SUPPORTING COLLABORATIVE DESIGN

Wang Yanjie
CAD Center, Southwest Jiaotong University (SWJTU), Chengdu, Sichuan, PR China
wangyjwx@163.net

Chris George
United Nations University International Institute for Software Technology (UNU/IIST), Macau
cwg@iist.unu.edu

Abstract Collaborative systems provide a rich but potentially chaotic environment for their users. During the collaborative design process, when different users share a common object, concurrency control is necessary to keep the shared object consistent. In this paper, we have developed a model for supporting concurrency control in CSCW applications. We also include version control, allowing versions of both the design entities and their relationships. Consistency in collaborative design is important: the use of the RAISE formal method provides confidence that the transaction mechanism fulfils the consistency requirements. We also generate a prototype from the specification, with a graphical output to show the structure of the design, the entities available, and who is currently locking what.

Keywords: Concurrency control, version control, CSCW, consistency, RAISE, formal methods

1. Introduction

The rapid advances in information technologies, particularly those comprising the Internet and the WWW, bring significant changes. More and more teamwork is required by many organisations to achieve their goals. With the improvement of computer network and multimedia technology, specialists from different domains can break through the information blocks of time, distance, and different kinds of computer devices, working together virtually.

[1]This work was done during the first author's fellowship at UNU/IIST.

To support this kind of application, it is necessary to develop a computer environment suitable for group cooperation. So, in the middle of 1980s, the term Computer Supported Cooperative Work (CSCW) was coined, which is concerned with how groups of people work together using computer technology (Young, 1996). Typical topics include: electronic mail, asynchronous conferencing, group decision support system, collaborative authoring systems, screen sharing software, video conferencing, and real-time shared applications, such as collaborative writing or drawing.

In CSCW, the electronic whiteboard, a shared, virtual workspace, is an important kind of communications medium. A whiteboard allows two or more people to view and draw on a shared drawing surface even when they are at a distance. Every designer has a view to the whiteboard. Different designers' views may be different, or the same, or only partly the same. Each designer only can edit the objects inside his own view. All the users can modify the whiteboard concurrently.

Version control, a special kind of software used to track and manage changes, is an important problem in CSCW applications. A versioning system typically allows multiple versions of an object to be simultaneously active. The introduction of version mechanisms can inform designers of: the presence of other designers who are currently investigating a version graph, the creation and availability of new versions, and changes to the content of current versions. In an *uncontrolled* site where multiple users have access rights to edit and contribute, the potential for conflict and problems arise — more so when these users work from different offices at different time.

When two or more users collaborate to manipulate a shared object, there may arise occasions where one user's action needs to be blocked so that it does not interfere with another user's action. A CSCW system must support individual or collective work on a shared object. An important function to handle the access to the shared objects is concurrency control. Concurrency control handles the concurrency access of multiple users to shared information in order to preserve information consistency. The choice of concurrency management strategy can have a significant impact on the styles of interaction which an application can support.

In this paper, we construct a collaborative design model in which we are concerned with how to implement concurrency control and version control. We do not construct directly the collaborative model. Instead, we obtain it step by step from the main requirements. This process helps us construct the model using a formal modelling technique.

The paper is structured in the following way. Section 2 briefly introduces the RAISE formal method. Then in section 3 we model the design process in three stages: section 3.1 describes the architecture of the basic model; section 3.2 adds the concurrency control transaction mechanism; section 3.3 includes the

version control mechanism. In each model, we give a brief introduction of the formal descriptions of the model and describe some operations based on the model. Section 4 presents a prototype implemented from an instantiated specification model. Finally, in section 5 we give some conclusions, describe related work, and talk about thoughts for future work.

2. RAISE

RAISE (Rigorous Approach to Industrial Software Engineering) was developed by the ESPRIT projects RAISE (1985–90) and its successor LaCoS (Large scale Correct Systems) (1990–5). The RAISE Specification Language (RSL) is described in (RAISE Language Group, 1992) and the method in (RAISE Method Group, 1995). These projects also produced a set of tools for RAISE, supporting inter alia type checking, pretty-printing, translation to Ada and C++, and theorem proving. More recently a collection of more portable tools has been developed at UNU/IIST (Chris George, 2001), and these were used for this paper.

RSL is a *wide-spectrum* language. Possible styles range from abstract types (sorts) plus signatures and axioms to completely concrete descriptions amenable to automatic translation to programming languages. Descriptions may be applicative or imperative, sequential or concurrent (for which RAISE includes a process algebra with channels and value-passing). RSL supports large specifications with modularity, including parameterised modules.

The style of specification used in this paper is fairly concrete. A few remarks about the type definitions used may be helpful to readers. Type definitions are introduced by the keyword **type**, and in this paper are of three kinds. An *abbreviation definition* just introduces a type identifier as an abbreviation for some type expression, as in

type
  Collection′ = Entity_Id $\overrightarrow{m}$ Entity,
  Collection = {| c : Collection′ · iswf(c) |}

which defines *Collection′* as an abbreviation for a finite map (many-one relation) between *Entity_Id*s and *Entity*s. Maps may be applied to values in their domains just like functions. Other possible type constructurs include finite sets (**-set**), finite sequences (*) and products (×).

The type *Collection* is defined by another abbreviation definition, involving a *subtype*: *Collection* contains those values in *Collection′* that satisfy the predicate *iswf*.

Records are similar conceptually to records in programming languages, except that the field names (called *destructors* in RSL) are functions. For example, we define an *Entity* as a record with three components:

type
  Entity ::
    attrs : Attributes $\leftrightarrow$ re_attrs
    parent : Super
    count : Nat1,
  Super ==
    super_project(proj : Project_Id) |
    super_part(part : Entity_Id)

Here *parent*, for example, is a function from *Entity* to *Super*, used to extract the component. Record components can optionally have a *reconstructor*, introduced by $\leftrightarrow$, which provides a function to change one component of a record. For example, if *e* is an *Entity* value and *a* an *Attributes* value, the expression *re_attrs(a, e)* is a *Entity* value in which the attrs component is *a* and the others are *parent(e)* and *count(e)*. *Nat1* is defined as the type of positive natural numbers.

The type *Super* is a *variant* type (with two variants). A *Super* value may be either *super_project(pid)* for some *Project_Id pid*, or *super_part(eid)* for some *Entity_Id eid*.

3. Modelling Design

3.1. Architecture of the Basic Model

3.1.1 Architecture. At the beginning, only one designer creates a whiteboard. There is no access problem: s/he can operate on the whiteboard as s/he likes.

First, we presume that a repository will contain a number of largely independent schemes which we term *project*s. To model a project, we allow a number of design components, which include the possibility of only one, or even none when the project is first established. We call design components *entities*.

Then we need an appropriate structure for entities. We think first that each entity should belong to precisely one project. If an entity appears to belong to two projects it is probably better to copy and so allow for the possibility of independent change. (We also allow for entities to have *multiplicities* (counts), allowing for something to have a number of identical components.)

Second, we think that a hierarchy (tree) structure is appropriate. Note that this allows a completely flat structure, in which each entity belongs only to a project, and there are no relations between entities. But it also allows much richer structures. The particular characteristics of trees are that each node has precisely one parent node, and that the parent relation is acyclic. This is a common means of structuring, and eases the problem of modelling change control, as we shall see, compared with a more general graph. This gives us

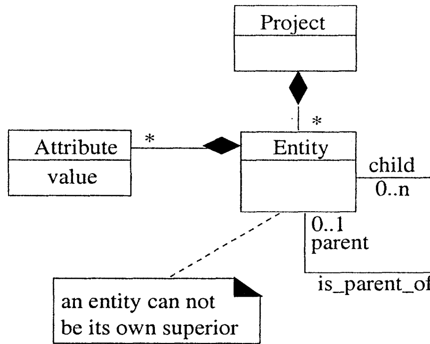

Figure 1. The relationships between the objects of the basic model

a total, many-one *parent* relation. We call its transitive closure the *superior* relation. In particular, each entity has a unique superior project. We use the term *node* to mean project or entity.

We allow entities to have *attributes*. We extend the *superior* relation by saying that the superiors of an attribute are the attribute's entity, and that entity's superiors. We extend the term *node* to mean project, entity, or attribute.

Next we consider how to identify projects, and entities. We could use the tree structure to generate path names (like paths in a computer file system) in which entity names are only required to be unique between siblings, but design projects are not likely to have sufficiently large numbers of entities to make this necessary, so we require entity identifiers to be unique within projects.

We use a UML class diagram in Figure 1 to show the relationships between the objects of the basic model.

3.1.2 Formal Description. Most of the types used in this model were described in section 2. We also have

type
Project = Collection,
Projects = Project_Id $\overrightarrow{m}$ Project,
Attributes = Attribute_Id $\overrightarrow{m}$ Val

All the identifiers, such as *Attribute_Id*, and *Val* are abstract types. All the projects are stored in the repository *Projects*.

There are some well-formedness conditions on collections, defined by the predicate *iswf*. In particular, a parent which is an *Entity_Id* must exist in the collection, and the transitive closure of parent (the *superior* relation) must be acyclic. Functions (and constants) are introduced by the keyword **value**:

value

iswf : Collection′ → **Bool**
iswf(c) ≡
  (∀ id : Entity_Id ·
    id ∈ c ⇒
      parent_exist(parent(c(id)), c) ∧ ∼issuptrans(id, id, c)),
issuptrans : Entity_Id × Entity_Id × Collection′ → **Bool**
issuptrans(id, id′ , c) ≡ ... /* id is a superior of id′ */

3.1.3 Operations. On the whiteboard, the designer can add, delete or copy entities, change their counts, and edit their attributes.

We must note that the deletion of an entity does not mean only removing the entity itself. If so, the sub-entities of it will lose their parent so that they will lose their connections with the tree structure. So removing an entity means removing not only the entity itself but also all its sub-entities.

Also, the copying operation means copying a tree, namely, copying the entity and all its sub-entities.

3.2. Supporting Multiple Designers

If several people join in the shared whiteboard and want to edit the same item (entity, attribute or project), inconsistency problems will arise. Our former model can not meet the requirements of multiple designers. In this section, we propose a new model.

The main requirement for collaborative design is that there should only be one person able to change an entity at any one time. This is not the only choice: multiple changes by different designers followed by some reconciliation procedure is also possible, but is more complicated and probably unnecessary. In a CSCW environment, the data may include the aggregations of structured data (graph, software group, etc.), and the relationships between data may become more complicated (Chen Qingzhang et al., 2000). And the two phase process of multiple change followed by reconciliation seems inappropriate for interactive whiteboard collaboration. So a system of locking a part by at most one designer is proposed.

We need to consider how the locks interact with the tree structure. Locking allows change, but what do we mean by change? We need to consider editing, creation, and deletion. Specifically:

1 Is deletion a change to its parent?
2 Is creation a change to its parent?
3 If X is the parent of Y, does the right to change to X allow changing Y?
4 If A is an attribute of X, does the right to change to X allow changing A?

5 Generalising the previous two, if X is superior to Z, does the right to change to X allow changing Z?

To all of these we answer “yes”. This has consequences for locking. We cannot allow, for example, a lock by one user on an entity and a lock by another user on its parent, or our main requirement would be violated. We impose the constraint that

- a lock on a node implies no lock on any superior node

Another way to see this is to say that locks are inherited, so this is the rule needed to ensure that there is at most one lock on a node.

Now we can be sure that the locking mechanism ensures that we meet our main requirement of at most one user being able to change something at any one time.

Note that it must be possible to lock a whole project, or an entity could not be created or deleted at the top level. Note also that it is allowed for different users to lock different attributes at the same time, since one attribute is never a superior of another.

In addition, we want to support a system of access rights. We could include read access, if projects might be secret, but here we concentrate on change access. We propose that each project, entity and attribute has a set of designers associated with it, and that only someone in the appropriate set of designers can lock, and hence change, an item.

How do access rights interact with the tree structure? It is tempting to think that sets of designers at lower levels should be subsets of those at higher levels, but this is in fact not appropriate. It is higher levels that need more protection, because changing them has wider consequences. So sets of users will tend to get larger rather than smaller as you move down the tree.

But it does seem appropriate that a user who has the right to change an entity should have the rights to change its children. So we propose

- a lock can be granted only to someone with access rights to this node or a superior one

In other words, access rights are also inherited.

Figure 2 shows the relationships between the objects of the new model.

3.2.1 Formal Description. The types for the revised model are simple additions to the first model: projects and entities have additional components recording designers (sets of user identifiers) and locks (nil or locked by a user).

The well-formedness conditions are extended to check that a lock implies no lock on any inferior or superior node and the appropriate permission. See (Wang Yanjie et al., 2001) for details.

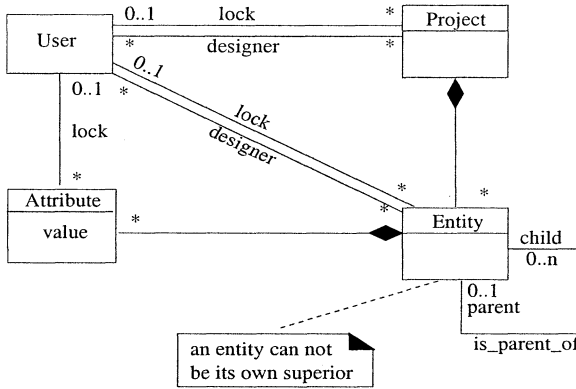

Figure 2. The relationships between the objects of the new model

3.2.2 Operations. In the new model, the designers can perform the basic operations introduced before. But all the basic operations need to maintain the consistency of the design. To achieve this, we need to define other operations, such as getting a lock or releasing a lock on an attribute, or an entity, or a project.

The operations also require preconditions to ensure that the reult of the operation will be well-formed.

3.3. Adding Version Control

Version control, a special kind of software used to track and manage changes, is an important problem in CSCW applications. A versioning system typically allows multiple versions of an object to be simultaneously active. The introduction of version mechanisms can inform designers of: the presence of other designers who are currently investigating a version graph, the creation and availability of new versions, and changes to the content of current versions.

In an *uncontrolled* site where multiple users have access to edit and contribute, the potential for conflict and problems arises — more so when these users work from different offices at different time. You may spend the day improving an html file for a customer. After you have made your changes, another user who works at home or another office after hours, may upload their own newly revised version of the same file, completely overwriting your work with, perhaps, no way to get it back. And often, a customer needs to remove a recently added page or content area for legal reasons or s/he simply prefers an earlier version of her/his site.

In our former models, we have not considered the version issue. In this section, we introduce version control to manage relationships between successive

instances of artifacts, organise those instances into meaningful structures, and support navigation and other operations on those structures.

3.3.1 Changes to Design. To allow versions of entities we decided to split a project's data into two pieces: a project has a *collection* of versions of entities and also a set of *parts* that refer to versions of entities in the collection rather than directly containing them. Parts are organised into hierarchies rooted in the project, like the entities themselves in the previous models. The entities with their attributes can then be changed independently of the hierarchy of parts, which defines how they are used to form a design. This has a number of consequences, which generally give more flexibility:

1 Locking an entity in the collection does not lock any part in the design, and vice versa.

2 We decided previously that multiple concurrent locks were inappropriate for CSCW. This means that attributes of an entity in the collection can no longer be locked individually. But we no longer need to worry about the attributes of an entity being dependent. (It would be easy to change this decision, and allow multiple concurrent changes to entities followed by reconciliation, for the entities in the collection.)

3 Different parts may refer to the same version of the same entity: sharing without copying is possible.

4 Entities and parts are both given designer sets, so the people allowed to change an entity can, if required, be different from the people allowed to use it in a design.

Figure 3 shows the relationships between the objects of the last model.

3.3.2 Formal Description. The following are the RSL type definitions for the last model.

```
type
  Entity ::
    attrs : Attributes
    e_per : User_Id-set ↔ re_e_per  /* designer set of entity */
    lock : Lock ↔ re_lock,
    Attributes = Attribute_Id →m Val,
  Lock == nil | locked(user : User_Id),
  Collection = Entity_Id →m Version →m Entity,
  Part ::
    parent : Super ↔ re_parent
    count : Nat1 ↔ re_count
```

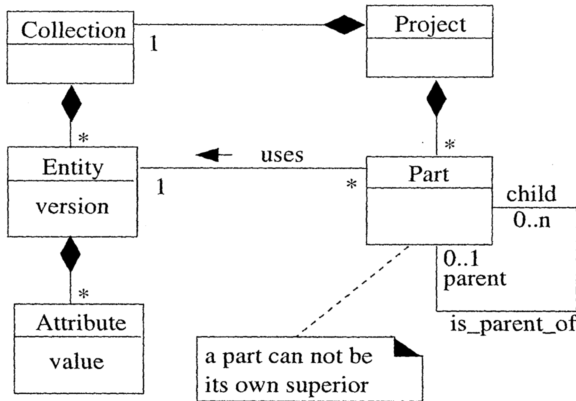

Figure 3. The relationships between the objects of the last model

$\quad$ p_per : User_Id-**set** $\leftrightarrow$ re_p_per /* designer set of part */
$\quad$ lock : Lock $\leftrightarrow$ re_lock
$\quad$ entity : Entity_Id $\leftrightarrow$ re_entity
$\quad$ version : Version $\leftrightarrow$ re_version,
Designs$'$ = Part_Id $\xrightarrow[m]{}$ Part,
Designs = {| d : Designs$'$ $\cdot$ iswf(d) |}
Project$'$ ::
$\quad$ collection : Collection $\leftrightarrow$ re_collection
$\quad$ lock : Lock $\leftrightarrow$ re_lock
$\quad$ p_per : User_Id-**set** $\leftrightarrow$ re_p_per
$\quad$ designs : Designs $\leftrightarrow$ re_designs,
Project = {| p : Project$'$ $\cdot$ iswf(p) |}
Projects$'$ = Project_Id $\xrightarrow[m]{}$ Project,
Projects = {| ps : Projects$'$ $\cdot$ iswf(ps) |}

In the definition of *part*, we use two components: *entity* and *version* to connect the part with a version of an entity.

We use *Collection* and *Designs* to store entities and parts, respectively. All of them are stored in the repository *Projects*.

The changes to the well-formedness conditions are straightforward. The previous ones remain (except for considering locks on attributes, now subsumed into locks on their entities) and we add the condition that entity versions referred to in the designs exist in the collection.

3.3.3 Operations. We add a few operations on entities, such as locking, unlocking, adding, deleting and editing. Since an entity version can be referred to by parts, we should make sure that before deleting an entity version, no part

refers to it. The first version is created when adding a new entity, while derived versions of the entity are created when editing it. We give here one example of an operation: that to lock a part in a project.

value
lock_part : Part_Id × User_Id × Project $\xrightarrow{\sim}$ Project
lock_part(pid, u, p) ≡
let
d = designs(p),
s = sub_ents(pid, d) ∪ {pid},
p1 = clear_locks_down(s, p, u),
d1 = designs(p1),
pa = re_lock(locked(u), d1(pid)),
d′ = d1 † [pid ↦ pa]
in
re_designs(d′, p)
end
pre
ispart(pid, designs(p)) ∧
can_clear_locks_down(sub_ents(pid, designs(p)) ∪ {pid}, p, u) ∧
can_lock_part(pid, p, u),

sub_ents recursively collects the inferior parts of a part. *clear_locks_down* clears any locks on a set of parts provided they are locked by the same user, checked by *can_clear_locks_down*. *can_lock_part* checks the user has permission (is in the designer set of this or a superior node) and that no superior node is locked. For the full details of this and the other operations, see (Wang Yanjie et al., 2001).

4. Prototype

We implemented a prototype by automatically translating the RAISE specification code into C++ (Univan Ahn and Chris George, 2000).

To translate we had only to instantiate the previously abstract types like the various identifiers and attribute values, for which we used typically RSL types like **Nat** and **Text**. The actual instantiation used in a real project would depend on what was being designed: html pages forming a "home" page, design drawings for a building, multimedia resources, etc. All the specification was translatable without modification: the translator deals with quantified and comprehended expressions provided they quantify or comprehend over some finite set (such as the domain of a map). Built-in types like sets, maps and products use pre-defined templates to define the RSL operators.

We also wrote a simple textual user interface to help the user to finish the initial design and modify the existing the model. The translated code preserves

the RSL names of operations, and also defines I/O operations for all types occurring in the specification, so this is straightforward.

We use the VCG (Visualisation of Compiler Graphs) (Sander, 1995) tool to give a simple visualisation of our instantiated model. In the visualisation we display both the collection of entities and the collection of parts in one project. So, it is easy for a designer to know how many entities there are and how many versions an entity has. VCG input is textual, so we specified in RSL as an extension of the specification a variable to hold the collection of projects, a function *output_project* that takes a *Project_id*, looks up its details in the variable, and generates the VCG input as a **Text**. The translator provides a function *RSL_to_cpp* that converts from the translation of **Text** to the C++ type `string`, so all the hand written part has to do to display a project `pid` is to open the VCG input file and write the result of `RSL_to_cpp(output_project(pid))` to it.

To aid in testing we extended the specification with additional versions of the operations that (a) check the relevant precondition is satisfied, (b) if so, apply the operation, and (c) check the result is well-formed. The user interface calls these versions of the operations, and reports any errors arising from (a) or (c).

An example output from the prototype is shown in figure 4.

This shows the parts in the project "Macao". User wz has locked node 5 (and hence also 9), while wyi has locked node 7. Locked part nodes are coloured red (otherwise green) as well as indicating the locking user as an arc label. The entity collection details have been hidden for clarity. Collection nodes are also red when locked (otherwise yellow) and again indicate who holds the lock.

This prototype allows our system to be evaluated in a more active way and helps users understand how our model works. It is useful for checking the deficiencies of our model and can be the basis for further study.

5. Conclusions and Future Work

Managing the consistency of distributed data is a critical issue for many collaborative systems. In a shared multi-user environment, in order to avoid conflict, increase efficiency, and ensure successful cooperation, it is important to coordinate the actions of the members. To achieve this goal, there must be a concurrency controller to protect the transactions from the disturbance between each other.

In CSCW applications, users may wish to receive immediate feedback to their local operations rather than wait until these operations have been invoked on all of the user copies of the interaction state. As a result, the local copies can become, at least temporarily, inconsistent with each other. So it is needed to

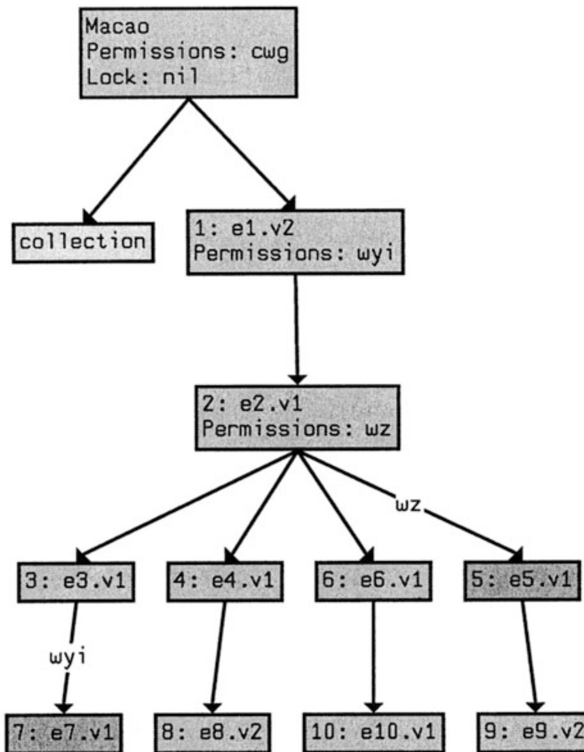

Figure 4. VCG output from the prototype

maintain user copies of the interaction state as separate logical versions rather than consistent physical replicas.

In this paper, we present a supporting multi-version concurrency control model in which we concentrate on how to use locking to achieve consistency during the collaborative design process. This model is particularly suited to the requirements of CSCW applications. We use the RAISE language to give a specification of our model. And we implement a prototype by automatically translating RSL specification into C++ code.

We think that the model we have designed and prototyped provides both the flexibility needed for collaborative design and proper version and change control. Building and using a prototype allows possible styles of interaction between designers to be explored. This might result in a new model being proposed: it could then be specified and prototyped in the same way. The concise formal specification allows critical properties to be stated clearly and abstractly, and supports rapid redevelopment.

5.1. Related Work

Some CSCW toolkits, such as Rendezvous (Ralph Hill et al., 1994) and GroupKit (Greenberg, S. and Marwood, D.,), are systems which make it easier for programmers to develop CSCW applications. They provide generic, reusable components (which may include shared data objects, or mechanisms to join and leave sessions), and behaviours (which application programmers can incorporate into their systems). In these systems, programmers can concentrate on the particular details of their own, specific applications.

Our model discusses a new technique to deal with the relationship between details of system design and details of use. This model can support a wide range of applications since the model structure can be instantiated for any design domain.

5.2. Future Work

In the future, we hope to consider the delay problem introduced by locking, and also fault tolerance for dealing with problems in the underlaying network. Designers play an active part in the cooperative design process: delays in such highly interactive activities cannot be tolerated.

References

Chen Qingzhang, Lin Jiaming, et al. (2000). Research on the New Mechanism of Concurrency Control in CSCW. Publishing House of Electronics Industry.

Chris George (2001). RAISE Tools User Guide. Technical Report 227, UNU/IIST, P.O. Box 3058, Macau.

Greenberg, S. and Marwood, D. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proceedings of ACM Conference on Computer Supported Cooperative Work 1994*, volume pp. 207-217.

RAISE Language Group, T. (1992). *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall. Available from Terma A/S. Contact jnp@terma.com.

RAISE Method Group, T. (1995). *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall. Available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book.

Ralph Hill, Tom Brinck, et al. (1994). The Rendezvous Architecture and Language for Multi-User Applications. In *ACM Transaction on Computer-Human Interaction*, volume 1(2), pp. 81-125.

Sander, G. (1995). Visualization of Compiler Graphs. I.Lemke, G.Sander and the COMPARE Consortium.

Univan Ahn and Chris George (2000). C++ Translator for RAISE Specification Language. Technical Report 220, UNU/IIST, P.O. Box 3058, Macau.

Wang Yanjie, Wang Zhuo, and Chris George (2001). Supporting Collaborative Design. Technical Report, UNU/IIST, P.O. Box 3058, Macau.

Young, R. (1996). Computer Supported Co-operative Working Course Outline. UMIST, University of Durham, Keele University.