A Calculus for the Refinement and Evolution of Multi-User Mobile Applications

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

[1] Electrical and Computer Engineering, Royal Military College of Canada, Kingston, Ontario, Canada K7K 7H6. `greg.phillips@rmc.ca`
Voice: +1-613-541-6000 ext. 6491   Fax: +1-613-544-8107
[2] School of Computing, Queen's University, Kingston, Ontario, Canada.
`[graham|wolfe]@cs.queensu.ca`

Abstract. The calculus outlined in this paper provides a formal architectural framework for describing and reasoning about the properties of multi-user and mobile distributed interactive systems. It is based on the Workspace Model, which incorporates both distribution-independent and implementation-specific representations of multi-user and mobile applications. The calculus includes an evolution component, allowing the representation of system change at either level over time over time. It also includes a refinement component supporting the translation of changes at either level into corresponding changes at the other. The combined calculus has several important properties, including locality and termination of the refinement process and commutativity of evolution and refinement. The calculus may be used to reason about fault tolerance and to define the semantics of programming language constructs.
Keywords: software architecture, model-based design, groupware, mobile applications, Workspace Model

1 Introduction

Recent years have seen the introduction of numerous architectural models and associated tools for the high level design of interactive systems. Interest in architectural models has continued with the advent of new styles of user interface such as groupware systems that allow users to collaborate asynchronously or in real time, mobile systems allowing users access to a wide variety of devices such as tablet PCs, PIMs and smart phones, and ubiquitous systems which are sensitive to the user's context.

While many architectural models have been proposed (*e.g.,* [2, 3, 4, 7, 8]; see [9] for detailed discussion), there is as yet little underlying theory to explain their semantics, to allow comparison of different models or to serve as a guide for implementing tools or applications based on these models. In this paper we provide such a theory, called the Workspace Model, formalized via an evolution calculus and a refinement relation.

The full formal specification of the model is available as [10]. In this paper we outline the formal underpinnings of the Workspace Model and illustrate its utility via two applications. The novel aspects of the model include:

- Its recognition that the structure of interactive systems change over their run times as the users of the system change, as their intentions change and as the physical run time platform changes.
- Its linkage of a conceptual architectural view to an implementation architecture, allowing high-level design to be mapped in a principled manner to a run time implementation, while avoiding premature commitment to a distributed implementation.
- Its treatment of partial failure as a first-class consideration in architectural models.
- Its ability to represent the use of multiple devices and device types in a single interactive session.

The paper is structured as follows. In section 2 we provide an overview of the Workspace Model, including the key elements of the conceptual level, the implementation level, the refinement rules and the evolution calculus. Section 3 presents key properties of the model. Finally, in section 4 we present two applications of the model, the first characterizing mechanisms for dealing with partial failure, and the second showing how the model can be used to give the semantics of a language supporting the development of groupware applications.

2 Elements of the Model

In this section, we provide an overview of the Workspace Model including its two architectural levels, the evolution operations, and the refinement relation.

Figure 1 shows the key elements of the workspace model and how they relate to one another. Architectures may be expressed at a conceptual level and at an implementation level. A conceptual architecture expresses the structure of the elements making up an interactive system, but does not specify how they are to be implemented as a distributed application. Conceptual architectures are illustrated in section 2.1 using a small example.

A refinement relation R composed of individual refinement rules r maps conceptual architectures to implementation architectures. In general, a given conceptual architecture can be mapped to many implementation architectures. R therefore captures a space of possible implementations. $R(c, i)$ indicates that i is a valid refinement of c. Refinement is discussed in section 2.2 and implementation architectures are presented in section 2.3.

Finally, an evolution operator e expresses runtime evolution of conceptual and implementation level architectures. In figure 1, $e_c(c)$ produces c', a modified conceptual architecture, and $e_i(i)$ produces i', a modified implementation level architecture. Evolution is discussed in section 2.4.

$$\begin{tikzcd}
c \arrow[r, "e_c"] \arrow[dd, "R"'] & c' \arrow[dd, "R"] & \text{conceptual level} \\
 & & \downarrow \text{refinement} \\
i \arrow[r, "e_i"'] & i' & \text{implementation level} \\
\xrightarrow{\text{evolution}} & &
\end{tikzcd}$$

Fig. 1. Key elements of the Workspace Model.

The two architectural levels have previously been described at greater length in [11][1] and the full definitions of evolution and refinement may be found in [10]. Here we present the architectural levels in the form of a small example and provide sample evolution operations and refinement rules, in order to give a flavour of the complete system.

2.1 Conceptual Architecture

We present the conceptual level using an example in which a presentation is attended by multiple audience members, some local and some remote. The presenter has a private view of the presentation including the current slide and associated notes, as well as controls affecting the presentation flow. Local audience members sit in an auditorium, view the presentation on a large screen, and hear the presenter's voice directly. Remote audience members view the presentation on their personal computers and listen to the presenter by means of a networked voice system. The remote audience members' slide view shows the same slide as the local audience members'.

A conceptual architecture supporting this example is depicted in figure 2, using the Workspace notation. There are two *workspaces*, indicated by the dotted lines, which represent distinct contexts of use. Within the workspaces we may find *people*, software and hardware *components* (rectangles; only software components are illustrated), and *connectors* between them. Components are attached to connectors at *ports* (circles, which we occasionally omit from the diagrams where this has no effect on semantics). Workspaces may also contain *nodes*, which are identifiable computational elements such as the presenter's computer. A node's presence in a workspace indicates that components in the workspace may be implemented on the node; nodes thus provide a bridge between the conceptual and implementation levels.

The conceptual level includes three kinds of components. *Actors* (not shown) have independent threads of control and may therefore initiate activity in a workspace. *Reactors*, such as the presenter view and audience view components, react to input calls or messages arriving on connectors but are otherwise inert. Finally, *stores* such as the presentation and voice components are purely passive and are analogous to the

[1] The visual notation used in the workspace model has been modified from that used in [11] based on the results of an informal usability study.

model of the model-view-controller architecture [6], with the added feature that they may be used to represent shared state.

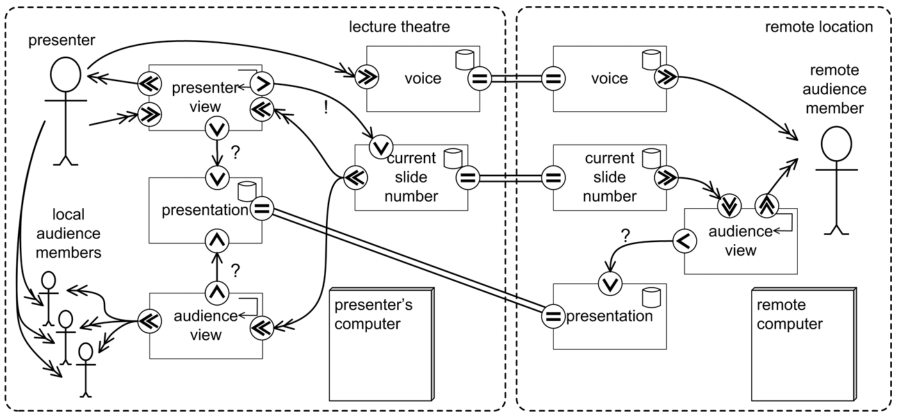


Fig. 2. Conceptual level view of the presentation example.

The conceptual level also includes three kinds of connectors. *Calls* (single arrow head) are point to point, blocking, and analogous to procedure calls. Calls which modify the state of the called component are *updates* and are indicated by an ! annotation; calls which return values are *requests* and are indicated by a ?. Complementing calls are *subscriptions* (double arrow head), which are asynchronous and provide multi-source to multi-target *message* delivery. The third connector type is the *synchronization* (double line), which provides an abstract representation of data sharing. Stores that are *synchronized* (such as the presentation components in figure 2) respond the same way to requests and emit consistent message streams; that is, they may be thought of as representing "the same thing."

The architecture depicted in figure 2 supports our example as follows. The presenter acts as the source of three subscriptions, two delivering voice to the local and remote audience members and one providing mouse and keyboard inputs to the presenter view component. Inputs to the presenter view may result in modifications to the state the current slide number via the call connector. Since the two current slide number components are synchronized, a change to the current slide number in the lecture theatre workspace will result in a message being delivered to the presenter view and to both audience views via the outgoing subscription connectors. The presenter view and audience view components will react to this message by querying their respective presentation components for the current slide and displaying it on their associated output devices.

In summary the conceptual level architecture deals with many of the issues that arise in modeling modern interactive systems. Users may have differing contexts, may use different devices, may be collocated, distributed, or even mobile, and the structure of the collaboration may change in real time as participants enter and leave and as the users' goals change.

2.2 Refinement

A traditional problem with the use of software architectures in interactive system 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 (e.g., [3, 7]). Architectural descriptions rarely address how to refine a conceptual architecture to an implementation, leaving it to users of the architectural style to determine how best to do so.

The Workspace Model provides a refinement relation *R* that precisely defines the legal implementations of any conceptual architecture. This helps developers by providing rules that they can follow in the implementation of their conceptual architectures, helping in the transition from conceptual to implementation view. The refinement relation also helps toolkit builders by providing precise semantics for implementation decisions embodied in the toolkit.

The refinement relation defines the space of possible implementations for any given conceptual architecture. The relation is specified via a graph grammar showing how conceptual elements may be rewritten to implementation elements. The graph-grammatical rules each specify one step of a refinement. The refinement relation *R* is therefore the reflexive transitive closure of this set of refinement rules.

Figure 3 shows example refinement rules for the implementation of components on nodes. Analogous rules specify the possible refinements for ports and connectors. All 29 refinement rules are specified in [10].

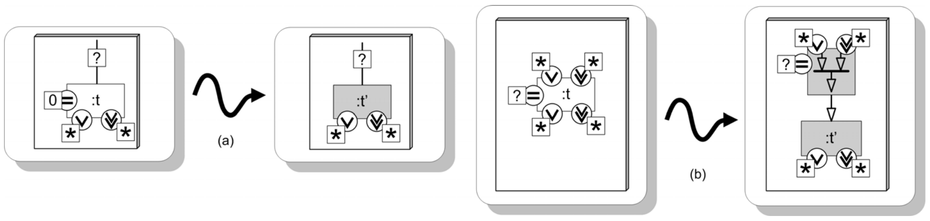


Fig. 3. Example refinement rules for implementation of components.

A refinement rule consists of a left-hand side *pattern* that may be matched in the current architecture. When a pattern is matched, 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".

Rules (a) and (b) specify the implementation of an anchored component of type t onto a node. Rule (a) says that an anchored component with no synchronization port and at most one incoming connector (call or subscription) may be refined to an implementation-level component (shaded) with corresponding type t'. The rule specifies that the matched component may have zero or more outbound ports of the call and subscription types, which are preserved in the implementation component.

Rule (b), whose pattern will also match any architecture satisfying rule (a)'s pattern, specifies that any anchored conceptual component may be implemented by a combination of an implementation level component matching type t' and a *concurrency control and consistency maintenance* component (CCCM) which mediates conflicting calls and messages. The conceptual component's incoming and synchronization ports are allocated to the CCCM while its outgoing ports are allocated to the t' component. In the case of infrastructure components inserted into an architecture by a refinement, the rule does not specify precisely how these components are implemented. For example, a CCCM may be based on locking or on one of may optimistic concurrency control protocols [5].

These examples give a flavour of the rules making up the refinement relation. Other rules match a partially refined architecture and refine it to more completely refined architecture, ultimately resulting in an implementation architecture.

2.3 Implementation Architecture

As shown in section 2.2, a conceptual architecture is refined to an implementation architecture by the application of a series of rules. These 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.

Figure 4 shows one valid implementation of the conceptual-level architecture of figure 2. In figure 4 we have instantiated components on the two computational nodes that were shown as available in figure 2.

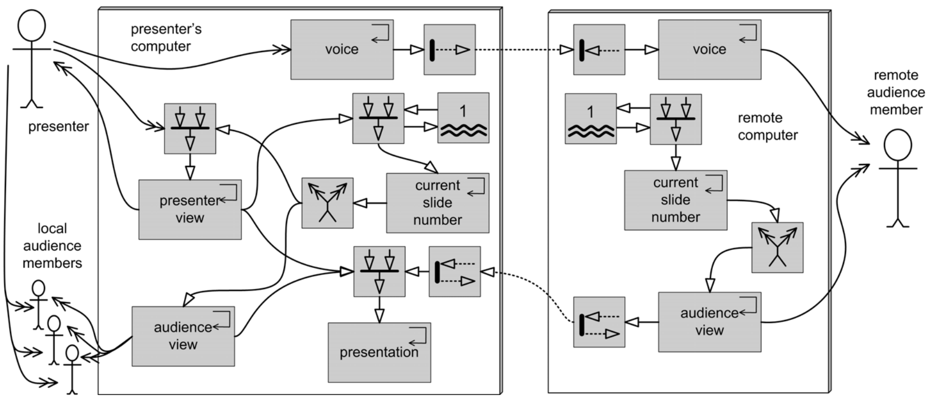


Fig. 4. One possible implementation of figure 2.

In this example we have made the decision to implement the presentation store solely on the presenter's node. The remote node's audience view component accesses it over the network, using *transceiver* components to marshal and unmarshal calls and return values. The network link is indicated by the dashed arrow.

Conversely, we have fully replicated the current slide number store, maintaining one copy on each node. The two copies' CCCMs maintain consistency by the execution of some replica consistency maintenance algorithm, *e.g.* locking or a distributed operation transform [12]. The two CCCMs communicate by means of a *shared channel*, implemented using *channel endpoints* (the components with the wavy lines). Channel endpoints are provided by many group communication frameworks, including for example Spread [1] and Horus [13], and provide a useful multipoint message distribution service with ordering and performance guarantees.

Finally, to implement subscription connectors, which have an asynchronous semantics and many-to-many topology, we introduce *message broadcaster* components in the connectors' implementation.

It is important to note that figure 4 represents just one valid refinement of the conceptual architecture of figure 2. For example, different decisions could have been made on the allocation of components to nodes or on the replication strategy for shared data; similarly a cache might be introduced on the remote node to retain local copies of the slides as they are taken from the presentation, improving responsiveness when previously viewed slides are revisited.

2.4 Evolution Calculus

An important characteristic of modern interactive systems is their support for 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 while distributed system changes typically result in change at the implementation level. When a change occurs at one level, the refinement rules are used to find a sequence of evolutions at either or both levels such that the refinement relation *R* between the levels is restored.

The evolution calculus consists of a set of operations at each of the two levels. Operations are defined using a graph grammar notation similar to that used for refinement rules. Each 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 operations are shown in figure 5, one at the conceptual level and one at the implementation level. There are a total of 49 operations in the calculus [10].

Figure 5(i) partially defines the attach operation for synchronization groups. The operation's signature is attach(A, k, p) where A is the architecture 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 box) and 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.

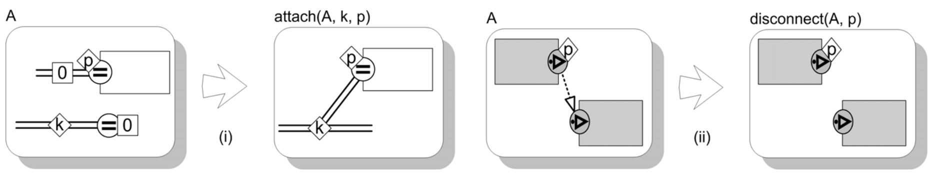


Fig. 5. Sample evolution specifications.

Figure 5 (ii) shows a disconnect operation at the implementation level. This evolution might be invoked in response to a conceptual level change or as a result of a network failure. Note that whereas conceptual level connectors exist independent of any connections, implementation level connectors are destroyed by a disconnect.

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. In this way, traceability between the two levels is retained. 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 Properties

As stated in the introduction, the Workspace Model has been designed to possess three properties that are critical to its practical application. These properties are straightforward to prove for our formalism via structural induction over the refinement rules and evolution calculus operations.

Refinements are local. This property states that the composition of refinement steps is commutative. That is, if A is an architecture and r_1 and r_2 are refinement rules such that r_1 and r_2 match non-overlapping portions of A, then $r_1(r_2(A)) = r_2(r_1(A))$.

The primary consequence of this property is that refinement rules can be applied locally (without reference to the context of their matches), whether the refinement is being carried out statically by a developer or automatically by some runtime agent.

Refinements Terminate. This property states that any non-trivial refinement sequence will eventually lead to a ground architecture. An architecture is *ground* if it consists only of implementation level elements and no further refinement rules match. A refinement r over a is *non-trivial* if r's pattern matches in a.

More precisely, the termination property states that for all architectures a, there exists some number n such that for every set of non-trivial refinements r_1 through r_n, $r_n(r_{n-1}(\ldots(r_1(a)\ldots)$ is ground.

This property is critical to automated implementations of refinement as it implies that any refinement sequence will eventually lead to an implementation architecture. Of course, the termination of refinements does not guarantee that all refinements will be appropriate choices!

Evolution and Refinement is Commutative. This property states that following an evolution in the conceptual or implementation architectures, there exists some sequence of evolutions at the conceptual or implementation level or both, such that the new conceptual architecture refines to the new implementation architecture.

More precisely, for any e_{i0} or e_{c0} resulting in a new combination of c_0 and i_0, there exists some finite set of e_i and e_c such that $R(e_{cm}(e_{cm\text{-}1}(\ldots(c_0)\ldots), e_{in}(e_{in\text{-}1}(\ldots(e_i(i_0)\ldots))$ where R is the reflexive transitive closure of r.

This property is trivially true as the sequence of evolutions could simply be to delete all workspace elements at both levels. However, as shown in section 4, achieving this traceability property with minimal changes to the architecture is the root of a good fault tolerance strategy.

4 Applications

In previous work, we reported on the use of the Workspace Model to support scenario modeling and automatic runtime distribution of real-time groupware systems [11]. In this section we present two further applications of the Workspace Model: reasoning about fault tolerance and the use of the model to define an extension to the C++ programming language.

4.1 Fault Tolerance

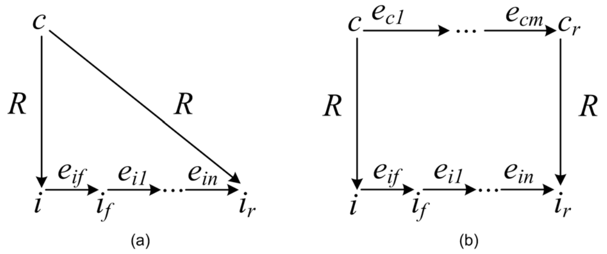
In the Workspace Model, a partial failure manifests itself as one or more un-requested evolutions at the implementation level. For example, the loss of contact with a node will initially be recognized as the disconnection of any remote connectors targeting components on that node.

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 the 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. The Workspace Model representations of these two possible responses are illustrated in figure 6.

Both parts of figure 6 begin with an implementation level architecture i that is a the current conceptual architecture c; that is, $R(c, i)$. The initial failure is represented as a series of evolutions e_{if} at the implementation level that result in a new implementation architecture i_f where i_f is not a valid refinement of c.

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

Where restoral is not possible, it is generally necessary to modify the conceptual architecture in some way in order to effect a recovery.



$$\begin{tikzcd}
c \arrow[d, "R"'] \arrow[drrrr, "R"] & & & & \\
i \arrow[r, "e_{if}"] & i_f \arrow[r, "e_{i1}"] & \cdots \arrow[rr, "e_{in}"] & & i_r
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
c \arrow[d, "R"'] \arrow[rr, "e_{c1}"] & & \cdots \arrow[rr, "e_{cm}"] & & c_r \arrow[d, "R"] \\
i \arrow[r, "e_{if}"] & i_f \arrow[r, "e_{i1}"] & \cdots \arrow[rr, "e_{in}"] & & i_r
\end{tikzcd}$$

(b)

Fig. 6. Workspace model representations of (a) restoral and (b) recovery.

This is represented in figure 6(b). Here, a series of evolutions e_{c1} through e_{cm} are applied to c while a series of evolutions e_{i1} through e_{in} are applied to i_f, such that $R(e_{cm}(\ldots(e_{c1}(c)\ldots),\ e_{in}(\ldots(e_{i1}(i_f)\ldots))$.

As noted in section 4 it is always possible to find such a series of evolutions, 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. However, figure 6(b) is far from vacuous. Rather, it suggests initial criteria on which the appropriateness of a restoral may be judged: the number (and nature) of conceptual level evolutions required to reestablish coherence between the conceptual and implementation levels.

4.2 Extension to C^{++}

A second application of the Workspace Model is its use to provide a rigorously defined alternate semantics for an existing programming language. Currently we are working with C^{++}. The aim is to enable conventionally written programs to be used in a mobile multi-user distributed setting with little or no modification, while maintaining a predictable, precise, "natural" semantics.

As an example, consider the C^{++} code `C *x = new C();` executed in the context of some object `q`. The normal C^{++} semantics of this statement creates a new instance of class `C` and declares a pointer `x` within the scope of `q` such that `x` refers that instance.

However, this is inadequate of we wish to be able to migrate the new instance across process boundaries or to refer to it from multiple processes. In effect, what we want is a pointer equivalent that allows the target to be remote and mobile, but with a well-defined semantics and fault tolerance strategy. This is precisely what the Workspace Model's call connector provides, so we can define the semantics of this code as shown in figure 7.

The text in the box shows the series of Workspace-level evolution operations defining the semantics of the given code. In the definition the architecture is an implicit parameter to each evolution and the semi-colons represent composition.

The diagram on the right side of the figure illustrates the complete effect of this series of evolutions on an initial architecture. A new conceptual level component of the appropriate type is created, source and target ports for a call connector are created on the appropriate components, and a call connector is created and attached to those ports.

```
Let
   q be the component in which the code is executed
   w be the workspace containing q
   $t_C$ be the conceptual component type for class C
   r, k, $p_1$ and $p_2$ be identifiers unused in the initial
      architecture

Then C *x = new C(); executed in q is defined as:
   createComponent($t_C$, w, r) ;
   createCallSource(q, $p_1$) ;
   createCallTarget(x, $p_2$) ;
   createCall(w, k) ;
   attach(k, $p_1$) ;
   attach(x, $p_1$) ;
where the port $p_1$ is associated with the variable x.
```

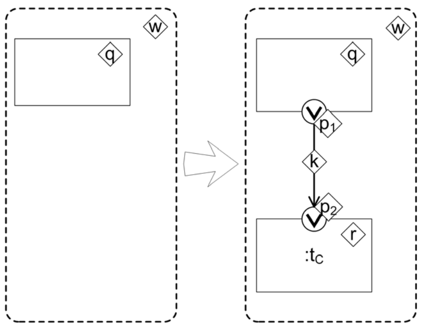


Fig. 7. Definition of the C^{++} new operator in terms of workspace conceptual evolutions.

As discussed in section 2.4, this change at the conceptual level will cause the current implementation architecture to no longer be a valid refinement of the conceptual level. The run-time system will respond by computing and executing a series of evolutions at the implementation level to reestablish the refinement relation. In the end result the syntactic variable x will be a pointer either directly to the implementation of the new component (if it was in fact instantiated on the same node as q) or to a run-time provided component that acts as a remote proxy.

The implementation of these semantics for C^{++} is provided by means of a pre-processor and a runtime support system.

Conclusion

In this paper, we have presented the Workspace Model and its associated refinement relation and evolution operations. The Workspace Model provides precise semantics for reification of conceptual architectures as distributed systems, and for the sorts of runtime evolution that occur over the lifetime of groupware or mobile applications. We have shown two applications of the model, one characterizing fault tolerance in distributed interactive systems, and the other providing semantics of a C^{++}-like programming language**.**

Two implementations of toolkits based on the workspace model are underway, one in Python and one in C^{++}. With these toolkits, we hope to gain further experience with the expressiveness of the workspace model, and to determine whether the refinement relation and evolution calculus provide an effective basis for formally-specified implementation of distributed interactive systems.

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. L. Braubach, A. Pokahr, D. Moldt, A. Bartelt, and W. Lamersdorf. Toolsupported interpreter-based user interface architecture for ubiquitous computing. In *Proceedings of the Ninth International Workshop on Design, Specification and Verification of Interactive Systems* (DSV-IS '02), pages 89–103. Springer-Verlag, 2002.
3. P. Dewan. Architectures for collaborative applications. In M. Beaudouin- Lafon, editor, *Computer Supported Co-operative Work*. John Wiley & Sons Ltd., January 1999. ISBN 0-471-96736-X.
4. W.K. Edwards. *Core Jini*. Prentice Hall PTR, 2nd edition, 2000. ISBN 0-13089-408-7.
5. 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.
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. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. R. van Renesse, K.P. Birman and S. Maffeis. Horus, a flexible group communication system, *Communications of the ACM*, April 1996.