Especificação Formal de uma Ferramenta de Trabalho Colaborativo através da Composição de Objetos Náutilus[1]

Cirney Carneiro [*]
Rodrigo Quites Reis [*, #]
Paulo Blauth Menezes [*]
{cirney, quites, blauth}@inf.ufrgs.br

[*] Universidade Federal do Rio Grande do Sul
Instituto de Informática - PPGC
Porto Alegre, Rio Grande do Sul, Brasil

[#] Universidade Federal do Pará - UFPA
Departamento de Informática
Belém, Pará, Brasil

Resumo

A construção de programas concorrentes e distribuídos que sejam corretos e confiáveis consiste em uma importante meta atual da Engenharia de Software. A linguagem baseada em objetos concorrentes Náutilus é um formalismo voltado a construção de programas desta natureza. Para ilustrar as facilidades de Náutilus, esta linguagem foi usada para especificar um editor colaborativo síncrono (CoEdit) anteriormente implementado em Java. Com a utilização de Náutilus, a especificação se mostrou mais clara, pois foi possível expressar a sincronização de processos concorrentes facilmente através das construções existentes na linguagem. Além disso, uma importante extensão ao programa CoEdit original foi realizada a partir da utilização dos mecanismos de abstração da linguagem, reutilizando os componentes especificados anteriormente e adicionando nova funcionalidade à especificação.

Palavras-chave: suporte por computador ao trabalho cooperativo, programação concorrente e distribuída, especificação formal, componentes de software, Teoria das Categorias.

Abstract

The construction of correct and reliable concurrent and distributed programs holds one of the most important issues in the Software Engineering field nowadays. The Náutilus object based language is a formalism developed to deal with this kind of programs. This language was used in the specification of a synchronous collaborative editor called CoEdit (which was originally developed with Java). CoEdit specification in Náutilus is more clear because of the facilities available in this language to express concurrent process synchronization. Also, an important extension to the original CoEdit program was implemented by the use of Náutilus' abstraction mechanisms in order to reuse previously specified components.

Keywords: *computer supported cooperative work, concurrent and distributed programming, formal specification, software components, Category Theory.*

1 Introdução

A busca de mecanismos que apóiam a construção de programas concorrentes e distribuídos vem absorvendo bastante interesse da Engenharia de Software. A crescente utilização de software com estas características determina que a qualidade destes programas seja uma meta crucial da área. Entretanto, os desenvolvedores de software enfrentam várias dificuldades durante a construção de programas concorrentes que sejam corretos e confiáveis. Tais problemas englobam características inerentes ao processo de desenvolvimento de

[1] Apoiado pela CAPES (Projeto TEIA e Bolsa de Doutorado PICDT), FAPERGS (Projeto QaP-For) e CNPq (Projetos ConcMod-VH).

software concorrente, como por exemplo, a complexidade referente a sincronização de processos, a dificuldade de expressar concorrência em paradigmas de linguagens de programação convencionais, e as dificuldades enfrentadas durante a verificação e manutenção de software concorrente.

O objetivo deste artigo é mostrar as características desejáveis para a Engenharia de Software de um novo formalismo para o desenvolvimento de sistemas concorrentes e distribuídos (com base semântica categórica) através da especificação de um editor síncrono que apóia o trabalho colaborativo de profissionais desta área. Este formalismo é Náutilus [18], uma linguagem baseada em objetos, textual (formal), de alto nível, que possui facilidades de abstração e suporta objetos distribuídos. Ela foi baseada em GNOME [27, 29], consistindo de uma parte representativa desta a qual foram adicionados mecanismos de abstração e composicionalidade. A semântica de Náutilus consiste em uma semântica basicamente operacional, categórica, que suporta objetos concorrentes e com especificação distribuída, dando suporte a sistemas reativos, comunicantes e concorrentes. Alguns aspectos de Náutilus foram primeiramente introduzidos em [17] e uma breve mas completa descrição da sua semântica foi apresentada em [19].

O artigo consiste no seguinte: a seção 2 aborda os relevantes conceitos de Náutilus, exemplificando agregação e reificação através de um objeto pilha. A seção 3 trata das características básicas da colaboração mediada por computador, apresentando um editor colaborativo chamado CoEdit (construído em Java). A seção 4 consiste na versão Náutilus do CoEdit e, reutilizando a pilha da seção 2, são mostradas as poucas alterações necessárias para adicionar a funcionalidade de desfazer a última edição (*undo*). Na seção 5 traçamos uma breve comparação das características entre as linguagens Náutilus e GNOME, Java e LOTOS. As conclusões estão na seção 6.

2 Visão Geral de Náutilus

Alguns conceitos de Náutilus são idéias novas introduzidas devido ao domínio semântico utilizado. De fato, as construções que fazem parte da linguagem são explicadas através de autômatos não seqüenciais, primeiramente introduzidos em [16]. Eles possuem a importante característica da composicionalidade diagonal e formam um domínio semântico categórico do tipo não intercalação para sistemas reativos, comunicantes e concorrentes. A semântica de Náutilus está completamente definida em [18, 19] e a linguagem é introduzida em [5, 17].

Os objetos Náutilus (código de um programa) são organizados em ações concorrentes (em geral, compostas por componentes mais simples) e dados. Uma ação em Náutilus é uma espécie de “unidade básica de execução” dentro de um objeto, sendo indivisível, finita, determinista e terminando em um tempo finito. A linguagem permite ações com alternativas que podem ser pensadas como diversas ações possuindo a mesma identificação. A escolha de qual ação ativada será executada é um não determinismo interno. Ação ativada é aquela em que todas as condições de habilitação são satisfeitas e todas as chamadas podem ser efetuadas.

As ações podem ser associadas a uma das seguintes categorias: *birth*, *death* e *request*. A primeira e a segunda determinam, respectivamente, a ativação e a desativação do objeto e uma ação da terceira categoria estando ativada só é executada quando estiver agregada ou se for chamada (*call*). Por outro lado, uma ação pode ser da categoria *birth request* quando a ação é de nascimento necessitando ser chamada e estar ativada para ocorrer; ou *death request*, sendo esta a ação de morte do objeto.

As ações exportadas (*export*) não sendo da categoria *request*, são públicas, podem ser referenciadas por outros objetos e podem ocorrer sempre que ativadas. Sendo *request*, para ocorrer necessita ser chamada também. Já as ações especificadas como *import* são ações importadas de outro objeto, sendo que nesse outro objeto essas mesmas ações são especificadas como *export*. As ações internas são privativas e podem ocorrer sempre que estiverem ativadas (ocorrência espontânea).

2.1 Concorrência e Distribuição em Náutilus

Náutilus possibilita a construção de programas concorrentes e distribuídos pois possui suporte de alto nível voltado para esse tipo de sistema. Não menos importante é o fato dessa linguagem possuir concorrência implícita, facilitando sobremaneira a tarefa do programador.

De fato, concorrência em Náutilus aparece em quatro ocasiões: numa ação através de uma atribuição múltipla; entre objetos através da agregação (quando os objetos são agregados), através da interação (*call* entre objetos) e pela unidade (composição paralela de objetos); entre ações através da reificação (refinamento) sobre um objeto base, onde transações são compostas com as ações da base de forma múltipla (cláusula *cps*); e, finalmente, entre ações dentro de um objeto através de ações que ocorrem espontaneamente.

Em princípio, não há necessidade de qualquer controle de sincronização pelo usuário (salvo quando desejado) haja vista todas essas formas de concorrência serem implícitas e naturais, garantidas pelo domínio semântico. É suficiente definir objetos e, geralmente, aplicar os construtores da linguagem sobre eles (de forma composicional) a fim de definir o comportamento de um sistema.

Assim, um sistema em Náutilus é composto, geralmente, por objetos simples concorrentes aos quais foram aplicadas, dentre outras, agregações, visões e reificações (que serão vistas nas seções seguintes). Por objeto simples entendemos que é aquele que não foi resultante diretamente de nenhuma outra construção da linguagem (ele "parte do zero"). A sua semântica é um morfismo de reificação de autômatos não seqüenciais. Com efeito, um único objeto Náutilus corresponde: dois autômatos, um origem (mais abstrato) e um destino (mais concreto e que fornece as computações); uma seta (relação semântica) entre eles [19].

2.2 Construções de Náutilus que Apoiam a Reutilização de Componentes

A reutilização de componentes em Náutilus se dá pelo uso da agregação e da visão, que aplicadas a um objeto já existente, alteram o seu relacionamento com outros objetos e a sua interface, respectivamente. De fato, na agregação temos um objeto (agregador) que fica responsável pelo relacionamento entre as ações dos demais objetos componentes da agregação. Analogia simplificada pode ser feita com a eletrônica, onde uma placa (objeto agregador) é uma unidade funcional que agrega componentes (objetos) e as portas (ações exportadas) são interligadas por trilhas (composições) conforme os sinais (entradas/saídas). Por outro lado, a visão encapsula ações, deixando na interface do objeto apenas as ações que serão efetivamente utilizadas. Mais precisamente:

- Agregação: sincronização entre dois objetos (autômatos). Ações comunicantes de uma agregação formam uma ação agregada mais complexa (e reetiquetada), deixando de serem vistas isoladamente e passando a serem parte de um objeto agregador com especificação distribuída. A agregação considera objetos que possuem portas (ações exportadas) e o

relacionamento entre essas portas; podemos dispor, nessas ações, de entradas ou saídas (ou ambas) análogas a parâmetros (informações que serão compartilhadas com outros objetos). O objeto agregador sincroniza eventualmente entradas de um objeto com saídas de outro.

- Visão: oculta uma parte de um autômato não seqüencial especificando o que se quer disponibilizar (ações a serem exportadas do objeto origem) e o restante é encapsulado. O objeto final disponibiliza somente as ações especificadas (eventualmente renomeadas) embora todas as outras ações permaneçam existindo.

2.3 Um Novo Mecanismo de Abstração: a Reificação

Náutilus possui um mecanismo de abstração poderoso chamado de reificação. A reificação consiste em especificar transações seqüenciais ou paralelas (complexas, de alto nível) usando ações de outro objeto (mais simples, mais concretas). Fazendo uma analogia com um compilador, ele implementa (traduz) uma instrução de uma linguagem, correspondendo-a a um conjunto de instruções de uma linguagem alvo. Geralmente, a instrução de mais alto nível é implementada por um conjunto de instruções de mais baixo nível.

Por conseguinte, a reificação (refinamento) visa implementar um objeto sobre as transações de outro objeto, o qual é denominado base de reificação. Está relacionada com implementação embora se assemelhe a algum tipo de chamada de procedimento. Assim, é possível especificar transações formadas por ações mais simples (de outro objeto); essa especificação pode ser de maneira seqüencial ou paralela.

2.4 Um Exemplo de Reificação e Agregação: uma Pilha em Náutilus

Uma pilha em Náutilus pode ser implementada através de uma agregação (usando os objetos Pointer e Array) e uma reificação. A figura 1 descreve a sincronização entre os objetos (na representação gráfica adotada para a linguagem [5]), onde ações são representadas por elipses, objetos por retângulos e entradas/saídas por cabeça e cauda de setas, respectivamente. Nas figuras 2 e 3, é apresentado o código-fonte correspondente.

Como reificação significa implementação, a pilha é implementada sobre o objeto agregado, realmente usando as ações da agregação. O objeto Stack é implementado sobre o objeto agregado usando as ações da agregação entre Pointer e Array. Importante mencionar que os construtores de Náutilus foram capazes de tornar mais modular a tarefa de construir (e, eventualmente, reutilizar) a pilha a partir do zero. Além disso:

- Stack é o único objeto que pode ser ativado;
- Pos é natural, logo *Sub_one* só é possível para valores de Pos maiores do que zero (item 1 na figura 2);
- As saídas Value (ações *Assign of Array* e *Consult of Array*, respectivamente 3 e 4 na figura 2) são derivadas para Pointer_Array e, depois, para Stack;
- As ações *Assign* e *Consult* do objeto Pointer_Array (1 e 2 – figura 3) ficam ativadas somente para os valores que permitem realizar o *match* dos argumentos (1<=Pos<=100) haja vista para os demais valores a ação *Position* do objeto Pointer (item 2 – figura 2) não

é integrante do objeto agregado e, portanto, não existe a possibilidade de um valor inválido para o índice de acesso a tabela.

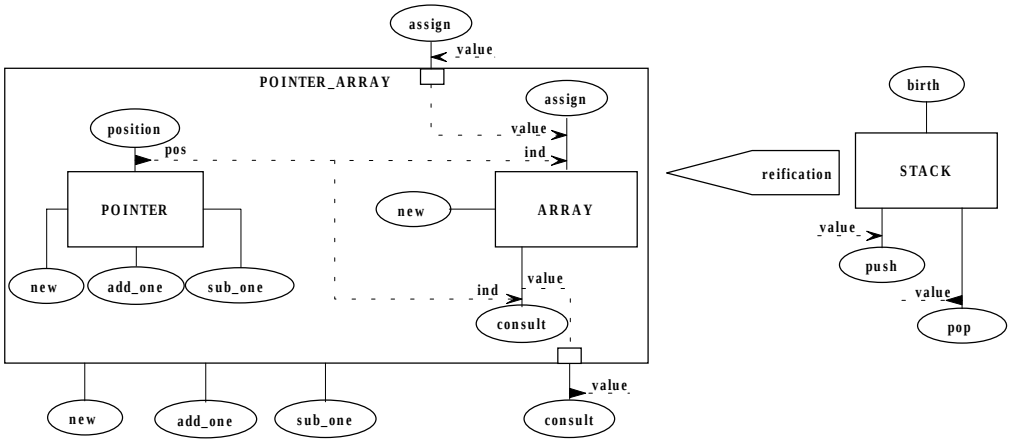

Figura 1. Esquema de sincronização para uma pilha

```
object Pointer
export
  New
  Add_one
  Sub_one
  Position out Pos: natural
category
  birth request New
  request Position
body
  slot Pos: natural
  act New
    val Pos << 0
  act Add_one
    val Pos << Pos + 1
  act Sub_one  (1)
    val Pos << Pos – 1
  act Position  (2)
    ret Position.Pos = Pos
end Pointer
```

```
object Array
export
  New
  Assign in Value: string
         in Ind: natural
  Consult out Value: string
          in Ind: natural
category
  birth request New
  request Assign Consult
body
  slot Tab: string [1..100]
  act New
  act Assign enb Assign.Ind <= 100  (3)
             enb Assign.Ind >= 1
    val Tab[Assign.Ind] << Assign.Value
  act Consult enb Assign.Ind <= 100  (4)
              enb Assign.Ind >= 1
    ret Consult.Value = Tab[Consult.Ind]
end Array
```

Figura 2. Os objetos Pointer e Array

3 Colaboração Mediada por Computador

A crescente complexidade das tarefas do mundo atual exige cada vez maior interação entre as pessoas. A disponibilização de redes de computadores de alta velocidade, cada vez mais acessíveis, aliada à evolução dos estudos sobre o comportamento dos grupos de pessoas ao desempenhar uma atividade, permitiu o surgimento de uma disciplina que se convencionou chamar Suporte por Computador ao Trabalho Cooperativo (CSCW - *Computer Supported Cooperative Work*).

Atualmente, computadores são instrumentos que além de apoiar a realização de tarefas individuais, são importantes meios de interação e cooperação entre as pessoas geograficamente dispersas [4, 23]. Neste sentido, CSCW representa o casamento de diversas questões, abordagens e linguagens, voltado ao estabelecimento de uma estrutura de apoio automatizado a atividades realizadas cooperativamente por pessoas, através do intercâmbio de informações [23].

É importante observar a confusão existente na área acerca da semântica dos termos cooperação e colaboração. Neste trabalho, é adotada a terminologia de [9], que define que uma equipe coopera se está estruturada de tal maneira que um dos participantes é o "mestre" ativo, e todos os outros participantes são seus "escravos" (adotam uma postura passiva na sessões de trabalho). O termo colaboração, ainda segundo [9], indica sessões de trabalho conjunto que são estruturadas com todos os usuários sendo ativos. Seguindo uma tradição na área, a palavra cooperação será utilizada neste texto sempre que ambos os termos (cooperação ou colaboração) forem válidos.

As seções a seguir apresentam algumas características básicas desta área.

```
Object Stack over Pointer_Array
export
  Birth
  Push der in Value: string by Assign.Value
  Pop der out Value: string by Consult.Value
category
  birth request Birth
  request Push Pop
body
  act Birth
    New
  act Push
    seq
      Add_one
      Assign
    end seq
  act Pop
    seq
      Consult
      Sub_one
    end seq
end Stack
```

```
Object Pointer_Array
aggregation of
  Pointer
  Array
export
  New
  Assign der in Value: string by Assign.Value of Array
  Consult der out Value: string by Consult.Value of Array
  Add_one     Sub_one
category
  birth New
body
  act New composed by
    New of Pointer
    New of Array
  act Assign composed by        (1)
    Position of Pointer
    Assign of Array
          match
            Position.Pos of Pointer
            Assign.Ind of Array
  act Consult composed by       (2)
     Position of Pointer
     Consult of Array
          match
            Position.Pos of Pointer
            Consult.Ind of Array
  act Add_one composed by
     Add_one of Pointer
  act Sub_one composed by
     Sub_one of Pointer
end Pointer_Array
```

Figura 3. Os objetos Stack (a reificação) e Pointer_Array (a agregação)

3.1 Conceitos Básicos

"O termo *groupware* costuma ser usado quase como sinônimo de CSCW, porém alguns autores identificam uma tendência diferenciada no emprego destes termos. Enquanto CSCW é usado para designar a pesquisa na área do trabalho em grupo e como os computadores podem apoiá-lo, *groupware* tem sido usado para designar a tecnologia (hardware e/ou software) gerada pela pesquisa em CSCW" [4]. Deste modo, segundo [8, 14], *groupware* corresponde aos sistemas baseados em computador que suportam grupos de pessoas engajadas em uma tarefa (ou objetivo) comum e que provêem interface a um ambiente compartilhado.

Um ambiente cooperativo pode ser caracterizado por prover um conjunto de objetos de dados, acessíveis por ferramentas, formando um espaço de trabalho compartilhado que permite que os usuários interajam entre si com um objetivo único (geralmente,

desenvolvimento de um produto) [23]. Portanto, o intercâmbio de informações de projeto entre as pessoas envolvidas no ciclo de vida de um produto constitui o objetivo principal das aplicações CSCW.

Duas características importantes proporcionadas pela tecnologia de CSCW, segundo [23, 24], são a **transparência de localização** e **consciência de trabalho cooperativo**, ou seja, ao mesmo tempo que os usuários possuem acesso pleno aos serviços do ambiente de forma independente da sua localização geográfica, há a consciência de que o trabalho sendo desenvolvido é resultado de um esforço coletivo. A combinação da transparência e consciência proporcionadas por CSCW produz um ambiente avançado adaptável para aplicações específicas.

Segundo Malone, citado em [3], cooperação implica em atingir objetivos comuns envolvendo atores diferentes. Deste modo, as aplicações para software cooperativo são direcionadas para efetivar as atividades de grupos de pessoas. De uma forma geral, as ferramentas CSCW estão sendo desenvolvidas para auxiliar as seguintes atividades [23]:

- *brainstorming* para geração de idéias;
- estruturação destas idéias;
- avaliação das idéias.

Consequentemente, um amplo espectro de aplicações pode ser assistido pelo uso de ferramentas CSCW. Por exemplo, ferramentas de Projeto Auxiliado por Computador (CAD), como apresentado em [22], representam um domínio de aplicações que se beneficia sobremaneira dos avanços da tecnologia de CSCW.

3.2 CoEdit em Java

CoEdit [25] é um editor colaborativo desenvolvido no Instituto de Informática da Universidade Federal do Rio Grande do Sul com o objetivo de permitir a edição síncrona de modelos de Engenharia de Software com os usuários distribuídos na Internet (abordagem "*what you see is what I see*" [7, 31]). Os usuários da ferramenta compartilham um modelo (normalmente, um diagrama) e suas operações são automaticamente percebidas pelos outros usuários. Na figura 4, é apresentada a interface deste editor em funcionamento na edição colaborativa de um Diagrama Entidade-Relacionamento compartilhado por 3 usuários (a ação de cada usuário sobre o diagrama é identificada com uma cor diferente).

Figura 4. Editor CoEdit

A primeira versão de CoEdit foi desenvolvida em Java [10, 32]. Deste modo, foram utilizados os mecanismos de programação de *threads* (para estabelecer o comportamento

concorrente do programa) e RMI - *Remote Method Invocation* (para a troca de mensagens entre objetos distribuídos na rede).

Do ponto de vista de implementação, foram construídos dois programas para fornecer esta funcionalidade do editor: o CoServer (o servidor que recebe as solicitações dos usuários) e o CoEdit (o programa cliente que interage e apresenta para o usuário o diagrama sendo compartilhado).

Java permite a construção de programas concorrentes através da utilização da classe *Thread*, disponível no pacote *java.lang* (pacote padrão da linguagem). Assim sendo, a execução de um programa Java pode se dividir em várias "linhas" de execução, havendo primitivas para criação, eliminação e comunicação entre estas *threads*. Especificamente no CoEdit (cliente), uma *thread* notificadora foi construída para verificar periodicamente se o desenho armazenado no servidor foi modificado.

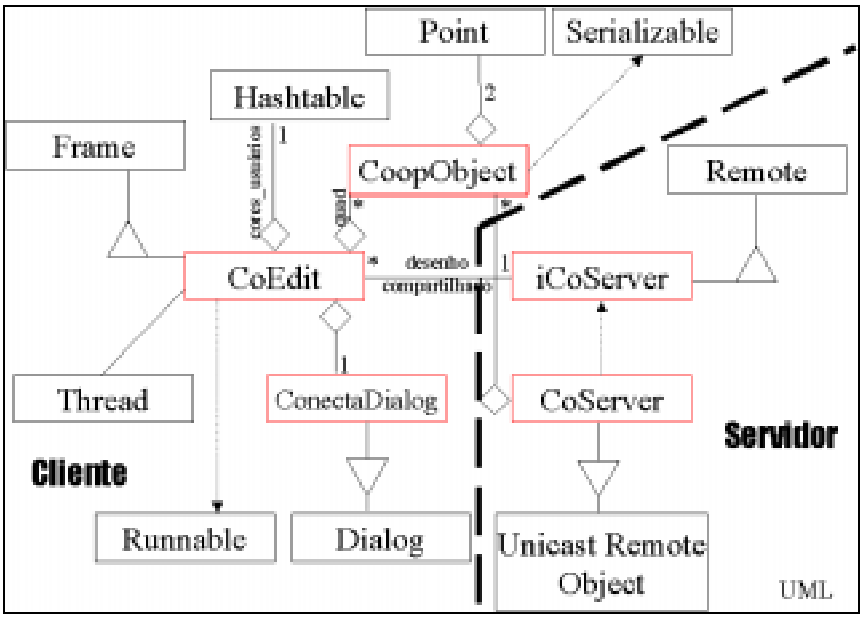

Figura 5. Diagrama de Classes para CoEdit e CoServer

O diagrama na figura 5 (na notação UML [21]) apresenta um modelo simplificado das classes construídas (em pontilhado) e reutilizadas (em sólido) no desenvolvimento desta aplicação.

4 CoEdit em Náutilus

Uma implementação do editor cooperativo CoEdit em Náutilus é mostrada aqui. O esquema de sincronização do editor para dois CoEdit está na figura 6 (apenas as ações exportadas são mostradas). A especificação de CoEdit aqui apresentada procurou respeitar as características da sua versão em Java. Assim, procurou-se criar uma especificação em Náutilus que fosse análoga ao código Java correspondente, mantendo, por exemplo, os nomes dos métodos Java para as ações Náutilus. É importante observar que, como a linguagem Náutilus é baseada em objetos, foi necessário especificar tantos objetos CoEdit quantos forem os clientes envolvidos na aplicação (no exemplo aqui especificado, foram considerados dois clientes).

Os códigos dos objetos CoEdit e CoServer em Náutilus estão na figura 7. Na figura 8, encontra-se a agregação que modela o comportamento conjunto dos objetos CoEdit e

CoServer. Nessa simplificação, só estão mostradas as ações que sincronizam entre si e isso é especificado no objeto agregador (CoEditServer) que se encarrega de relacionar as saídas/entradas das ações exportadas.

Uma agregação do objeto CoServer com objetos CoEdit modela o comportamento do editor cooperativo em Java da seção 3.2. Esta agregação foi implementada da seguinte forma:

- Objetos CoEdit: representa o cliente; pode adicionar/remover elementos gráficos (retângulos, elipses, etc.), além de possuir ações internas para receber o desenho e a versão do desenho do servidor.
  - ações *ask_add* e *ask_rem*: solicitam adicionar e remover elementos do desenho; elas fornecem ao objeto CoServer a identificação do usuário e o elemento através da saída Obj. Sincronizam com *add_obj of CoServer* e *rem_obj of CoServer*, respectivamente.
  - ação *get_cur_ver*: recebe pela entrada Ver o número da versão atual do desenho no servidor CoServer.
  - ação *repaint*: recebe o desenho atual através da entrada D para mostrá-lo na tela do cliente. Recebe o número atual da versão do desenho que está no servidor CoServer (entrada Ver). Atualiza o número da versão do desenho.
- Objeto CoServer: representa o servidor que possui ações de adicionar (remover) elementos e ações internas para enviar o desenho e sua versão para os clientes.
  - ações *add_obj* e *rem_obj*: serviços de adicionar e remover elementos do desenho editado. Sincronizam, respectivamente, com *ask_add of CoEdit* e *ask_rem of CoEdit* através da entrada Obj. Atualizam a versão do desenho.
  - ação *send_pic*: envia o desenho (saída D) e a versão atual dele (saída Ver) para os clientes.
  - ação *cur_ver*: sincroniza com *get_cur_ver of CoEdit*, fornecendo a versão atual do desenho (saída Ver).

O funcionamento do objeto CoEditServer é dado pelo comportamento conjunto dos objetos CoEdit e CoServer. Após a ativação dos objetos (ações da categoria birth), o cliente pode adicionar/remover elementos gráficos do desenho através das ações *ask_add*/*ask_rem*, cuja saída (Obj) contém a identificação do cliente e o elemento (slot *obj[1]* e *obj[2]*, respectivamente) adicionado/removido do desenho. A ação interna *get_cur_ver* recebe (de modo não determinístico) a versão do desenho e atualiza o slot *server_version*. O valor desse slot habilita ou desabilita a ação interna *repaint* de atualização do desenho e da versão do desenho que tem execução não determinista também.

Por outro lado, temos no servidor as ações *add_obj* e *rem_obj* responsáveis pela efetiva adição de elementos ao desenho. Elas atualizam o desenho (slot *server_picture*) e a versão do mesmo (*server_version*). A ação *send_pic* simplesmente retorna a versão do desenho e o próprio desenho para a ação *repaint of CoEdit*. A ação *cur_ver* retorna a versão do desenho no servidor para a ação *get_ cur_ver of CoEdit*.

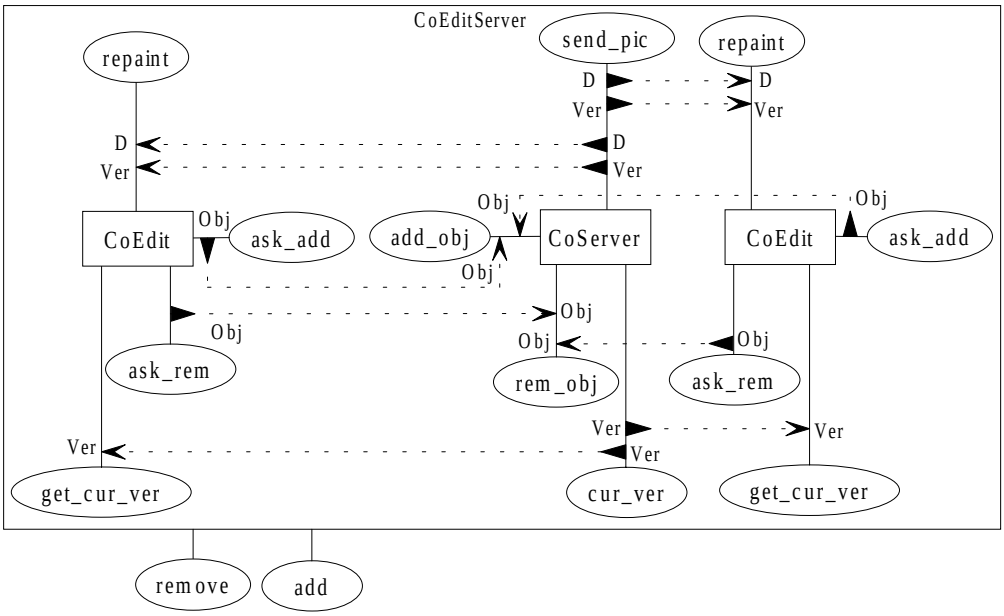

Figura 6. Esquema de sincronização do editor cooperativo

```
object CoEdit
category
  birth CoEditStart
export
  ask_add out obj: string
  ask_rem out obj: string
  repaint in Ver: natural
          in D: ...
  get_cur_ver in Ver: natural
slot
  client_version: natural
  server_version: natural
  obj: string
  client_picture: ...
body
  act CoEditStart
    cps
      val client_version << 0
      val obj[1] << ...
    end cps
  act ask_add
    val obj[2] << ...
    ret ask_add.obj = obj[2]
  act ask_rem
    val obj[2] << ...
    ret ask_rem.obj = obj[2]
  act get_cur_ver
    val server_version << get_cur_ver.Ver
  act repaint enb server_version > client_version
    cps
      val client_version << repaint.Ver
      val client_picture << repaint.D
    end cps
end CoEdit
```

```
object CoServer
category
  birth CoServerStart
export
  add_obj  in obj: string
  rem_obj  in obj: string
  send_pic out Ver: natural
           out D: ...
  cur_ver  out Ver: natural
slot
  server_version: natural
  server_picture: ...
body
  act CoServerStart
    cps
      val server_picture << ...
      val server_version << 1
    end cps
  act add_obj
    cps
      val server_picture << server_picture + add_obj[2]
      val server_version << server_version + 1
    end cps
  act rem_obj
    cps
      val server_picture << server_picture + rem_obj[2]
      val server_version << server_version + 1
    end cps
  act cur_ver
    ret cur_ver.Ver = server_version
  act send_pic
    ret send_pic.Ver = server_version
    ret send_pic.D = server_picture
end CoServer
```

Figura 7. Objetos CoServer e CoEdit em Nautilus

```
object CoEditServer                          act add composed by
agregation of                                    ask_add of CoEdit
  CoEdit                                         add_obj of CoServer
  CoServer                                           match
category                                               ask_add.obj of CoEdit
  birth CoEditServerStart                              add_obj.obj of CoEdit
export                                         act remove
  add                                            ask_rem of CoEdit
  remove                                         rem_obj of CoServer
body                                                 match
  act CoEditServerStart composed by                    ask_rem.obj of CoEdit
    CoEditStart                                        rem_obj.obj of CoEdit
    CoServerStart                            end CoEditServer
```

Figura 8. A agregação dos objetos CoServer e CoEdit em Náutilus

4.1 Reutilizando CoEdit, Stack e CoServer: Adicionando a Funcionalidade *Undo*

Pode-se estender o editor cooperativo para suportar a opção de desfazer a última operação sobre o modelo compartilhado (*undo*) através da reutilização dos componentes vistos Stack, CoEdit e CoServer. Em Java, a mesma tarefa exige grande alteração no código dos módulos. Devemos enfatizar que Náutilus, no seu estágio atual, é baseada em objetos [33,34]; por isso, o acréscimo da ação de desfazer considerou a especificação do novo objeto CoEdit'.

Foi suficiente adicionar a ação de desfazer ao objeto CoEdit (criando outro objeto – CoEdit'). Depois, agregar Stack e CoServer formando o objeto CoServerStack e especificando a ação efetiva de desfazer a última edição no servidor. A agregação entre CoServerStack e CoEdit reproduz o comportamento desejado (CoEditServerUndo). Esse esquema de sincronização pode ser visto nas figuras 9 e 10; o código Náutilus, nas figuras 11 e 12. O funcionamento do objeto agregado CoEditServerUndo é, basicamente, análogo ao do objeto CoEditServer, com a diferença que a ação de desfazer está disponível agora.

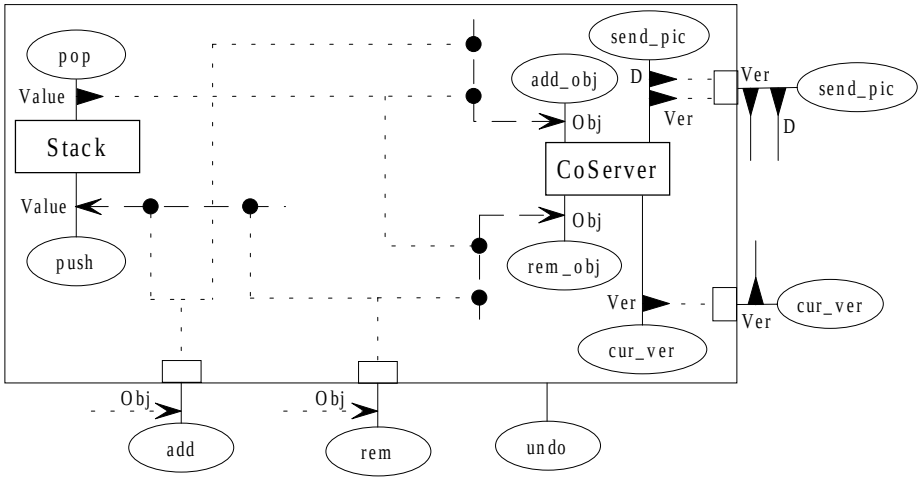

Figura 9. Esquema de sincronização do objeto CoServerStack

Observamos que no objeto agregado CoServerStack as ações *add*, *rem* e *undo* são compostas por ações dos objetos CoServer e Stack da seguinte maneira:

- ação *add*: composta pelas ações *add_obj of CoServer* e *push of Stack*. A entrada *add.Obj* é concatenada com o valor "A" antes de ser empilhada.
- ação *rem*: similar a ação *add* mas concatena um "R".

- ação *undo*: desempilhará a última edição (pela saída *pop.Value*) que possui um string com o login do cliente, o último elemento editado e “R” ou “A”. Se a última edição foi de remoção (um “R”), *undo* realizará uma adição (*add_obj of CoServer*) do elemento contido em *pop.Value*. Caso contrário, *undo* realizará uma remoção.

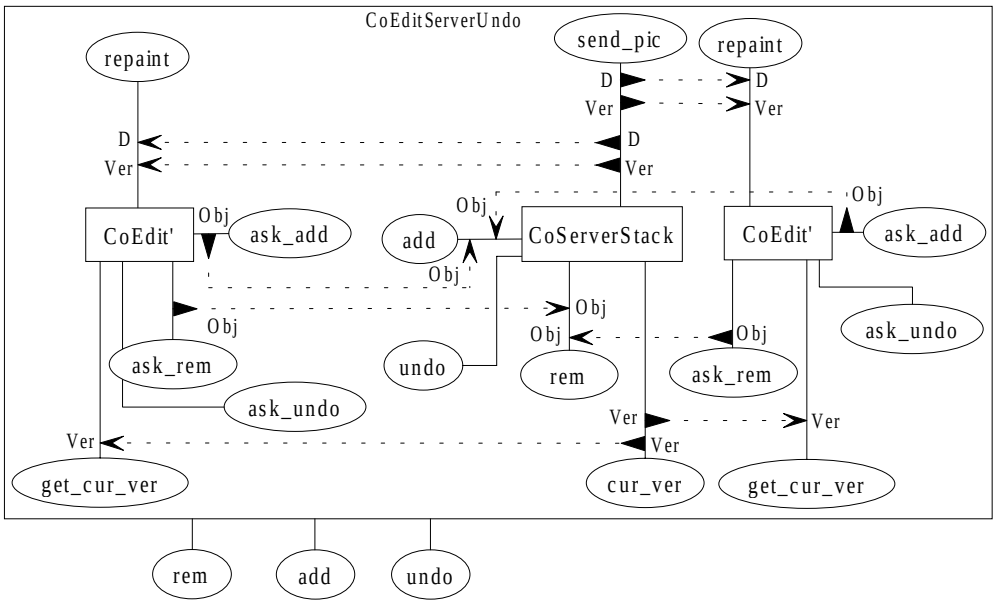

Figura 10. Esquema do editor cooperativo com a opção de desfazer edição

5 Trabalhos Relacionados

É feita uma breve comparação das características comuns entre as linguagens com base semântica categórica Náutilus e GNOME; e também entre as linguagens Java e LOTOS.

Dentre os aspectos relacionados de Náutilus e GNOME destacamos: GNOME é orientada a objetos, permite criação e destruição dinâmica de objetos, permite especificar outras classes/tipos sobre classes (“região”) e possui biblioteca parametrizada. Ambas possuem ações internas (encapsuladas) e externas, sendo que ações internas podem ocorrer sempre que ativas. Em GNOME, ações externas só podem ocorrer quando chamadas; já em Náutilus, podem ocorrer sempre que ativas. GNOME suporta interação e alternativas mas como não suporta agregação e reificação, não possui agregação e reificação dependentes de estado quando da utilização de alternativas. Náutilus possui a especificação de cláusulas de maneira seqüencial ou múltipla além de possibilitar a extração de uma visão de um objeto.

Apesar dos inúmeros benefícios da aplicação de técnicas de especificação formal no contexto de desenvolvimento de *groupware*, o seu uso ainda é limitado [25]. Por exemplo, em [26] é apresentada a especificação em LOTOS [13] de um ferramenta CSCW (o editor We-Met). Comparando-se as duas abordagens, verifica-se que Náutilus é uma linguagem mais adequada para especificação de sistemas desta natureza, por agregar características do paradigma de objetos, uma notação gráfica e diversos construtores semânticos que atuam em conjunto para a construção de especificações concisas e verificáveis. Os mecanismos de abstração propostos por Náutilus (em especial, a reificação) permitem a composicionalidade de objetos de uma maneira não usual para linguagens desta natureza.

```
object CoEdit'
category
  birth CoEdit'Start
export
  ask_add out obj: string
  ask_rem out obj: string
  repaint in Ver: natural
          in D: ...
  get_cur_ver in Ver: natural
  ask_undo
slot
  client_version: natural
  server_version: natural
  obj: string
  client_picture: ...
body
  act CoEdit'Start
    cps
      val client_version << 0
      val obj[1] << ...
    end cps
  act ask_add
    val obj[2] << ...
    ret ask_add.obj = obj
  act ask_rem
    val obj[2] << ...
    ret ask_rem.obj = obj
  act get_cur_ver
    val server_version << get_cur_ver.Ver
  act repaint enb server_version > client_version
    cps
      val client_version << repaint.Ver
      val client_picture << repaint.D
    end cps
  act ask_undo
    ...
end CoEdit
```

```
Object CoServerStack
aggregation of
  CoServer
  Stack
category
  birth CoServerStackStart
export
  send_pic der out Ver: natural by send_pic.Ver of CoServer
           der out D: ... by send_pic.D of CoServer
  cur_ver der out Ver: natural by cur_ver.Ver of CoServer
  add der in obj: string by add_obj of CoServer
  rem der in obj: string by ask_rem of CoServer
  undo
body
  act CoEditStackStart composed by
    CoServerStart of CoServer
    birth of Stack
  act send_pic composed by
    send_pic of CoServer
  act cur_ver composed by
    cur_ver of CoServer
  act add composed by
    val add.obj << add.obj + "A"
    add_obj of CoServer
    push of Stack
      match
        add_obj.obj of CoServer
        push.Value of Stack
  act rem composed by
    val rem.obj << rem.obj + "R"
    rem_obj of CoServer
    push of Stack
      match
        rem_obj.obj of CoServer
        push.Value of Stack
  act undo composed by
    pop of Stack
    alt enb pop.obj[3] = "A"
      rem_obj of CoServer
      match
        rem_obj.obj of CoServer
        pop.Value of Stack
    alt enb pop.obj[3] = "R"
      add_obj of CoServer
      match
        add_obj.obj of CoServer
        pop.Value of Stack
end CoEditStack
```

Figura 11. Objetos Coedit' e CoServerStack

```
Object CoServerUndo
aggregation of
  CoServerStack
  CoEdit'
category
  birth CoServerUndoStart
export
  add
  remove
  undo
body
  act CoServerUndoStart composed by
    CoServerStackStart
    CoEdit'Start
  act add composed by
      ask_add of CoEdit'
      add of CoServerStack
        match
          ask_add.obj of CoEdit'
          add.Value of CoServerStack
    act remove composed by
      ask_rem of CoEdit'
      rem of CoServerStack
        match
          ask_rem.obj of CoEdit'
          rem.Value of CoServerStack
    act undo composed by
      undo of CoServerStack
      undo of CoEdit'
  end CoServerUndo
```

Figura 12. Objeto CoServerUndo como uma agregação entre CoEdit' e CoServerStack

Por sua vez, Java é uma linguagem de programação aonde os aspectos de portabilidade, concorrência e distribuição influenciaram decisivamente no seu projeto. Java utiliza um paradigma de programação concorrente bastante conhecido (*threads* com primitivas de sincronização no estilo monitor [12]), sendo que este paradigma também é adotado por diversas extensões concorrentes de C++. O paradigma adotado por Náutilus simplifica a definição da sincronização de eventos. Muito código extra é necessário em um programa Java para expressar um comportamento similar ao descrito em Náutilus. Apesar de bastante conhecido, o paradigma para expressão de sincronização de eventos adotado por Java dificulta o processo de manutenção de programas concorrentes, devido a sua complexidade.

As principais deficiências da implementação atual de Náutilus consistem na ausência dos conceitos de classe e herança, e a inexistência de ações recursivas. Além disso, Náutilus é uma linguagem concorrente de propósito geral que foi aplicada na especificação do editor colaborativo CoEdit. Linguagens específicas para esta finalidade são propostas na literatura (como, por exemplo, COCA e DCWPL [15]), não sendo objetivo deste texto uma comparação com estas abordagens.

6 Conclusões e Trabalhos Futuros

Este artigo apresentou os recursos existentes na linguagem Náutilus relativos a especificação de sistemas concorrentes. Para tanto, foi especificado um editor colaborativo síncrono em Náutilus a partir de um programa escrito em Java. A especificação foi facilmente estendida, com a utilização dos construtores existentes na linguagem, para adicionar a funcionalidade *undo* na ferramenta. Em Java, a mesma tarefa exige grande alteração no código dos módulos [25].

Náutilus faz parte de uma nova geração de linguagens que tem como base semântica a Teoria das Categorias [1, 2], bem como GNOME [27, 29], OBLOG [28]. As especificações construídas com a linguagem são simples, compactas e verificáveis (embora o assunto de verificação não tenha sido abordado aqui). Os aspectos de sincronização da linguagem são similares aos adotados por LOTOS [13]. Entretanto, Náutilus possui um melhor suporte às características do paradigma de objetos.

É importante salientar que essa facilidade da linguagem é conseqüência direta dela ter sua semântica baseada num domínio categórico. A Teoria das Categorias fornece uma formalização adequada para tratar propriedades abstratas que são independentes de estruturas (independência de implementação), permitindo assim alto grau de expressividade nas suas construções, além de formalizar idéias mais complexas de forma mais simples, relacionadas com a Ciência da Computação. Outrossim, apenas parte do domínio autômato não seqüencial foi usado para fornecer semântica a Náutilus de modo que se trata de um de um domínio assaz poderoso.

Náutilus foi concebida como linguagem para especificação formal de sistemas concorrentes [18]. Porém, diversos experimentos foram conduzidos no sentido de comparar Náutilus com outras linguagens de programação (ou especificação) com primitivas que suportem o processamento concorrente/distribuído. Em [5], é apresentado um comparativo entre os aspectos de processamento concorrente em Náutilus e Java, usando um estudo de caso que foi especificado nestas duas linguagens. Constatamos que a extensão desse exemplo em Náutilus foi mais simples do que em Java. Isso motivou este presente trabalho no sentido de fazer o inverso: partir de uma especificação de um sistema em Java e chegar a uma de mesmo comportamento em Náutilus. Náutilus se comportou muito bem em ambos.

A tarefa de sincronização (objetos, ações) é simplificada em Náutilus. Por exemplo, a concorrência interna em Náutilus é implícita através da utilização da cláusula *cps*; isso em Java envolveria a criação de um número maior de *threads*, cuidando diretamente com o controle da sincronização. Não menos importante é a reutilização de componentes. De fato, vimos que uma nova característica (oriunda da agregação em Náutilus) foi obtida, principalmente, através da reutilização dos objetos Stack mais CoServer e depois do objeto CoEdit'. Além disso, a implementação em Java é mais complexa e extensa, sendo necessário bastante código para expressar o mesmo comportamento descrito em Náutilus.

A aplicação da linguagem Náutilus vem sendo feita em diversos projetos no PPGC-UFRGS para a especificação de sistemas concorrentes e distribuídos, obtendo excelentes resultados. Atualmente baseada em objetos, a sua semântica vem sendo modificada para suportar classes e herança [7]. Além disso, estão sendo construídos um protótipo da linguagem e uma interface gráfica apropriada (com suporte ao formalismo gráfico aqui utilizado). Estudos vem sendo feitos sobre o uso de Náutilus como primeira linguagem de programação no curso de Ciência da Computação da UFRGS [6].

7 Referências

[1] ASPERTI, Andrea; LONGO, Giuseppe. **Categories, Types and Structures – An Introduction to the Working Computer Science.** Foundations of Computing. M. Garey, A. Meyer - Editors. The MIT Press, 1991.

[2] BARR, Michael; WELLS, Charles. **Category Theory for Computing Science**. London: Prentice Hall International. 1990.

[3] BISCHOFBERGER, W. et al. **Computer Supported Cooperative Software Engineering with Beyond-Sniff.** (http://www.ubs.com/research/ubilab/Publications/Bis94c.html) 1995.

[4] BORGES, M. et al. **Suporte por Computador ao Trabalho Cooperativo.** Canela-RS: Instituto de Informática da UFRGS, 1995. JAI, 14., 1994, Canela.

[5] CARNEIRO, C.; REIS, R.; MENEZES, P.B. Processamento Concorrente em Náutilus e Java. Simpósio Brasileiro de Linguagens de Programação, 3. **Anais...** Porto Alegre: SBC, 1999.

[6] CARNEIRO, C. et. al. Náutilus: its Concurrent and Distributed Characteristics and as an Academic Language. In: Parallel and Distributed Processing Techniques and Applications, 1999, Las Vegas. **Proceedings...** Athens: C.S.R.E.A., 1999.

[7] CARNEIRO, C. **Autômato Não Seqüencial Identificado como Suporte para Classes em Náutilus**. Porto Alegre: UFRGS, II-PPGC, 1999. Dissertação de Mestrado.

[8] ELLIS, C.; GIBBS, S.; REIN, G. Groupware: some issues and experiences. Communications of the ACM, v.34, n.1, p.28-36, May 1994.

[9] ELLIS, C. **A Framework and Mathematical Model for Collaboration Technology.** Berlin: Springer-Verlag, 1998 (Lecture Notes in Computer Science 1364)

[10] GOSLING, J.; JOY, Bill.; STEELE, Guy. **The Java Language Specification**. (http://java.sun.com/doc/books/jls/index.html).

[11] GUTWIN, C.; GREENBERG, S. Workspace Awareness. In: CHI'97 Workshop on Awareness in Collaborative Systems, **Proceedings...** 1997.

[12] HOARE, C.A.R. Monitors – An Operating System Structuring Concept. **Communications of the ACM**, v.17, n.10. p.549-557. 1974.

[13] ISO-Information Processing Systems-Open Systems Interconection. **LOTOS - A**

Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. DIS 8807. [S.l.]. 1987.

[14] KHOSHAFIAN, S.; BUCKIEWICZ, M. **Introduction to Groupware, Workflow and Workgroup Computing.** New York: John Wiley & Sons, 1995.

[15] LI, D.; MUNTZ, R. COCA: Collaborative Objects Coordination Architecture. In: CSCW'98. ACM 1998 Conference on CSCW, **Proceedings...** p. 179-188, 1998.

[16] MENEZES, P.B.; COSTA, J.F. **Compositional Reification of Concurrent Systems**. Journal of the Brazilian Computer Society. v.2, n.1, p.50-67. Jul. 1995.

[17] MENEZES, P.B. et al. Refinement in a Concurrent Object-Based Language. In: Simpósio Brasileiro de Linguagens de Programação, 1. **Anais...** SBC: Belo Horizonte. 1996.

[18] MENEZES, P.B. **Reificação de Objetos Concorrentes**. Tese de doutorado. Instituto Superior Técnico. Universidade Técnica de Lisboa. 148 p. 1997.

[19] MENEZES, P.B. et al. Nonsequential Automata Semantics for a Concurrent Object-Based Language. In: US-Brazil Workshop on Formal Foundations of Software Systems, 1. **Proceedings…** Rio de Janeiro. Electronic Notes in Theoretical Computer Science 14. 29 p. (http://www.elsevier.nl/locate/entcs/ volume1.html). 1998.

[20] MENEZES, P.B. A Categorial Framework for Concurrent, Anticipatory Systems. In: INTERNATIONAL CONFERENCE ON COMPUTING ANTICIPATORY SYSTEMS, 2., 1998, Liege, BE. **Abstract Book**. Liege, BE: Chaos ASBL, 1998.

[21] RATIONAL (http://www.rational.com)

[22] REGLI, W.C.Internet-Enabled Computer-Aided Design. **IEEE Internet Computing**, New York, v. 1, n. 1, p. 39-50, Feb. 1997.

[23] REINHARD, W. et al. CSCW Tools: Concepts and Architectures. **IEEE Computer**, v.27, n. 5, p.28-36, May 1994.

[24] REIS, R.Q. **Uma Proposta de Suporte ao Desenvolvimento Cooperativo de Software no Ambiente PROSOFT.** Dissertação de mestrado, CPGCC-UFRGS, 1998. (http://www.inf.ufrgs.br/~prosoft)

[25] REIS, R.Q. et al. **CoEdit – Um editor colaborativo síncrono em Java**. Relatório Técnico, PPGC-UFRGS, 1998.

[26] REKERS, J.; SPRINKHUIKEN-KUYPER, I. **A LOTOS Specification of a CSCW Tool.** (ftp://ftp.wi.LeidenUniv.nl/pub/CS/TechnicalReports/1992/tr92-28.ps.gz), 1992.

[27] SERNADAS, A.; RAMOS, J. A. **Linguagem GNOME: Sintaxe, Semântica e Cálculo**. Relatório técnico. Universidade Técnica de Lisboa. 1994.

[28] SERNADAS, C.; GOUVEIA, P.; SERNADAS, A. **OBLOG: Object-Oriented Logic-Based Conceptual Modelling**. Research report. Section of Computer Science. Department of Mathematics. IST. 1096 Lisboa. Portugal. 1992.

[29] SERNADAS, C.; CARMO, P.; PENEDO, P. **The GNOME compiler**. Research Report. Section of Computer Science. Department of Mathematics. Instituto Superior Técnico. 1096 Lisboa. Portugal. 1994.

[31] STEFIK, M. et al. Beyond the Chalk Board: Computer Support for Collaboration and Problem Solving in Meetings. **Communications of the ACM**, v. 30, n. 1, 1987.

[32] SUN. **Java 1.2 Documentation**. Disponível por www em (http://java.sun.com/ products/jdk/1.2/download-docs.html). 1999.

[33] TAKAHASHI, T. et al. **Programação Orientada a Objetos.** SP: IME-USP. 1990.

[34] WEGNER, P. **Concepts and Paradigms of Object-Oriented Programming**. OOPS MESSENGER. ACM Press. vI. nI. Aug 1990.