrency Control Issues

In multi-user interactive systems, concurrency control is required at a number of levels.

Interactive systems should not behave in ways that violate either semantic consistency or user expectations. For example, in a drawing editor the system should not allow two users to concurrently move a single object in different directions. This kind of concurrently control is specific to the interface and underlying data, and must necessarily be addressed by the developer of the interface. In this example, the developer might provide a mechanism requiring a user to acquire an exclusive "move lock" on an object before being allowed to move it.

At an intermediate level, the system should support transactions (multi-valued sequences of requests and updates) on the level of atomic user actions in the interface.

At the lowest level, it is necessary that mutually incompatible method invocations not be active within a single component, as this could lead to invalid component state. In the example above, ensuring that two users not simultaneously acquire a move lock requires that multiple threads not have conflicting access to the lock variable. One approach would be to require the component developer to specify resource locking strategies in the component code itself, as is commonly done in Java using the `synchronized` keyword.