GOD - Um Gerenciador de Objetos Distribuídos para Ambientes de Desenvolvimento de Software

Silvio R. L. Meira
Universidade Federal de Pernambuco
Centro de Ciências Exatas e da Natureza
Departamento de Informática
50.739 Recife-PE BR

Jorge H. C. Fernandes
Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas
Departamento de Informática e Matemática Aplicada
59.075 Natal-RN BR

Sumário

Este artigo apresenta GOD, o Sistema Gerenciador de Objetos Distribuídos do ambiente ForMooZ. A arquitetura do sistema é apresentada e discutida com ênfase no uso de técnicas de descrição formal como MooZ, CCS e Lógica Temporal. O trabalho compara ainda GOD com outras propostas de Gerenciadores de Objetos.

Abstract

This paper presents GOD, the Distributed Object Management System for the ForMooZ environment. The architecture of the system is presented and discussed with emphasis in the use of formal description techniques like MooZ, CCS and Temporal Logics. A comparisson is given with respect to other proposals.

1 Introdução

A complexidade e o tamanho de um sistema de software industrialmente acabado excede a capacidade intelectual humana. O desenvolvimento de software tende a ser cada vez mais uma atividade dependente da cooperação entre grupos de pessoas, as quais tratam de aspectos diferentes do sistema, seja entre fases diversas do processo como análise de requisitos, projeto, implementação e treinamento ou em diferentes aspectos da funcionalidade do software, como interface com usuário e com o hardware, gerenciamento de dados, etc.

Grande parte das ferramentas CASE atualmente fornece suporte a atividades isoladas do processo de desenvolvimento, onde o maior impedimento para sua integração é a ausência de uma representação unificada da base de dados do projeto. As duas principais propostas para criação de ambientes integrados, onde convivem sinergeticamente várias ferramentas, aptas a suportar as mais diversas fases do processo, são a do Portable Commom Tools Environment [PCT87], PCTE, e a do Commom Ada Interface Set[Obe88], CAIS.

Sejam quais forem as propostas de ambientes para desenvolvimento de software, um elemento que sempre está presente é o gerenciador de objetos, que se propõe a resolver o principal problema na convivência das várias ferramentas, que é a integração das informações por elas manipuladas.

Neste artigo apresentamos um sistema Gerenciador de Objetos Distribuídos, GOD, em uso num ambiente de desenvolvimento de software baseado em especificações formais, ForMooZ[MSC91]. Inicialmente descrevemos os objetivos e o estado do projeto ForMooZ; na Seção 3 descrevemos e comentamos a arquitetura do gerenciador, considerando os aspectos de engenharia envolvidos na construção de um ambiente de suporte ao trabalho cooperativo. A especificação formal em MooZ[MC90] do Módulo de Conversão de Tipos é base para a verificação mais aprofundada de suas características. O funcionamento de GOD baseia-se na atuação de cinco tipos de processos, que se comunicam em um ambiente de workstations em rede. Na Seção 4 fazemos uso de CCS[Mil89], da Lógica Modal μ[CPS89] e do Edinburgh Concurrency Workbench[Mol91] na modelagem e validação do protocolo

de comunicação entre seus processos. Na Seção 5, comparamos o trabalho aqui descrito com outras propostas de gerenciadores de objetos e finalmente, na Seção 6, apresentamos as nossas conclusões, indicado o estado atual do gerenciador e suas perspectivas de uso mais amplo.

2 O Projeto ForMooZ

ForMooZ é um ambiente de suporte ao desenvolvimento de sistemas baseado no uso de especificações formais e prototipação, atualmente em desenvolvimento na UFPE. A linguagem de especificação formal MooZ é a base notacional utilizada no ambiente. MooZ é uma extensão da linguagem Z[Spi89] na qual foram introduzidas modificações para torná-la orientada a objetos. A linguagem MooZ é de concepção recente e sua semântica formal está em definição. Usos significativos da linguagem são descritos em [MHFC91, MSC92] e ela está sendo utilizada no próprio desenvolvimento do ambiente.

ForMooZ dá suporte ao trabalho cooperativo na elaboração, verificação e validação de especificações formais através de editores sintáticos e compiladores e da geração semi-automática de protótipos e sua execução dentro do ambiente. A linguagem orientada a objetos Sather[Omo91], desenvolvida pelo International Computer Science Institute, em Berkeley, está sendo utilizada na sua implementação.

A interface de ForMooZ é baseada em um modelo de hipertexto similar ao de Acqua[MATS91], no qual o usuário tem acesso às informações através da visita a nós que se interligam a outros nós através de links. Estes permitem que o usuário navegue entre as definições de classes e tipos de dados de MooZ.

Um protótipo do ambiente ForMooZ está sendo validado no desenvolvimento de sua próxima versão: a máquina de hipertexto, o compilador de MooZ e GOD estão sendo utilizados nesta fase.

3 O Gerenciador de Objetos Distribuídos

A funcionalidade de GOD baseia-se prioritariamente nos seguintes requisitos:

1. Multiusuário, permitindo o trabalho simultâneo de vários usuários acessando uma base de dados comum. O sistema deve, sempre que possível, se encarregar da coordenação automática de conflitos.

2. Cooperativo, posto que à medida que o sistema a ser desenvolvido aumenta de tamanho e o número de pessoas envolvidas cresce nas mesmas proporções, multiplicam-se as possibilidades de desorganização interna pela dificuldade de coordenação das atividades. Por isto o sistema deve também promover a integração efetiva das tarefas que os usuários estão desenvolvendo, minimizando a desinformação.

3. Distribuído, pois com a crescente utilização de ForMooZ por vários usuários, além da perspectiva de integração com outras ferramentas presentes em Ambientes de Desenvolvimento de Software, a carga de processamento exigida afetaria significativamente o desempenho de um sistema centralizado.

A implementação de GOD segue o modelo cliente/servidor, no qual existem um ou mais processos clientes, que solicitam serviços a um ou mais processos servidores. Os processos clientes solicitam, aos vários processos servidores disponíveis no ambiente, o armazenamento, recuperação e controle de acesso a objetos, além do controle de classes e transações de trabalho.

Apresentamos a seguir o modelo arquitetural de GOD, que servirá para uma maior compreensão de suas características.

3.1 O Modelo Arquitetural de GOD

Além dos processos Clientes, GOD possui processos Servidores de Objeto e de Classe, Servidores de Transação e Notificadores.

A Figura 1 ilustra uma configuração padrão de processos que estão ativos no momento de funcionamento de ForMooZ.

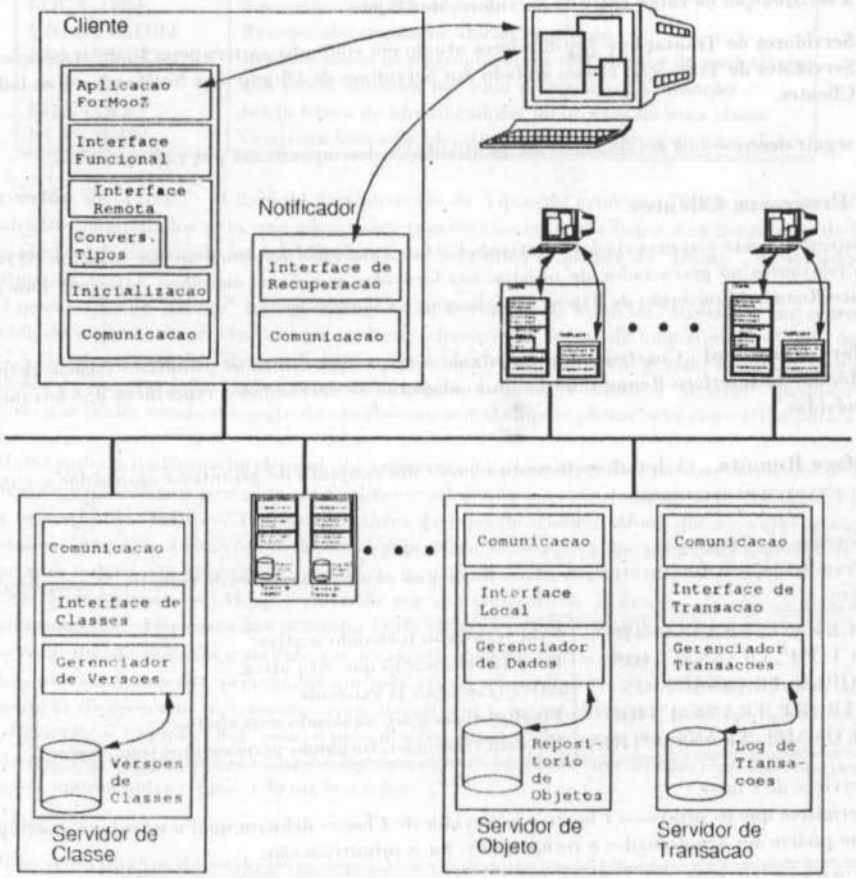

Figura 1: Lay-out dos processos de GOD, em uma configuração padrão do ambiente ForMooZ

- Clientes fornecem a principal interface externa para o uso de GOD, é a eles que a aplicação ForMooZ recorre para armazenar e recuperar objetos, definir novas classes e versões de classe, além do controle de sessões de trabalho ou transações.

- Servidores de Objeto atendem às solicitações de armazenamento e recuperação globais de objetos feitas pelos processos Clientes e coordenam o compartilhamento de objetos entre os Clientes através de bloqueios baseados em padrões, similares ao definido em [NSZ90]. Os Servidores de Objeto ainda fornecem suporte à recuperação de transações, coordenados pelos processos Servidores de Transação.

- Servidores de Classe gerenciam a criação de classes e versões de classes ou tipos. Definem ainda a distribuição de carga entre os Servidores de Objeto.

- Servidores de Transação e Notificadores atuam em conjunto na recuperação de transações. Os Servidores de Transação atuam ao lado dos Servidores de Objeto e os Notificadores ao lado dos Clientes.

A seguir descrevemos detalhadamente as atividades desempenhadas por cada processo.

3.2 Processos Clientes

No processo Cliente são executadas as atividades funcionais das ferramentas que utilizam GOD. As partes relevantes ao gerenciador de objetos são formadas por cinco módulos: Interface Funcional, Interface Remota, Conversão de Tipos, Inicialização e Comunicação.

Interface Funcional Constrói uma camada de serviços que utiliza as primitivas básicas definidas pelo Módulo de Interface Remota, de modo a adaptá-lo às necessidades específicas das ferramentas desenvolvidas.

Interface Remota. A Interface Remota oferece um conjunto de primitivas agrupadas segundo o tipo de serviço que executam.

- Serviços de Transação
  Permitem que o usuário ative, desative, finalize ou aborte transações de trabalho. Suas primitivas são:

CREATE_TRANSACTION	Cria transação tornando-a ativa
CLOSE_TRANSACTION	Desativa transação que está ativa
OPEN_TRANSACTION	Ativa transação já existente
ABORT_TRANSACTION	Finaliza transação, anulando seus efeitos
COMMIT_TRANSACTION	Finaliza transação, tornando permanentes seus efeitos

- Serviços de Classe
  Permitem que os processos Cliente e o Servidor de Classes definam qual a estrutura dos objetos que podem ser armazenados e recuperados. Suas primitivas são:

CREATE_CLASS	Cria classe
CREATE_VERSION	Cria versão de classe
FETCH_VERSION_MAP	Busca definição de estrutura de versão de classe
FIRST_CLASS	Inicia busca de informações sobre classes
NEXT_CLASS	Continua busca de informações sobre classes

- Serviços de Objeto
  Permitem a criação, recuperação, remoção, alteração, bloqueio e desbloqueio de objetos, além de

uma busca sequencial em todos os identificadores de objetos de uma determinada classe. Suas primitivas são:

CREATE_OBJ	Cria objeto
FETCH_OBJ	Busca objeto
DELETE_OBJ	Remove objeto
MODIFY_OBJ	Modifica objeto
LOCK_OBJ	Sincroniza acesso a objeto
UNLOCK_OBJ	Remove sincronizador alocado a objeto
RELEASE_LOCKS	Remove todos os sincronizadores dos objetos de uma classe que foram alocados por uma determinada transação
FIRST_KEY	Inicia busca de identificadores de objetos de uma classe
NEXT_KEY	Continua busca de identificadores de objetos de uma classe

Conversão de Tipos. O módulo de Conversão de Tipos do processo Cliente converte o formato dos objetos manipulados para que possam ser transferidos entre o Cliente e os Servidores de Objeto. A conversão é bidirecional: no sentido Cliente → Servidor é chamada de "Dump", no sentido oposto de "Restore".

O mecanismo de "Dump" baseia-se na geração de cadeias de bytes ou "streams", que representam o estado do objeto e os objetos aos quais ele se referencia na forma de uma cópia profunda ou "deep copy". O mecanismo de "Restore" utiliza o "stream" anteriormente gerado e juntamente com as definições da estrutura das classes reconstrói o objeto original. Durante o "Restore" qualquer objeto recebido que tenha versão diferente da atualmente utilizada pelo cliente será convertido para a versão local.

Muito embora tenhamos implementado a conversão de tipos específica para o compilador de Sather, o modelo é bastante genérico para ser facilmente adaptado a outras linguagens orientadas a objeto.

A operação de "Restore" é descrita através da especificação em *MooZ* que se segue, apenas com os detalhes essenciais à sua compreensão. A especificação completa dos principais aspectos de todo o gerenciador é dada em [Fer92].

Uma Especificação em MooZ é formada por um conjunto de definições de classes relacionadas hierarquicamente. As classes são definidas entre cláusulas Class *e* EndClass. *Procuramos introduzir novos conceitos da notação à medida que a especificação for apresentada.*

A conversão é descrita principalmente pela classe *ClientTypeConversorModule*. Porém antes de apresentá-la descrevemos as representações dos objetos manipulados pelos processos Clientes (classes *Obj*, *BasicObj* e *ComplexObj*); suas representações armazenáveis nos Servidores de Objeto (classes *ObjStream*, *BasicObjStream* e *ComplexObjStream*); e as entidades que armazenam informações sobre as classes manipuladas (classe *ClientClassInfo*).

Class *Obj*

Representa objetos de modo geral, segundo a visão do processo Cliente.

EndClass *Obj*.

Class *BasicObj*

Representa objetos básicos, segundo a visão do processo Cliente.

superclasses *Obj*

Objetos desta classe herdam todas as definições da classe *Obj*.

state

Na cláusula **state** definimos o estado que os objetos desta classe apresentam, através de um esquema anônimo.

$$value : BasicValue$$

O estado de um objeto básico é representado pelo componente *value*, que pode assumir qualquer valor do tipo *BasicValue*.

O "free type" *BasicValue* descreve os tipos básicos considerados nesta descrição.

$$BasicValue ::= Integer \mid Real \mid Char$$

initialstates

A cláusula **initialstates** *define esquemas que representam as mensagens de criação de objetos da classe. Neste caso específico um objeto da classe BasicObj pode ser criado através do envio da mensagem Init a esta classe.*

Init

$$value? : BasicValue$$
$$value = value?$$

EndClass *BasicObj*.

Class *ComplexObj*

Representa objetos complexos, segundo a visão do processo Cliente.

superclasses *Obj*

state

$$Local\_Class\_Id == \mathbb{N}$$

$$classId : Local\_Class\_Id$$
$$state : Seq(Obj)$$

O estado de objetos complexos é formado pelo seu identificador local de classe e por uma seqüência de outros objetos de qualquer tipo.

initialstates

Init

$$classId? : Local\_Class\_Id$$
$$classId = classId?$$
$$state = \langle\rangle$$

O esquema *Init* descreve a mensagem de criação de objetos complexos, onde apenas o identificador local da classe à qual pertence o objeto é informado.

EndClass *ComplexObj*.

Class *ObjStream*

Define a estrutura comum dos "streams" que representam o estado de objetos.

EndClass *ObjStream*.

Class *BasicObjStream*

Define a estrutura das representações armazenáveis dos objetos básicos.

superclasses *ObjStream*

state

value : *BasicValue*

O estado de um objeto desta classe é idêntico ao da classe *BasicObj*. Em uma representação mais concreta poderíamos tratar o estado como uma seqüências de bytes, mas achamos razoável considerar que esta representação abstrata serve aos propósitos de demonstrar a funcionalidade do processo Cliente.

initialstates

Init

value? : *BasicValue*
value = *value*?

EndClass *BasicObjStream*.

Class *ComplexObjStream*

Define a estrutura das representações armazenáveis dos objetos complexos.

superclasses *ObjStream*

state

version : *VERSION_ID*
state : *Seq(ObjStream)*

O estado de um objeto de tipo *ComplexObjStream* é definido pelo identificador da versão estrutural a que pertence o objeto original, e por uma seqüência de *ObjStream*, representando os objetos componentes do objeto original. Não há indicação da classe à qual pertencem.

EndClass *ComplexObjStream*.

Class *ClientClassInfo*

Esta classe descreve as informações sobre classes de objetos de GOD que são mantidas pelos processos Clientes, à exceção de seu componente extensional, isto é, seus objetos.

state

Local_Class_Id == N

Versões de classes são definidas pela seqüência das classes dos objetos que compõem os objetos desta versão de classe. A abreviação *Version_Struct* define a estrutura de um objeto deste tipo.

Version_Struct == *Seq(Global_Class_Id)*

localClassId : *Local_Class_Id*
globalClassId : *Global_Class_Id*
localVersionId : *VERSION_ID*
versionsStruct : *VERSION_ID* ⇸ *Version_Struct*
localVersionId ∈ *versionsStruct rng*

O componente *localClassId* denota o identificador interno, utilizado pelo processo cliente, para se referir à classe.

Para obter persistência dos objetos entre diferentes versões de processos é necessário estabelecer um mecanismo de nomeação, identificando cada classe de modo global, único e conciso. O componente *globalClassId* denota esta identificação global.

O componente *localVersionId* denota a identificação da versão de classe correntemente utilizada pelo processo Cliente.

O componente *versionsStruct* denota uma função que relaciona identificadores de versão (*VERSION_ID*) a definições da estrutura da versão (*Version_Struct*).

operations

A operação *localVersionStruct* resulta na definição da versão estrutural localmente utilizada pelo processo cliente.

localVersionStruct : *Version_Struct*
localVersionStruct = *versionsStruct* (*localVersionId*)

EndClass *ClientClassInfo*.

Class *ClientTypeConversorModule*

Descreve as operações de conversão de tipos e geração de representações armazenáveis entre o processo Cliente e os processos Servidores de Objeto.

superclasses *ClientInitializationModule*

Omitiremos a descrição da superclasse *ClientInitializationModule* por questão de espaço. Nos limitamos a indicar que esta classe é responsável pelas funções do Módulo de Inicialização do Cliente.

constants

basicClasses : P *Global_Class_Id*

Do ponto de vista do Módulo Conversor existem dois tipos de classes: complexas, cujos objetos são formados por seqüências de outros objetos, e básicas, formadas a partir de objetos primitivos da linguagem. O conjunto de classes básicas é definido pela constante *basicClasses*, enquanto que todas as outras classes são consideradas complexas.

state

classes : P *ClientClassInfo*

O estado dos objetos desta classe é formado pelo componente *classes*, que representa o conjunto de informações sobre classes que são conhecidas pelo processo Cliente.

operations

A operação semântica "Restore" define a criação de um objeto do tipo *ComplexObj* a partir de um objeto de tipo *ComplexObjStream*. O estado do objeto é criado a partir de sua representação armazenável, conforme foi originalmente definida em um processo Cliente qualquer. Para compatibilizar o tratamento de objetos com diferentes versões estruturais, todo "stream" de objeto que tenha versão diferente da versão localmente utilizada é necessariamente convertido para a versão local, onde todos os componentes do objeto recuperado cuja classe não é compatível com a classe local são eliminados quando da criação do objeto no processo Cliente.

$$
\begin{array}{l}
Restore: \quad (classes: \mathbb{P}\ ClientClassInfo) \times ComplexObjStream \times Global\_Class\_Id \leftrightarrow ComplexObj \\
\hline
\forall\ stream : ComplexObjStream; cId : Global\_Class\_Id \bullet \\
\quad (cId \notin basicClasses \wedge \\
\quad stream\ version \in classes\ (cId)\ versionsStruct\ dom) \Rightarrow \\
\qquad \exists\ class : ClientClassInfo \mid class \in classes \bullet \\
\qquad\quad class\ globalClassId = cId \Rightarrow \\
\qquad\qquad \exists\ obj : ComplexObj \bullet \\
\qquad\qquad\quad obj = ComplexObj\ Init(cId/classId?) \wedge \\
\qquad\qquad\quad Restore(classes, stream, cId) = RestoreComponents(classes, class, obj, stream, 1)
\end{array}
$$

Um detalhe sintático importante a ressaltar em MooZ é que na definição de operações semânticas os componentes do estado do objeto que são manipulados ou "alterados" devem ser declarados na "signature" da definição. No caso da operação Restore o componente do estado classes é manipulado pela operação.

A operação *RestoreComponents* cria os componentes do estado de um objeto complexo, conforme sua ordem crescente de declaração na classe. O componente não será criado se a classe à que pertence não for equivalente à classe descrita na estrutura de sua versão local. O resultado da operação é o objeto inicialmente informado com seus componentes criados.

$$
\begin{array}{l}
RestoreComponents: \quad (classes: \mathbb{P}\ ClientClassInfo) \times ClientClassInfo \times ComplexObj \times \\
\qquad\qquad ComplexObjStream \times Int \rightarrow ComplexObj \\
\hline
\forall\ class: ClientClassInfo; obj: ComplexObj; stream: ComplexObjStream; featNum: Int \bullet \\
\quad \exists\ localVStruct : Version\_Struct \bullet \\
\qquad localVStruct = class\ localVersionStruct \wedge \\
\qquad (featNum > localVStruct\# \Rightarrow \\
\qquad\quad RestoreComponents(class, obj, stream, featNum) = obj) \wedge \\
\qquad (featNum \leq localVStruct\# \Rightarrow \\
\qquad\quad \exists\ obj' : ComplexObj \bullet \\
\qquad\qquad obj'\ cId = obj\ cId \wedge \\
\qquad\qquad \exists\ streamVStruct : Version\_Struct \bullet \\
\qquad\qquad\quad streamVStruct = class\ versionsStruct\ (stream\ version) \wedge \\
\qquad\qquad\quad (localVStruct(featNum) = streamVStruct(featNum) \Rightarrow \\
\qquad\qquad\qquad (localVStruct\ (featNum) \notin basicClasses \Rightarrow \\
\qquad\qquad\qquad\quad obj'\ state = obj\ state \dagger \\
\qquad\qquad\qquad\qquad \{featNum \mapsto \\
\qquad\qquad\qquad\qquad\quad (Restore\ (classes, stream\ state\ (featNum), localVStruct(featNum)))\}) \wedge \\
\qquad\qquad\qquad (localVStruct\ (featNum) \in basicClasses \Rightarrow \\
\qquad\qquad\qquad\quad obj'\ state = obj\ state \dagger \\
\qquad\qquad\qquad\qquad \{featNum \mapsto (BasicObj\ Init(stream\ state\ (featnum)/value?))\})) \wedge \\
\qquad\qquad\quad (localVStruct(featNum) \neq streamVStruct(featNum) \Rightarrow \\
\qquad\qquad\qquad obj'\ state = obj\ state \dagger \\
\qquad\qquad\qquad\quad \{featNum \mapsto (ComplexObj\ Init(featCId/classId?))\}) \wedge \\
\qquad\qquad RestoreComponents(classes, class, obj, stream, featNum) = \\
\qquad\qquad\quad RestoreComponents(classes, class, obj', stream, featnum + 1))
\end{array}
$$

Inicialização Nesta versão do ForMooZ os serviços de Classe não precisaram de tratamento funcional. Toda a negociação sobre as classes e versões a serem utilizadas foi embutida neste módulo.

Comunicação. Estes suportam a troca de mensagens entre os vários processos presentes no ambiente.

3.3 Processos Servidores de Objeto

Os Servidores de Objeto implementam os Serviços de Objeto, descritos em 3.2, são formados por três módulos: Comunicação, Interface Local e Gerenciador de Dados. Descrevemos abaixo as funções do módulos Interface Local e Gerenciador de Dados.

Interface Local. À exceção da recuperação de falhas em transações, a Interface Local está apta a ser utilizada diretamente por aplicações que não necessitem compartilhar dados com outros usuários. A comunicação entre processos é desprezada e o Servidor de Objeto pode ser utilizado no modo monousuário.

A Interface Local coordena as solicitações de serviço, utilizando-se basicamente dos serviços disponíveis no Gerenciador de Dados.

Gerenciador de Dados. Implementa o acesso aos objetos em GOD, onde cada objeto armazenado está associado a uma classe.

Em ambientes multiusuários se torna imprescindível a adoção de mecanismos de sincronização de atividades entre as várias transações ativas. Em SGBDs convencionais esta sincronização é normalmente implementada através "locks" sobre elementos de dados. A sincronização de operações em GOD é feita através de bloqueios baseados em padrões, similares aos definidos em [NSZ90]. Ao solicitar o bloqueio de um objeto armazenado as aplicações cliente enviam sincronizadores, definidos por uma máquina de estados finitos estendida, na qual existe um conjunto de estados válidos que podem ser alcançados pela travessia de arcos dirigidos. Para cada arco estão definidos o tipo de operação primitiva (FETCH_OBJ, DELETE_OBJ, MODIFY_OBJ) e o identificador da transação que pode atravessá-lo. Cada operação efetuada sobre um objeto implica na mudança do estado do sincronizador. A solicitação de pares (operação,id_transação) que não estão definidos em algum arco que sai do estado atual do sincronizador é recusada pelo servidor. A Figura 2 exemplifica o uso de sincronizadores através de uma sequência de operações executadas pelo Gerenciador de Dados.

1. A transação 1 solicita o bloqueio do objeto com o sincronizador α no estado 1.
2. Se o bloqueio for aceito o Servidor de Objeto recusará qualquer operação,
3. exceto a leitura do objeto pela transação 2,
4. ou a modificação do objeto pela transação 1.
5. Após a execução da operação (MODIFY_OBJ,1) o sincronizador ficará no estado 2, quando voltará a aceitar qualquer solicitação de operação sobre o objeto. O estado 2 do sincronizador α equivale à remoção implícita do bloqueio.

O sincronizador α poderia ser utilizado por um usuário que estivesse alterando um determinado componente de software, cuja liberação para outros usuários só seria feita após as modificações serem efetivadas, com exceção do usuário da transação 2, este já ciente de que o objeto será modificado.

3.4 Processo Servidor de Classe.

Objetos só podem ser armazenados caso a versão de classe a qual pertençam já esteja definida no ambiente, o que implica no armazenamento, pelo Servidor de Classe, do mapeamento de versões que é utilizado durante a conversão de tipos efetuada nos Processos Clientes.

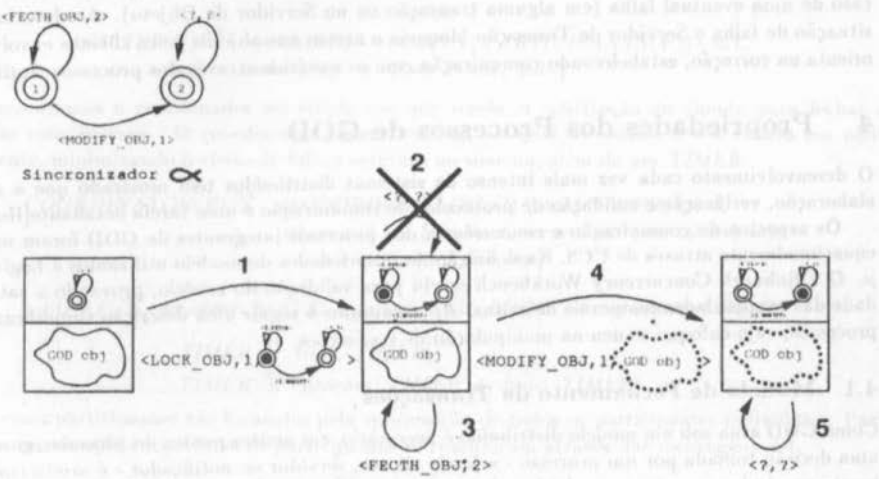

Figura 2: Exemplo do uso de sincronizadores

3.5 Processos Servidores de Transação e Notificadores

Mecanismos de gerenciamento de transações tradicionalmente implementados em SGBDs convencionais não se adaptam ao modo de trabalho cooperativo como exigido em ambientes de projeto[EGR91]. Em vista disto torna-se necessário desenvolver alternativas ao modelo de transações que podem ser radicalmente diferentes das abordagens tradicionais.

Transações em SGBDs destinados a sistemas como controle bancário ou reserva de passagens aéreas apresentam características de alto volume de transações processadas por unidade de tempo, transações curtas, atômicas, independentes e serializáveis[BHG87].

Transações em sistemas de desenvolvimento cooperativo (classificados como "groupware"), apresentam as seguintes características:

- pequeno número de transações ativas no mesmo tempo. O usuário cria transações para auxiliar na coordenação de suas atividades com outros usuários.
- transações longas e não atômicas, normalmente associadas à jornada de trabalho ou aos projetos nos quais estão envolvidos os usuários.
- transações não serializáveis. É natural que os usuários se agrupem e estabeleçam padrões de comportamento (sincronização) adaptados às idiossincrasias dos projetos aos quais estão ligados.
- recuperação não automatizada de erros. Dado que as transações em "groupware" são não-serializáveis e não atomicas, os seus efeitos não devem (a princípio) ser completamente anulados no caso de interrupção ou cancelamento de uma operação.

O gerenciamento de transações em GOD apresenta estas características, implementadas através da atuação de processos Servidores de Transação e Notificadores. A cada processo Servidor de Objeto

está associado um processo Servidor de Transação, que armazena as informações passíveis de uso no caso de uma eventual falha (em alguma transação ou no Servidor de Objeto). Ao detectar alguma situação de falha o Servidor de Transação bloqueia o acesso aos objetos pelos clientes envolvidos e os orienta na correção, estabelecendo comunicação com os usuários através dos processos Notificadores.

4 Propriedades dos Processos de GOD

O desenvolvimento cada vez mais intenso de sistemas distribuídos tem mostrado que a adequada elaboração, verificação e validação de protocolos de comunicação é uma tarefa desafiante[Hol92].

Os aspectos de comunicação e concorrência dos processos integrantes de GOD foram modelados equacionalmente através de CCS. Na definição de propriedades do modelo utilizamos a Lógica Modal μ. O Edinburgh Concurrency Workbench serviu para validação do modelo, provando a satisfatibilidade das propriedades temporais descritas. Apresentamos a seguir uma descrição simplificada destes processos, cujo enfoque se deu na manipulação de transações.

4.1 Modelo de Fechamento de Transações

Como GOD atua sob um modelo distribuído é necessário, em muitos pontos do processo, garantir que uma decisão tomada por um processo - seja ele cliente, servidor ou notificador - é aceita consensualmente por todos os outros envolvidos. Se alguém discordar da decisão todos os outros devem ficar cientes. Este tipo de negociação é bastante comum em sistemas distribuídos e se situa em uma classe de protocolos de "fechamento atômico".

Um exemplo é o tratamento de transações distribuídas, quando um cliente decide encerrar uma transação. Cada transação em GOD é coordenada por um servidor específico que inicia o seu fechamento a pedido de quem a criou. No entanto os outros Servidores de Transação, que atuam junto aos seus respectivos Servidores de Objetos, também podem influenciar no resultado do fechamento, posto que falhas podem ter ocorrido localmente a estes e ainda serem desconhecidas pelo coordenador. Tais falhas impedem que a transação sejam fechada com sucesso.

GOD utiliza o protocolo de fechamento atômico em duas fases ou "atomic commitment protocol - two-phase locking" (ACP-2PL). O algoritmo baseia-se na ação de um processo coordenador e de vários participantes.

Ao receber do cliente uma solicitação de encerramento de transação (COMMIT_TRANSACTION), o coordenador solicita a confirmação a todos os participantes. Os participantes avaliam o estado interno da transação e verificam se estão em condições de ser finalizá-la localmente com sucesso. Caso negativo o participante decide abortá-la unilateralmente e envia a decisão ao coordenador. Caso positivo o participante responde afirmativamente e aguarda a decisão final. Se nenhum participante abortou, o coordenador decide encerrar a transação com sucesso e envia a decisão a todos. Se pelo menos um participante abortou, o coordenador irrevogavelmente aborta a transação e envia a decisão final a todos os outros participantes.

O protocolo está ainda preparado para suportar falhas de comunicação entre o coordenador e os participantes. A solução é chamada de *terminação cooperativa* e consiste no questionamento entre os participantes sobre a decisão tomada pelo coordenador.

A partir do modelo algorítmico descrito em [BHG87], elaboramos um modelo equacional em CCS, no qual apenas os detalhes relevantes foram descritos.

O Sistema Completo é formado por dois tipos de agentes. Um coordenador e os participantes, que se comunicam através das mensagens $m\,I$, onde P é definido como o conjunto de nomes de todos os participantes da transação.

$$ACP\_2PI(P) \stackrel{def}{=} (COORDINATOR(P) \mid PARTICIPANTS(P)) \setminus m1,$$
$$where\ m1 = \{vote\_count,\ vote\}$$

Representamos o coordenador no estado em que recebe a solicitação do cliente para fechar a transação com sucesso. O coordenador possui um log, no qual armazena o seu estado em meio permanente, minimizando o efeito de falhas externas ao sistema, além de um *TIMER*.

$$COORDINATOR(P) \stackrel{def}{=} ((COORD(P) \mid LOG\_C) \setminus m2 \mid TIMER) \setminus m3,$$
$$where\ m2 = \{wr\_start,\ wr\_commit,\ wr\_abort\}\ and$$
$$m3 = \{time,\ timeout\}$$

$$LOG\_C \stackrel{def}{=} wr\_start.\ LOG\_C\ +\ wr\_commit.\ LOG\_C\ +\ wr\_abort.\ LOG\_C$$

$$TIMER \stackrel{def}{=} time.\ TIMER'$$
$$TIMER' \stackrel{def}{=} timeout.\ TIMER\ +\ time.\ TIMER'$$

Os processos participantes são formados pela composição de todos os participantes individuais. Para efetuar a terminação cooperativa os participantes se comunicam através das mensagens m4.

$$PARTICIPANTS(P) \stackrel{def}{=} (\prod_{p \in P} PARTIC_p(P)) \setminus m4,$$
$$where\ m4 = \{dec\_req,\ dec\_resp\}$$

Cada participante é formado por um PART, além de processos *LOG_P* e *TIMER*.

$$PARTIC_p(P) \stackrel{def}{=} ((PART_p(P) \mid TIMER) \setminus m2 \mid LOG\_P) \setminus m5,$$
$$where\ m2 = \{time,\ timeout\}\ and$$
$$m5 = \{wr\_yes,\ wr\_abort,\ wr\_commit\}$$

$$LOG\_P \stackrel{def}{=} wr\_yes.\ LOG\_P\ +\ wr\_commit.\ LOG\_P\ +\ wr\_abort.\ LOG\_P$$

O Processo Coordenador do Fechamento (*COORD*) apresenta 4 fases:
1 - Envia solicitações de votos a todos os participantes;

$$COORD(P) \stackrel{def}{=} VOTE\_REQ(P,\ \{\}),$$
$$where\ X \neq \{\},\ VOTE\_REQ(X,\ Y) \stackrel{def}{=}$$
$$\sum_{x \in X} \overline{vote\_req_x}(X \cup Y).\ VOTE\_REQ(X \setminus \{x\},\ Y \cup \{x\})$$
$$X = \{\},\ VOTE\_REQ(X,\ Y) \stackrel{def}{=}$$
$$\overline{wr\_start}.\ \overline{time}.\ VOTE\_COUNT(Y,\ \{\},\ \{\})$$

2 - aguarda as respostas;

$$TO\_COUNT \neq \{\},\quad VOTE\_COUNT(TO\_COUNT, YES, ABORT) \stackrel{def}{=}$$
$$timeout.\ ABORT(TO\_COUNT \cup YES)\ +$$
$$\sum_{x \in TO\_COUNT} vote_x(response).$$
$$if\ response = yes\ then$$
$$VOTE\_COUNT(TO\_COUNT \setminus \{x\},\ YES \cup \{x\},\ ABORT)\ else$$
$$VOTE\_COUNT(TO\_COUNT \setminus \{x\},\ YES,\ ABORT \cup \{x\})$$

$$TO\_COUNT = \{\},\quad VOTE\_COUNT(TO\_COUNT, YES, ABORT) \stackrel{def}{=}$$
$$if\ ABORT = \{\}\ then$$
$$\overline{wr\_commit}.\ COMMIT(YES)\ else$$
$$\overline{wr\_abort}.\ ABORT(YES)$$

3 - se algum participante votou *"abort"* toda a transação é abortada, os outros participantes são informados e o coordenador finaliza com a ação $\overline{failure}$.

$$ABORT(YES) \stackrel{def}{=} (\prod_{p \in YES} \overline{abort_p}.\ 0 \mid \overline{failure}.\ 0)$$

4 - se todos os participantes votaram *"yes"* a transação é encerrada com sucesso, todos os participantes são informados e o coordenador finaliza com a ação $\overline{success}$.

$$COMMIT(YES) \stackrel{def}{=} (\prod_{p \in YES} \overline{commit_p}.\ 0 \mid \overline{success}.\ 0)$$

Os processos participantes implementam o mecanismo de terminação cooperativa, permitindo que um participante não dependa obrigatoriamente do coordenador para saber qual a decisão final sobre a transação. Os participantes que já sabem da decisão devem estar preparados para responder às perguntas dos indecisos.

1 - O participante inicializa o timer, garantindo que não irá esperar eternamente pela solicitação de voto do coordenador.

$$PART_P(P) \stackrel{def}{=} \overline{timer}.\ WAIT\_VOTE\_REQ_p(P)$$

2 - Em caso de timeout o participante decide unilateralmente abortar a transação. Se receber a solicitação de voto do coordenador irá verificar se está em condições de finalizar a transação e prontamente votar. Caso ele receba a pergunta de algum outro participante sobre o resultado da transação, provavelmente houve problemaa de comunicação com o coordenador e ele decide abortar unilateralmente (provavelmente os outros, à exceção dele e do indeciso que lhe questionou, já abortaram a transação).

$$\begin{aligned} WAIT\_VOTE\_REQ_p(P) \stackrel{def}{=}\ & timeout.\ \overline{wr\_abort}.\ RESP_p(\text{abort}) + \\ & vote\_req_p.\ DECIDE_p(P) + \\ & dec\_req_p(q).\ \overline{wr\_abort}.\ \overline{dec\_resp_q(\text{abort})}.\ RESP_p(\text{abort}) \end{aligned}$$

3 - A decisão será a favor do fechamento com sucesso ou abortar a transação.

Se for *"yes"* o participante ficará no estado indeciso até receber a decisão do coordenador. Caso contrário abortará unilateralmente a transação e ficará aguardando questionamento dos outros participantes.

$$\begin{aligned} DECIDE_p(P) \stackrel{def}{=}\ & \overline{wr\_yes}.\ \overline{vote(\text{yes})}.\ \overline{time}.\ WAIT\_DECISION_p(P) + \\ & \overline{wr\_abort}.\ \overline{vote(\text{abort})}.\ RESP_p(\text{abort}) \end{aligned}$$

4 - Neste ponto o participante não pode modificar seu voto, e em caso de *"timeout"* à espera do resultado final, ele tentará a terminação cooperativa:

$$\begin{aligned} WAIT\_DECISION_p(P) \stackrel{def}{=}\ & commit_p.\ \overline{wr\_commit}.\ RESP_p(\text{commit}) + \\ & abort_p.\ \overline{wr\_abort}.\ RESP_p(\text{abort}) + \\ & timeout.\ TERM\_COOP_p(P) \end{aligned}$$

5 - O participante que já conhece a decisão final fica esperando as eventuais perguntas dos outros que estão na terminação cooperativa.

$$RESP_p(response) \stackrel{def}{=} dec\_req_p(q).\ \overline{dec\_resp_q(response)}.\ RESP_p(response)$$

6 - Na terminação cooperativa os processos questionam a decisão final a todos os outros participantes que ele conhece, até que obtenha a resposta.

$$TERM\_COOP_p(Q) \stackrel{\text{def}}{=} \sum_{q \in Q} \overline{dec\_req_q}(p).\ \dot{time}.\ (timeout.\ TERM\_COOP_p(Q) + dec\_resp_p(response).\ \textit{if}\ response = \text{commit}\ \textit{then}\ \overline{wr\_commit}.\ RESP_p(\text{commit})\ \textit{else}\ \overline{wr\_abort}.\ RESP_p(\text{abort}))$$

4.2 Prova de Propriedades

A possibilidade de prova formal de propriedades é um dos maiores benefícios no uso de técnicas de descrição formal. A partir da definição formal de propriedades podemos verificar a consistência do modelo elaborado. Algumas destas propriedades devem ser normalmente satisfeitas por qualquer protocolo de comunicação, como "liveness", "safety" e ausência de "deadlock". Outras propriedades são específicas de cada protocolo.

Descreveremos aqui as propriedades desejáveis do mecanismo de fechamento de transações de GOD utilizado a Lógica Proposicional Modal μ, que permite a expressão de várias propriedades temporais de modelos em CCS. O Concurrency Workbench possui um "Model Checker", que verifica a satisfatibilidade de proposições nesta lógica com relação a modelos de CCS.

A Lógica Modal μ é uma lógica modal estendida com operadores de ponto fixo e seu uso como uma lógica temporal para CCS é descrito em[Sti89]. Suas proposições lógicas são formadas por **identificadores**, que representam os agentes de CCS; **conectivos proposicionais** T (true), F (false), $\neg$ (negação), $\wedge$ (conjunção), $\vee$ (disjunção) e $\Rightarrow$ (implicação) e **operadores modais**. Se P é uma proposição e $a_1, ..., a_n$ são ações em CCS, então os seguintes operadores modais denotam:

- $[a_1, ..., a_n]P$ (necessidade forte);
- $[-a_1, ..., a_n]P$ (necessidade complementar forte);
- $[[a_1, ..., a_n]]P$ (necessidade fraca);
- $[[-a_1, ..., a_n]]P$ (necessidade complementar fraca);
- $\langle a_1, ..., a_n \rangle P$ (possibilidade forte);
- $\langle -a_1, ..., a_n \rangle P$ (possibilidade complementar forte);
- $\langle\langle a_1, ..., a_n \rangle\rangle P$ (possibilidade fraca);
- $\langle\langle -a_1, ..., a_n \rangle\rangle P$ (possibilidade complementar fraca);

e **operadores de ponto fixo** maximal ν e minimal μ, sob as condições de monotonicidade usuais.

4.2.1 Propriedades Gerais

Um propriedade que deveria ser satisfeita por todos os sistemas comunicantes e concorrentes é a ausência de "deadlock". Na Lógica μ esta propriedade pode ser expressa pela equação abaixo.

$$DEADLOCK\_FREE = \nu(X.\langle - \rangle T \wedge [-]X)$$

4.2.2 Propriedades Específicas

Como descrito em [BHG87], qualquer protocolo de fechamento atômico deve satisfazer um conjunto definido de propriedades. Apresentaremos aqui proposições que definem duas destas propriedades. As proposições foram verificadas com auxílio do "model checker" disponível no Concurrency Workbench e foram satisfeitas no modelo aqui descrito.

1 - **Todos os processos que chegam a uma decisão chegam à mesma decisão.**

Para todos os efeitos a decisão de um processo se dá quando da escrita, no log de transação, das ações *"wr_abort"* ou *"wr_commit"*. Para expressarmos esta propriedade necessitamos estabelecer relações de causalidade entre o registro das ações no log de todos os processos presentes no fechamento da transação. Para observarmos a ação final dos processos foi necessário introduzir sondas, convenientemente localizadas nos agentes que simulam o log de transações, modificando as equações dos agentes *LOG_P* e *LOG_C* para:

$$LOG\_C \stackrel{\text{def}}{=} wr\_start.\ LOG\_C\ +\ wr\_commit.\overline{comm}.\ LOG\_C\ +\ wr\_abort.\overline{abrt}.\ LOG\_C$$

$$LOG\_P \stackrel{\text{def}}{=} wr\_yes.\ LOG\_P\ +\ wr\_commit.\overline{comm}.\ LOG\_P\ +\ wr\_abort.\overline{abrt}.\ LOG\_P$$

A proposição que se segue expressa esta propriedade

$$P_1 \stackrel{\text{def}}{=} INV\ (\neg DISAGREE(P)),$$

$$where\ DISAGREE(P) \stackrel{\text{def}}{=} \bigvee_{p \in P} (\langle abrt_p \rangle (\bigvee_{q \in P \setminus \{p\}} \mu(X.\ \langle comm_q \rangle T\ \vee \langle - \rangle X)) \vee \quad (1)$$

$$\langle comm_p \rangle (\bigvee_{q \in P \setminus \{p\}} \mu(X.\ \langle abrt_q \rangle T\ \vee \langle - \rangle X)))$$

onde *INV* expressa a invariância de proposições e é definida como

$$INV\ prop = \nu(X.prop \wedge [-]X)$$

A propriedade foi satisfeita no modelo *ACP2PL*.

$$ACP2PL(P)\ \models\ P_1$$

2 - **A finalização com sucesso só pode ser feita se todos os participantes votaram *"yes"*.**

A dificuldade para provar esta propriedade através de CCS reside na impossibilidade de antever qual a ordem em que as ações são observadas. Com relação à ação final do coordenador, que indica se a transação foi concluída ou abortada, não há como verificar se ela será observada antes ou depois das ações de sondagem, introduzidas na prova da propriedade anterior.

A solução encontrada foi introduzir *agentes sonda* no log dos participantes, encarregados de, após a ação *"wr_yes"*, emitirem intermitentemente ações *"out_yes$_p$"*, indicando a "gravação" no log do processo p.

Modificamos então o processso *LOG_P*, que ficou definido como abaixo.

$$LOG\_P \stackrel{\text{def}}{=} wr\_yes.\ (OUT\_YES \mid LOG\_P)\ +\ wr\_commit.\ LOG\_P\ +\ wr\_abort.\ LOG\_P,$$

$$where\ OUT\_YES \stackrel{\text{def}}{=} \overline{out\_yes}.\ OUT\_YES$$

A proposição abaixo expressa a propriedade

$$P_2 \stackrel{\text{def}}{=} \mu(Z.\langle \overline{success} \rangle (\nu(X.(\bigwedge_{p \in P} \mu(Y.\langle \overline{out\_yes_p} \rangle T \vee \langle - \rangle Y)) \wedge [-]X)) \vee \langle - \rangle Z)$$

Que foi satisfeita no modelo.

$$ACP2PL(P)\ \models\ P_2$$

5 Outras Propostas de Gerenciadores de Objetos

A quase totalidade das propostas atuais de gerenciadores de objetos para Ambientes de Desenvolvimento de Software - O_2[VBD89], GemStone[BOS91], etc - baseia-se no uso de modelos orientados a objetos. No entanto propostas de comitês como PCTE e CAIS são mais conservadoras e baseiam-se no uso do Modelo Entidade Relacionamento[Che76].

Comparado aos requisitos funcionais do gerenciador de objetos do sistema O_2, GOD implementa um subconjunto das funções desempenhadas pelo primeiro. O modelo de dados de GOD suporta objetos como no O_2, no entanto o mecanismo de herança não foi tratado. Da mesma forma que em O_2, GOD suporta a evolução do ambiente de forma dinâmica, onde é possível a convivência entre usuários finais e programadores desenvolvendo novas aplicações. O modelo de distribuição da base de dados também é similar ao de O_2.

No caso de GOD é destacável a flexibilidade dos mecanismos de sincronização de atividades e recuperação de falhas.

6 Conclusões

Consideramos que o desenvolvimento de GOD foi tratado sob dois objetivos. Em primeiro lugar desejávamos abordar o desenvolvimento de modo rigoroso, validando o uso de notações e ferramentas formais. A crescente experiência no uso de MooZ tem demonstrado a expressividade da notação na descrição de sistemas não-concorrentes. A maior dificuldade no uso de CCS para modelagem de processos foi a não observação de ações sincronizadas, o que nos obrigou a construir sondas, modificando o modelo original e correndo o risco de introdução de erros. Quanto ao Workbench, as dificuldades surgiram na inabilidade para tratar diretamente o "full CCS", pelo que fomos forçados a traduzir todas as equações para o CCS puro, além da complexidade das fórmulas da Lógica Modal μ, que normalmente fazem extenso uso dos operadores de ponto fixo.

Em segundo lugar desejávamos obter um protótipo em estado operacional, a curto prazo, com características funcionais e arquiteturais bastante distintas de SGBDs convencionais. Optamos pelo desenvolvimento de um sistema distribuído, provendo serviços avançados de coordenação de trabalho cooperativo, como notificadores e sincronizadores.

O sucesso destes objetivos se deu graças ao efeito sinergético decorrente do uso de técnicas formais associadas ao extenso reuso de componentes de software como as bibliotecas de RPC e ndbm do SunOS, além de Sather, que é uma linguagem orientada a objetos em crescente uso e que já possui uma extensa biblioteca de classes.

O núcleo de GOD já está em funcionamento nos laboratórios da UFPE, onde está sendo desenvolvida a parte funcional de ForMooZ. Os processos Supervisores, Notificadores e o Servidor de Classe ainda não estão implementados e seus serviços estão sendo executados pelo Servidor de Objeto, que no momento é único. No entanto esperamos que estes componentes estejam finalizados no final do ano, junto com a distribuição da versão beta de ForMooZ.

Usos preliminares de GOD indicam que suas características são bastante úteis ao controle do trabalho cooperativo, e já existem propostas para sua utilização em outros ambientes como DisCo[FC92], em desenvolvimento na UFPE.

Agradecimentos

Agradecemos especialmente a Cássio Santos, Gustavo Motta e Ana Lúcia Cavalcanti as valiosas sugestões na definição de GOD e na preparação deste artigo.

Referências

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goldman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BOS91] P. Butterworth, A. Otis, and J. Stein. The GemStone Object Database Management System. *Communications of The ACM*, 34(10):61–77, 1991.

[Che76] P. Chen. The Entity-Relationship Model: Toward an Unified View of Data. *ACM Transactions on Database Systems*, 1(1), 1976.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. LFCS Report Series 89-83, University of Edinburg, Edinburg, England, 1989.

[EGR91] C. A. Ellis, S. G. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, 34(1):38–58, jan 1991.

[FC92] J. A. L. Filho and P. R. F. Cunha. DisCo: Um Ambiente de Programação distribuída com Configuração Dinâmica de Processos. Technical report, Universidade Federal de Pernambuco, Departamento de Informática,. Recife-PE, 1992. A ser apresentado no CLEI 92.

[Fer92] Jorge H. C. Fernandes. GOD - Um Gerenciador de Objetos para Ambientes de Desenvolvimento de Software. Master's thesis, Universidade Federal de Pernambuco, Recife - PE, 1992. Em preparação.

[Hol92] G. J. Holzmann. Protocol Design: Redefining the State of the Art. *IEEE Software*, 9(1):17–22, 1992.

[MATS91] S. R. L. Meira, E. S. Albuquerque, J. F. Tepedino, and C. S. Santos. An Experiment in Object-Oriented Design and Programming: The Acqua Hypertext System. Technical report, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1991.

[MC90] S. R. L. Meira and A. L. C. Cavalcanti. Modular Object-Oriented Z Specifications. In *Proceedings of the Fourth Annual Z users Meeting*, pages 1–19, Oxford - England, 1990.

[MHFC91] S. R. L. Meira, S. M. Holanda, J. H. C. Fernandes, and A. L. C. Cavalcanti. An object-oriented formal specification of a distributed object-oriented programming language. Technical report, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1991.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Mol91] Faron Moller. The Edinburgh Concurrency Workbench (Version 6.0). Technical report, University of Edinburg, Edinburg, England, 1991.

[MSC91] S. R. L. Meira, C. S. Santos, and A. L. C. Cavalcanti. ForMooZ: An Environment for Formal Object Oriented Specifications and Prototyping. Technical report, Universidade Federal de Pernambuco & Instituto Tecnológico de Pernambuco, 1991.

[MSC92] S. R. L. Meira, C. S. Santos, and A. L. C. Cavalcanti. The UNIX Filing System: A MooZ Specification. Technical report, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1992.

[NSZ90] M. H. Nodine, A. H. Skarra, and S. B. Zdonik. Synchronization and Recovery in Cooperative Transactions. Technical report, Brown University, Providence, RI 02912, 1990.

[Obe88] P. Oberndorf. The Common Ada Programming Support Environment (APSE) Interface Set (CAIS). *IEEE Transactions on Software Engineering*, 14(6):742–748, 1988.

[Omo91] S. Omohundro. The Sather language. Technical report, International Computer Science Institute, Berkeley, California, 1991.

[PCT87] PCTE Project Report. *ESPRIT 86: Results and Achievements*, pages 53–71. Elsevier Science Publishers, 1987.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. C. A. R. Hoare Series Editor. Prentice Hall, 1989.

[Sti89] C. Stirling. *Temporal Logics for CCS*, volume 354, pages 660–675. Springer Verlag, 1989.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: an Overview. Technical Report 27-89, Altair, Paris - France, fev 1989.