JOSIANE MICHALAK HAUAGGE

UMA PROPOSTA DE ESPECIFICAÇÃO FORMAL PARA DATA WAREHOUSING

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Mestrado em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Martin A. Musicante

CURITIBA
2000

UFPR

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática da aluna ***Josiane Michalak Hauagge***, avaliamos o trabalho intitulado "***Uma Proposta de Especificação Formal para Data Warehousing***", cuja defesa foi realizada no dia 10 de novembro de 2000. Após a avaliação, decidimos pela aprovação da Candidata.

Curitiba, 10 de novembro de 2000.

Prof. Dr. Martin Alejandro Musicante
Presidente - DINF/UFPR

Prof. Dr. Fernando da Fonseca de Souza
CIn/UFPE

Prof. Dr. Elias Procópio Duarte Jr.
DINF/UFPR

Aos meus pais,
pelo incentivo e apoio dedicados.

Agradecimentos

A Deus pelo dom da vida e porque é por Ele que todas as coisas são feitas.

Especialmente ao meu orientador, Martin A. Musicante, por tudo que me ensinou, por toda dedicação à este trabalho e principalmente por ter acreditado em mim.

Aos professores Elias Procópio Duarte Jr. e Fernando da Fonseca de Souza por todas as sugestões que enriqueceram este trabalho.

Aos meus pais, Salim e Zelia, por todo amor, paciência, ajuda e força que sempre souberam me dar.

As queridas companheiras Inali e Luciane, porque foram presença verdadeiramente amiga em todos os momentos.

Ao colega Diógenes, pela disponibilidade e atenção que sempre soube ter.

Ao colega Alessandro, por toda colaboração à este trabalho, pela participação ativa na especificação dos módulos do Projeto SAGU, principalmente pela descrição dos diferentes modelos de Monitores.

A todos os professores do curso de mestrado, por todos os conhecimentos transmitidos.

Aos professores e funcionários da UNICENTRO por todas a colaboração dispendida.

Aos meus dois irmãos, Cristiane e Fábio, pela paciência, pelo amor e por todo auxilio que prontamente me deram.

Ao Marcelo, por todo apoio, força e carinho que sempre demonstrou.

Aos amigos, especialmente a Elisa, por todo encorajamento recebido.

A todos aqueles que de uma forma ou de outra fizeram parte desta etapa.

Resumo

No desenvolvimento de Sistemas de Suporte à Decisão surge a necessidade da criação de um ambiente adequado para o armazenamento e gerenciamento eficiente dos dados produzidos pelas transações do dia-a-dia da organização, possibilitando a produção de informações que auxiliarão seus usuários na direção e alcance de seus objetivos. Para a construção de tal ambiente, se faz necessária uma reestruturação na atual arquitetura dos sistemas e uma reorganização dos dados existentes. Os *data warehouses* ou armazéns de dados surgiram desta necessidade e servem como um repositório único de informações (construído com base nos dados produzidos pelas *aplicações operacionais*) apropriado para a realização do processamento de Apoio à Decisão. Na Universidade Federal do Paraná - UFPR - está sendo desenvolvido um Sistema de Suporte à Decisão: o Projeto SAGU (Sistema de Apoio ao Gerenciamento Universitário), para atender às necessidades de informações relevantes para apoiar e agilizar o processo decisório de seus dirigentes, proporcionando o desenvolvimento acadêmico. O presente trabalho é parte integrante do projeto SAGU e seu objetivo principal é a elaboração de uma especificação formal para os componentes do sistema de *data warehouse* do projeto. Para isto, utilizamos o formalismo conhecido como semântica de ações. O propósito deste trabalho é ajudar à compreensão do processo e construir uma documentação formal do projeto. Neste trabalho descrevemos a arquitetura dos módulos de software do *data warehouse*. A especificação apresentada descreve o relacionamento entre os diferentes módulos do sistema, assim como a funcionalidade de cada um deles. O resultado do trabalho é uma documentação formal da aplicação.

Índice

Lista de Figuras

Capítulo 1

Introdução

A informação vem desempenhando um papel fundamental em todos os setores da sociedade. Cada vez mais, o sucesso das organizações torna-se dependente da utilização da informação, em particular no uso relativo à tomada de decisões [Dev97].

Porém, a maioria das aplicações informatizadas, existentes atualmente nas organizações, não são apropriadas para possibilitar o entendimento do funcionamento dos negócios como um todo, nem para realizar consultas com tempo de resposta satisfatório para atender às necessidades dos seus usuários (executivos, gerentes, analistas financeiros e de negócios), pois foram projetadas para suportar as transações que ocorrem no dia-a-dia, ou seja, realizar o processamento operacional da organização [CD97].

Surge então, a necessidade da criação de um ambiente adequado para o armazenamento e gerenciamento eficiente dos dados produzidos pelas aplicações operacionais, no qual seja possível produzir-se informações que auxiliarão seus usuários, na direção e alcance de seus objetivos.

Para a construção de tal ambiente, faz-se necessário uma reestruturação da atual arquitetura dos sistemas e uma reorganização dos dados existentes. Esta adequação pode ser obtida, através da elaboração de uma arquitetura alternativa, que consiste na separação do processamento em duas grandes categorias - o processamento operacional e o processamento informacional [Oli98], descritos a seguir.

O *processamento operacional* envolve as aplicações que foram desenvolvidas para atender

às necessidades imediatas dos usuários de departamentos específicos da empresa, ou seja, para automatizar o processamento de dados relativo às tarefas de escritório, tais como pedidos de compras, transações bancárias, cadastro de clientes. Essas operações costumam ser estruturadas e consistem de transações isoladas e atômicas, trabalhando com poucos registros que possuem dados detalhados, que precisam estar sempre atualizados e que geralmente são acessados por suas chaves primárias [CD97].

Geralmente as bases de dados dessas aplicações armazenam gigabytes de informações e a medida de sua performance é referente aos resultados de suas transações, sendo seus pontos críticos relativos à consistência e à recuperação dos dados. Conseqüentemente, as bases de dados dessas aplicações são projetadas para atender à demanda operacional e para minimizar os conflitos de concorrência.

Ao contrário das aplicações operacionais, as *aplicações informacionais* necessitam de dados históricos sobre um período de tempo mais considerável (talvez até de cinco a dez anos atrás), derivados das operações que acontecem no dia-a-dia da organização e que geralmente precisam ser sumarizados e consolidados. Assim, essas bases de dados são projetadas para armazenar uma quantidade muito maior de informações que as operacionais.

As operações realizadas nas bases de dados informacionais são basicamente consultas complexas que podem acessar milhões de registros e executar muitas pesquisas, junções e agregações de dados, sendo sua performance medida em relação aos resultados de suas consultas.

Os *data warehouses* ou armazéns de dados surgiram desta crescente necessidade de criação de um novo ambiente, que sirva como um repositório único de dados, construído com base nos dados produzidos pelas aplicações operacionais, para a realização do processamento de suporte à decisão, ou seja, o processamento informacional, aonde seus usuários possam realizar consultas que produzam dados que ajudem a entender as tendências dos negócios e fazer previsões futuras.

O desenvolvimento de sistemas de informação que usam este tipo de tecnologia representa uma forma inovadora de utilização dos dados nas organizações [IWG99].

1.1 Motivação e Objetivos

Como se trata de uma tecnologia recente, existem várias questões relacionadas a *data warehousing* que não constituem consenso e diversos tópicos ainda devem ser definidos.

Algumas arquiteturas de *data warehouse* foram definidas [Dev97, WGL96, LZW97, HAM95, ZGH95], mas essas definições não são formais. A definição formal permite uma melhor compreensão do problema pois é capaz de refletir conceitos fundamentais, tais como: a ordem de execução das operações, os escopos e associações das informações processadas, o armazenamento de valores e a comunicação entre processos.

O formalismo de Semântica de Ações [Mos92] foi escolhido porque ele foi especialmente projetado para a especificação formal de sistemas de software reais, apresentando uma grande facilidade para realizar extensões e atualizações para uma descrição em Semantica de Ações, geralmente afetando somente aquelas partes da descrição que tratam diretamente com as construções envolvidas.

Este trabalho tem como objetivo formular a especificação formal dos componentes de software do *data warehouse* do Projeto SAGU [Poz00], utilizando Semântica de Ações.

Esta especificação formal pode ser utilizada para chamar a atenção para alguns detalhes que muitas vezes são negligenciados durante o projeto de um *data warehouse*, possibilitando um melhor entendimento do processo de construção como um todo e provendo uma documentação clara e concisa deste processo.

O propósito do trabalho é ajudar à compreensão do processo e constuir uma documentação formal do projeto.

1.2 Organização da Dissertação

Esta dissertação está organizada como segue. O capítulo 2 apresenta uma introdução à Semântica de Ações. O capítulo 3, aborda a terminologia e os principais conceitos relacionados com *data warehousing*. O capítulo 4 apresenta a descrição formal dos componentes do *data warehouse*

do Projeto SAGU, usando Semântica de Ações. As conclusões do trabalho são apresentadas no capítulo 5.

Capítulo 2

Semântica de Ações

Este capítulo apresenta uma introdução aos conceitos e ao formalismo usados em Semântica de Ações e aborda as suas principais características.

A seção 2.1 apresenta o formalismo utilizado para descrever dados, produtores e ações em Semântica de Ações. A seção 2.2 descreve como as principais características da computação são abordadas, mostrando as ações definidas pelo formalismo, além de exemplos de utilização destas ações.

2.1 Formalismo

Semântica de Ações é um formalismo desenvolvido para prover descrições compreensíveis de aplicações e linguagens da vida real [Mos92, Wat91].

A Semântica de Ações foi desenvolvida a partir da *Semântica Denotacional* [Sch86], combinando formalidade com aspectos pragmáticos desejáveis, tais como:

- facilidade de leitura, pois faz uso de termos utilizados na linguagem informal;
- modularidade;
- capacidade de tratar abstrações,

- facilidade para provar propriedades algébricas para produzir semânticas equivalentes.

A Semântica de Ações utiliza entidades *ad-hoc* que são chamadas de ações e que são operacionais, ou seja, quando executadas processam as informações gradualmente [Mos92].

O formalismo de Semântica de Ações utiliza uma metalinguagem formal para descrever ações, denominada *Notação de Ações*, que apresenta ações primitivas e combinadores para formar ações complexas, mas que pode ser ampliada pela inclusão de novos tipos de dados e pela criação de abreviaturas para entidades já especificadas, sem a necessidade de reformular as descrições já realizadas.

Os símbolos usados na notação de ações são intencionalmente verbosos, tal que frases em inglês - desde que completamente formais - podem ser usadas para expressar a maioria dos conceitos presentes em computação.

As descrições em Semântica de Ações definem funções semânticas para mapear entidades sintáticas em entidades semânticas.

As *funções semânticas* são composicionais, ou seja, elas mapeam frases complexas através dos mapeamentos individuais de suas frases componentes. As funções semânticas utilizam equações semânticas para a sua especificação.

As *entidades sintáticas* representam a sintaxe concreta e abstrata dos programas de uma linguagem.

As *entidades semânticas* são a parte da especificação que define dados e ações auxiliares usados pelas funções semânticas.

A notação de ações apresenta três classes de entidades: *dados, produtores* e *ações*, utilizadas para realizar descrições. Essas classes de entidades serão apresentadas a seguir.

2.1.1 Dados

A *notação de dados* [Mos92], incluída na *notação de ações*, é usada para descrever a informação processada pelas ações, provendo uma coleção de tipos de dados abstratos definidos algebricamente, incluindo números, caracteres, valores-verdade, strings, tuplas, listas, árvores,

conjuntos, mapeamentos, além de dados que podem ser especificados *ad-hoc*.

Também são incluídos identificadores, células e agentes, que são utilizados para acessar outros dados e algumas entidades compostas por dados, tais como mensagens, contratos e abstrações.

Todos os dados da Semântica de Ações são especificados algebricamente usando *álgebras unificadas* [Mos89], um modelo algébrico não convencional onde a diferença entre *sorte*[1] e elemento é eliminada. Elementos individuais não são distinguidos de conjuntos contendo apenas aquele elemento. O sorte vazio é representado usando a constante nothing.

2.1.2 Produtores

São entidades que podem ser *avaliadas* para gerar dados. Um produtor tem um desempenho com relação ao dado produzido, que pode:

- *resultar em um item de dado*, quando o dado é gerado normalmente;
- *resultar em* nothing, quando o dado não é gerado porque uma das pré-condições necessárias para que isto ocorresse não foi satisfeita, ou quando o dado gerado não possui o tipo esperado.

Além das próprias constantes e operadores de dados poderem ser classificados como produtores, existem produtores específicos para transformar os diversos tipos de informação processada (transitória, com escopo, estável e permanente - apresentadas adiante) em dados. Por exemplo, o produtor current storage resulta no mapeamento que representa a memória no momento em que o produtor for chamado.

2.1.3 Ações

São entidades semânticas utilizadas para representar o controle e o processamento de informações. A execução de uma ação corresponde aos possíveis comportamentos dos programas,

[1] Conjunto de indivíduos que possuem propriedades em comum.

apresentando vários resultados possíveis:

- terminação normal (complete - execução completa);
- terminação excepcional (escape - saída anormal);
- terminação sem sucesso (fail - falha);
- não-terminação (diverge - divergência).

Uma ação pode interagir com outras ações recebendo informações antes de sua execução e fornecendo informações após sua execução. As informações processadas pela execução da ação podem ser classificadas de acordo com o contexto em que elas se encontram, incluindo a forma pela qual os dados são manipulados e a forma pela qual eles tendem a se propagar.

Entre os tipos de informações processadas pelas ações, temos *informações transitórias, informações com escopo, informações estáveis* e *informações permanentes.*

As *informações transitórias* representam os dados que são passados de uma ação para outra e que são disponibilizadas a uma ação para uso imediato. Este tipo de informação corresponde à avaliação de expressões em linguagens de programação.

Informações com escopo são associações de identificadores a dados com a finalidade de vincular um nome àquele dado, tal associação é denominada *binding.* Essas informações podem ser propagadas durante a execução de uma ação ou podem ser omitidas temporariamente quando uma nova associação a um identificador já existente for gerada. As informações de escopo fazem referência ao ambiente de definição de identificadores utilizados em linguagens de programação.

Informações estáveis representam dados armazenados em células de memória. Essas informações podem ser modificadas, mas não omitidas (como no caso das informações de escopo), pois persistem até serem explicitamente destruídas. Este tipo de informação corresponde aos valores atribuídos às variáveis em linguagens de programação.

Informações permanentes correspondem aos dados que são trocados entre diversas ações num ambiente de execução distribuído e não podem ser modificadas, apenas ampliadas. Representam informações correspondentes às mensagens trocadas entre processos ou entre máquinas

distintas. É definido um local de armazenamento, chamado *buffer* para sua recepção.

2.2 Facetas

O comportamento das ações é dividido em *facetas*, de acordo com os tipos de informações por elas processadas. Cada faceta possui algumas ações primitivas, combinadores para formar ações complexas, além de mecanismos para manipular e para gerar dados a partir de um tipo específico de informação.

2.2.1 Faceta Básica

As ações de faceta básica estão relacionadas especificamente com o *fluxo de controle* na execução de ações, não envolvendo o tratamento de *dados* e *produtores*. A execução de uma ação é uma sequência de passos atômicos, que devem ser ordenados para representar algum tipo de comportamento.

As ações primitivas existentes na faceta básica correspondem aos vários tipos de terminações e confirmações, abaixo apresentadas:

- complete: é uma ação indivisível que termina normalmente, executando o próximo passo da ação.
- escape: é uma ação indivisível que termina de forma excepcional.
- fail: é uma ação indivisível que termina com falha.
- diverge: é uma ação que nunca termina.
- commit: ação similar à ação complete, porém com o efeito colateral de sinalizar uma alteração nula na informação estável ou permanente processada pela ação.
- unfold: é uma ação auxiliar usada com o combinador unfolding para desviar o fluxo de controle de volta para o início deste combinador.

Os combinadores da faceta básica correspondem à sequência, intercalação e apresentação de alternativas. Eles são:

- unfolding A: a ação A será executada normalmente até o momento de executar um unfold, quando este será substituído pela própria ação A.
- indivisibly A: a ação A é desempenhada como se fosse composta apenas por um único passo.
- A_1 or A_2: introduz a idéia de escolha não-determinística e limitada de ações. Uma das ações A_1 ou A_2 é escolhida ao acaso e somente se esta ação falhar a outra será executada. No caso de uma ação commit ter sido executada, a ação alternativa fica descartada e o combinador or retorna falha no caso de falha na execução da primeira ação. Seu elemento neutro é fail.
- A_1 and A_2: corresponde à execução intercalada das ações A_1 e A_2. Seu elemento neutro é complete.
- A_1 and then A_2: corresponde à execução sequencial das ações A_1 e A_2, executando A_2 somente quando terminar a execução de A_1. Seu elemento neutro é complete.
- A_1 trap A_2: é um combinador que permite que uma ação auxiliar A_2 seja executada quando a ação A_1 escapar. Seu elemento neutro é escape.

Exemplos

- Supondo que A é uma ação arbitrária, que possivelmente processa ou altera informações, a seguinte ação pode falhar em alguns casos, mas em outros ela pode executar A, pois as duas ações são executadas em uma ordem não determinística.

```
| A
and
| fail
```

- A seguinte ação corresponde a um *loop* sem condição explícita de parada executando a subação A.

```
unfolding
| | A
| and then unfold
```

2.2.2 Faceta Funcional

Esta faceta envolve o tratamento do fluxo de *dados* e *produtores* que ocorrem em ações. A faceta funcional está relacionada com o processamento de *informações transitórias*, representando os valores intermediários processados durante a execução do programa, que podem ou não ser passados de uma ação à outra.

Itens de informações transitórias correspondem a indivíduos do sorte[2] data, definidos formalmente e representados como tuplas de dados trocadas entre ações. O sorte data abrange os sortes numéricos, truth-value, caracter, string entre outros.

Os transitórios passados à ação representam o influxo de dados. Os transitórios passados por uma ação representam os dados resultantes dos processamentos realizados com os transitórios recebidos pela ação.

As ações descritas na faceta funcional são:

- give Y: resulta em informação transitória com os dados fornecidos pelo produtor Y.
- escape with Y: escapa retornando como informação transitória os dados fornecidos pelo produtor Y; esta informação pode ser usada para identificar o motivo do escape.
- check Y: testa se o dado fornecido pelo produtor Y é igual ao sorte true, ou seja, se é verdadeiro, terminando normalmente; caso contrário, falha.
- choose Y: retorna um único elemento dos dados gerados pelo produtor Y.
- regive: propaga toda informação transitória recebida. É uma abreviatura para give the given data.

[2]Conjunto de elementos de um mesmo tipo, incluindo a forma com que esses elementos são expressos.

Os produtores descritos na faceta funcional são:

- given R: resulta em dados a partir de toda informação transitória recebida para ser avaliada, desde que esta informação seja do sorte R.
- given $R \# N$: resulta em dados a partir do N-ésimo componente da informação transitória recebida para ser avaliada, desde que esta informação seja do sorte R.
- it: produtor para uma peça de informação, sem importar o seu tipo. É uma abreviatura para the given datum.
- then: produtor para toda informação fornecida, sem importar o seu tipo. É uma abreviatura para the given data.

O combinador de ações descrito na faceta funcional é:

- A_1 then A_2: corresponde à composição funcional da informação transitória, permitindo que somente as informações transitórias geradas pela subação A_1 sejam propagadas para a subação A_2. Seu elemento neutro é regive.

Exemplos

- A seguinte ação representa um condicional que executa a subação A se a string recebida for igual a "OK" e executa a subação B se ela não for igual a "OK".

```
| check(the given string is "OK") and then A
or
| check not(the given string is "OK") and then B
```

- A seguinte ação calcula o fatorial de um número dentro de um *loop*, no qual a condição de parada é que o número inteiro recebido seja igual a zero.

Enquanto este número inteiro for maior ou igual a um, a ação devolve-o (através da ação give it), juntamente com o seu predecessor (através da ação give the predecessor of it then unfold) e volta ao início do *loop* (de acordo com o combinador unfold).

Quando a condição de parada for atingida, a ação realiza o produto de todos os números inteiros obtidos de acordo com as iterações realizadas dentro do *loop*.

```
unfolding
| | check(the given integer is less than 1) and then give 1
| or
| | check(the given integer is greater than 0)
| | and then
| | | | | give it
| | | | and
| | | | | give the predecessor of it then unfold
| | | then
| | | | give the product of(the given integer#1, the given integer#2)
```

2.2.3 Faceta Declarativa

Ações de faceta declarativa estão relacionadas com o processamento de *informações com escopo*, que geralmente são propagadas além das *informações transitórias*. As informações com escopo representam as associações (*bindings*) de identificadores (símbolos) a dados identificáveis, tais como constantes, variáveis e procedures em linguagens de programação.

As associações são representadas através do mapeamento de símbolos (*tokens*) para itens individuais de dados associáveis. Esses símbolos correspondem diretamente ao identificadores usados em programas, sendo representados por strings de caracteres.

Os dados avaliados como informação com escopo são definidos formalmente como indivíduos do sorte bindings. Os idenficadores são definidos como elementos do sorte token. Os valores que podem ser associados aos *tokens* são pertencentes ao sorte bindable. Um outro valor que pode ser assoicado a um *token* é o unknow, que indica um *binding* desconhecido.

As ações descritas na faceta declarativa são:

- bind K to Y: produz como informação com escopo o *binding* do identificador K para o dado avaliado pelo produtor Y.

- produce Y: produz *bindings* para o mapeamento de identificadores a valores do sorte bindable fornecido pelo produtor Y.

- unbind K: desfaz o *binding* relativo ao identificador K. É uma abreviatura para bind K to unknown.

- rebind: propaga todos os *bindings* recebidos para o próximo escopo. É uma abreviatura para produce the current bindings.

Os produtores descritos na faceta declarativa são:

- current bindings: resulta no mapeamento correspondente à informação com escopo do ambiente corrente.

- the R bound to K: resulta em um dado com a informação associada ao identificador K, desde que este *binding* esteja contido na informação com escopo recebida e desde que o valor associado ao identificador K seja do sorte R.

- Y_1 receiving Y_2: retorna os dados produzidos pelo produtor Y_1, tendo este recebido os *bindings* produzidos pelo produtor Y_2.

Os combinadores de ações descritos na faceta declarativa são:

- A_1 moreover A_2: a informação com escopo resultante deste combinador será composta pela união das informações com escopo produzidas pelas subações A_1 e A_2. Caso alguma associação seja gerada em ambas subações, então valerá a produzida pela subação A_2. As ações A_1 e A_2 são executadas intercaladamente. Seu elemento neutro é complete.

- A_1 hence A_2: corresponde à composição funcional da informação com escopo, permitindo que somente as informações com escopo geradas pela subação A_1 sejam propagadas para a subação A_2. As ações A_1 e A_2 são executadas sequencialmente. Seu elemento neutro é rebind.

- A_1 before A_2: a informação com escopo recebida pela ação A_1 é a recebida pelo combinador; já a informação com escopo recebida pela ação A_2 é formada pela junção da informações com escopo gerada pela ação A_1 com a recebida pelo combinador. A informação com escopo gerada pelo combinador será a junção da informação com escopo gerada pelas duas subações. As ações A_1 e A_2 são executadas sequencialmente. Seu elemento neutro é complete.

- furthermore A: é uma abreviatura para rebind moreover A.

Exemplos

- A seguinte ação cria a associação da string "Cooperative" ao identificador "Source_Type".

  bind "Source_Type" to "Cooperative"

- A seguinte ação retorna a string associada ao identificador Source_Type.

  give the string bound to "Source_Type"

- No início da ação seguinte é feita a associação do valor 1 ao identificador Count e a associação do valor 0 ao identificador Total, então todos os bindings dessas duas subações são repassados à segunda parte da ação pelo combinador hence.

  Através da ação regive them os bindings recebidos são passados à frente, para uma próxima ação ou como resultado da execução da ação. A próxima parte da ação realiza a associação do valor 100 ao identificador Total (na ação bind "Total" to 100). De acordo com o combinador moreover o resultado final da ação será uma tupla de dados, contendo a combinação das associações resultantes dessas duas últimas subações (regive them e bind "Total" to 100), e no caso de alguma associação ser gerada em ambas as ações (como no caso do identificador Total no nosso exemplo), então valerá somente a produzida pela segunda ação.

```
| | bind "Count" to 1
| and
| | bind "Total" to 0
hence
| | regive them
| moreover
| | bind "Total" to 100
```

2.2.4 Faceta Imperativa

As ações de faceta imperativa estão relacionadas com o processamento de *infomações estáveis*, que geralmente são propagadas além das *informações transitórias* e *com escopo*, permanecendo disponíveis até serem atualizadas por alguma ação primitiva.

Informações estáveis consistem de uma coleção de itens independentes e cada atualização realizada somente atinge um item. As atualizações realizadas são definitivas e somente podem ser desfeitas se uma cópia anterior à atualização estiver disponível.

As informações estáveis representam os valores atribuídos às variáveis em linguagens de programação. As implementações representam as informações estáveis utilizando *Memória de Acesso Randômico* (RAM) ou dispositivos de acesso secundário, tais como fitas magnéticas e discos.

Os dados avaliados como informações estáveis são definidos formalmente como indivíduos do sorte storage, que são mapeamentos de células para valores armazenáveis. A memória é formada por um conjunto (potencialmente infinito) de células, que são definidas como elementos do sorte cell. Em um determinado momento, cada célula de memória pode estar alocada ou não, e se estiver alocada, poderá armazenar dados. Os valores que podem ser armazenados em células são pertencentes ao sorte storable. Um outro valor que pode ser vinculado a uma célula é o uninitialized, que indica que nenhum dado está armazenado naquela célula.

As ações descritas na faceta imperativa são:

- store Y_1 in Y_2: produz como informação estável o armazenamento do valor fornecido pelo produtor Y_1 na célula fornecida pelo produtor Y_2, desde que esta célula faça parte da informação estável já reservada.

- unstore Y: remove os dados armazenados na célula fornecida pelo produtor Y, desde que esta célula faça parte da informação estável já reservada.

- reserve Y: reserva a célula fornecida pelo produtor Y para uso, desde que esta célula não faça parte da informação estável já reservada.

- unreserve Y: desaloca a célula fornecida pelo produtor Y, desde que esta célula faça parte da informação estável já reservada.

- allocate a cell: escolhe uma célula dentre as células não alocadas, faz sua reserva e a devolve como informação transitória. É uma abreviatura para:

```
allocate a cell =
    indivisibly
    | | choose a cell [not in mapped-set of the current storage]
    | then
    | | reserve the given cell and give it
```

Os produtores descritos na faceta imperativa são:

- current storage: resulta no mapeamento correspondente à informação estável recebida pela ação.

- the R stored in Y: resulta em um dado com a informação armazenada na célula de memória retornada como dado pelo produtor Y, desde que o produtor Y retorne uma célula que faça parte da informação estável já reservada e desde que o valor armazenado nesta célula seja do sorte R.

Exemplo

- A seguinte ação, primeiramente aloca uma nova célula de memória não ocupada e passa essa célula à sua segunda parte (após o then). Então a ação associa esta célula ao identificador "Source_Type" e armazena a string "Cooperative" nela. Estas duas últimas ações são executadas intercaladamente de acordo com as características do combinador and.

```
| allocate a cell
then
| | bind "Source_Type" to the given cell
| and
| | store the string "Cooperative" in it
```

2.2.5 Faceta Reflexiva

As ações de faceta reflexiva estão relacionadas com o *encapsulamento* de ações como *dados*, gerando *abstrações*.

Uma *abstração* corresponde ao código de uma ação, que pode ser implementado como uma sequência de instruções de máquina ou como um *ponteiro* para tal sequência. As abstrações são usadas para representar *procedures* em linguagens de programação. A ação encapsulada na abstração geralmente é executada em um contexto diferente daquele em que a própria abstração acontece.

A faceta reflexiva define o sorte abstraction para representar as abstrações.

A ação descrita na faceta reflexiva é:

- enact Y: executa a ação incorporada na abstração fornecida pelo produtor Y.

Os produtores descritos na faceta reflexiva são:

- abstraction of A: resulta em um dado a partir do encapsulamento da ação A.
- application A to Y: resulta em um dado a partir do encapsulamento da ação A juntamente com a informação transitória proveniente como resultado fornecido pelo produtor Y.
- closure A: resulta em um dado a partir do encapsulamento da ação A juntamente com a informação com escopo recebida.

Exemplo

- A seguinte ação executa a ação incorporada na abstração associada à string "Fact", fornecendo à ela o inteiro recebido como uma informação transitória .

  enact the application of the abstraction bound to "Fact" to the given integer

2.2.6 Faceta Comunicativa

As ações de faceta comunicativa estão relacionadas com o processamento de informação em *sistemas distribuídos de agentes*, aonde cada agente pode comunicar-se com outros agentes, enviando e recebendo *mensagens* e oferecendo *contratos*.

Um agente representa a identidade de uma entidade responsável pela execução das ações. A comunicação entre agentes é *assíncrona*: para enviar uma mensagem, um agente emite a mensagem em um meio, com outro agente especificado como destinatário e continua executando as suas ações, sem a necessidade de esperar até que a mensagem seja entregue.

Cada mensagem transmitida constitui uma informação permanente, não existindo uma forma de apagar uma comunicação, uma vez que ela tenha sido iniciada.

Cada agente tem seu próprio *buffer* e armazenamento. As células alocadas para um agente são referentes ao seu armazenamento local e são distintas daquelas alocadas em outros agentes.

As mensagens recebidas por um agente são colocadas em seu *buffer* e não interrompem a ação que o agente receptor está executando, permanecendo desconhecida até que o mesmo inspecione o *buffer*.

Toda mensagem possui a identificação do agente remetente, a identificação do agente destinatário, o conteúdo (que pode ser um dado, uma abstração ou a identificação de outro agente) e um número serial (determinado pelo agente remetente) que a distingue de todas as outras mensagens.

Inicialmente, existe um único agente especial ativo, denominado *user agent*, que é responsável pela inicialização do processo. Qualquer agente ativo pode contratar outros agentes inativos e especificar as ações que eles devem executar.

As ações descritas na faceta comunicativa são:

- send Y: transmite a mensagem fornecida pelo produtor Y para o agente especificado na mensagem.

- remove Y: remove a mensagem fornecida pelo produtor Y do *buffer* do agente que está executando a ação.

- offer Y: realiza o contrato fornecido pelo produtor Y.

- receive Y: espera pela mensagem dada pelo produtor Y até que esta seja recebida, então a mensagem é removida do *buffer* e retornada como informação transitória. É a abreviatura para:

```
receive Y : yielder =
    patiently
    | | choose a Y [in set of items of the current buffer]
    | then
    | | remove the given message and give it
```

- subordinate Y: cria um novo agente, que ficará esperando por uma mensagem contendo uma ação a ser executada. É uma abreviação para:

```
subordinate Y : yielder =
    | offer a contract[to Y][containing abstraction of subordinate-action]
    and then
    | | receive a message[form Y][containing an agent]
    | then
    | | give the contents of the given message
```

onde subordinate-action é a primeira ação que será executada pelo novo agente. Ela envia a identificação do novo agente para o agente que o criou e espera deste uma ação a ser executada a seguir.

```
subordinate-action =
    | send a message[to the contracting-agent][containing the performing-agent]
    then
    | receive a message[from the contracting-agent][containing an abstraction]
    then
    | enact the contents of the given message
```

Os produtores descritos na faceta comunicativa são:

- current buffer: resulta em dados a partir do buffer do agente que está executando a ação.
- performing-agent: resulta em um dado com a identidade do agente que está executando a ação.
- contracting-agent: resulta em um dado com a identidade do agente que contratou o agente que está executando a ação.

O combinador descrito na faceta comunicativa é:

- patiently A: representa espera ocupada enquanto a ação A falha, ou seja, a ação A é executada enquanto seu estado de terminação for de falha.

Exemplo

- A seguinte ação, primeiramente contrata um novo agente e passa esse agente à sua segunda parte (após o and then). Então a ação associa este agente ao identificador "Monitor" e envia uma mensagem contendo a abstração da ação que o agente deve executar. Estas duas últimas ações são executadas intercaladamente de acordo com as características do combinador and.

```
| subordinate an agent
and then
| | bind the given agent to "Monitor"
| and
| | send a message[to the given agent][containning abstraction of Monitor-Function]
```

2.3 Considerações Finais

Este capítulo apresentou os conceitos básicos do formalismo da Semântica de Ações, que será utilizado no restante deste trabalho.

Para uma noção mais detalhada da Semântica de Ações, pode-se referir a [Mos92]; para uma introdução, pode-se referir também a [Wat91]. Para uma comparação entre Semântica de Ações e outros formalismos pode-se consultar [Mos92, Lee89, Mus96].

Para ter acesso a trabalhos gerando implementações a partir de ações, consulte [BMW92], que define o sistema ACTRESS e [Men98], que define o sistema ABACO.

Capítulo 3

Data Warehousing

Este capítulo expõe os conceitos e definições relacionados a *data warehousing*. A seção 3.1 aborda as principais características dos componentes de um sistema de *data warehousing*. As seções 3.2 e 3.3 abordam os sistemas de *data warehouse* WHIPS e SAGU, que estão sendo desenvolvidos e implementados por pesquisadores das universidades de Stanford e da Federal do Paraná, respectivamente.

Um *data warehouse* [Squ95, Wid95] é um grande repositório de dados integrados, coletados de fontes distribuídas, autônomas e heterogêneas, tais como: sistemas legados, aplicações operacionais e fontes externas.

A finalidade específica de um sistema de *data warehousing* é proporcionar um ambiente apropriado, tipicamente mantido separado das bases de dados operacionais da organização [CD97], aonde Sistemas de Suporte à Decisão [Oli98] possam produzir informações para auxiliar seus usuários nos processos de tomadas de decisão, descoberta de informações implícitas nos dados da organização e identificação de tendências de mercado.

A tecnologia de *data warehousing* é considerada como uma *"abordagem antecipada"* [Wid95] para integração de dados, porque os dados que são de interesse para o *warehouse*[1] são extraídos antecipadamente de suas fontes, traduzidos e filtrados apropriadamente para serem armazenados em um repositório único e centralizado. Então, quando a consulta do usuário chega ao

[1] O termo *warehouse* neste contexto, é utilizado para representar a base de dados do *data warehouse*.

warehouse, ela não tem que ser traduzida aos moldes dos fontes originais para execução, economizando tempo e recursos especialmente se as fontes forem muitas e distantes entre si [HAM95].

Esta abordagem é especialmente apropriada para [Wid95, HAM95]:

- Usuários que necessitam de dados específicos, utilizados somente para operações de leitura.
- Aplicações que utilizam consultas complexas que exigem alto desempenho de execução, mas que não necessariamente precisam dos estados mais recentes das informações.
- Aplicações que necessitam de acesso a cópias privadas dos dados produzidos pelo processamento operacional, para transformação dos mesmos. Essas transformações podem incluir agregação e sumarização de dados de outras aplicações, eliminação ou correção de dados duplicados e inconsistentes ou inserção de valores padrão.
- Aplicações que necessitam armazenar informações que não são mantidas nas fontes originais, envolvendo um histórico dos dados da organização.

3.1 Componentes de um Sistema de Data Warehousing

Os componentes de um sistema de *data warehousing* (figura 3.1, extraída de [WGL96]) podem ser divididos em dois grupos de software:

- *Componentes de integração*: responsáveis pela coleta de dados das fontes e manutenção das visões materializadas [ZGW95] no *warehouse*.
- *Componentes de análise e consulta*: responsáveis por satisfazer as necessidades de informações de usuários específicos do *data warehouse*.

Esses componentes trabalham em conjunto, pois os dados que os *componentes de integração* irão disponibilizar no *warehouse* são aqueles utilizados pelos *componentes de análise e consulta* realizarem suas operações.

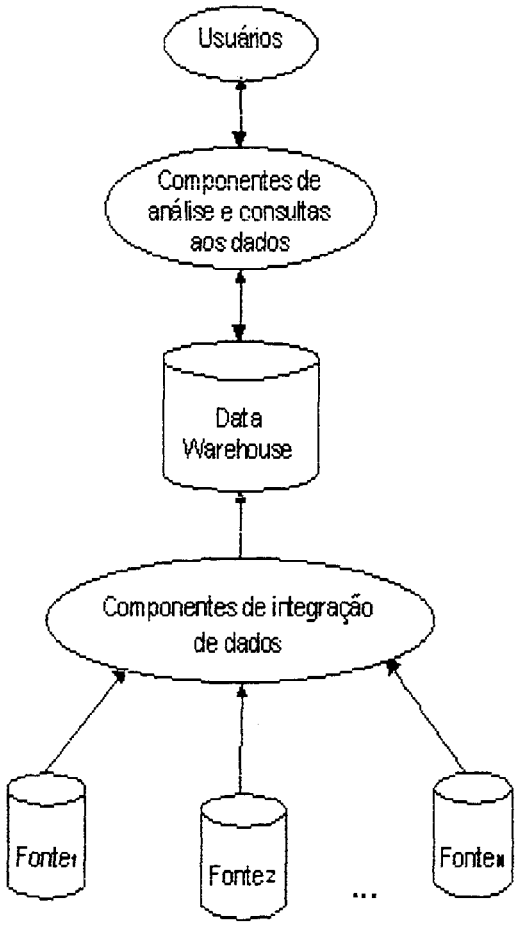

Figura 3.1: Arquitetura Básica de um Sistema de Data Warehousing

3.1.1 Componentes de Integração

São os componentes responsáveis por realizar o carregamento inicial de dados e verificar as ocorrências de atualizações nas fontes de dados externas, notificá-las ao *data warehouse*, obtê-las das fontes, carregá-las e executar os procedimentos necessários para integrá-las e armazená-las no *warehouse*.

Os componentes de integração podem ser divididos em módulos de software responsáveis por desempenhar tarefas específicas. Existe um componente responsável por monitorar as fontes e notificar ao *warehouse* a ocorrência de atualizações de seu interesse. Neste ponto, as fontes devem ser consultadas e as atualizações devem ser carregadas. Como as fontes são heterogêneas,

faz-se necessário um componente para realizar a tradução das informações do formato utilizado nas fontes para o formato utilizado no *data warehouse.*

Como os dados são originários de diferentes fontes, antes de serem armazenados no *warehouse*, eles são validados por um conjunto de programas de integração e transformação de dados [CD97], que realizam operações de consolidação, formatação e depuração dos dados, com o intuito de prover uma visão unificada de todos os dados da organização.

Os programas de integração e transformação de dados são muito suscetíveis à manutenção, pois precisam ser modificados cada vez que o ambiente operacional muda, e também cada vez que o próprio ambiente do *data warehouse* sofre alterações. Eles são responsávies por uma grande variedade de funções [Squ95], tais como:

- reformatação e recálculo de dados;
- modificação de estruturas chaves dos dados;
- adição de elementos de tempo aos dados no *warehouse*;
- sumarização de dados;
- combinação de dados de múltiplas fontes,
- resolução de inconsistências dos dados.

Para realizarem essas transformações, os programas de integração e transformação de dados utilizam informações armazenadas no depósito de metadados, que é uma base de dados que contém informações sobre os dados existentes no *warehouse* e de onde seu conteúdo provém; informações sobre as fontes e como acessá-las; informações sobre as transformações e os mapeamentos entre os dados. Existem diferentes estilos de dados em um *data warehouse*: alguns são detalhados, outros sumarizados, dados históricos da organização são também mantidos no *warehouse.* Mesmo que eliminados das aplicações operacionais, torna-se necessário manter também no depósito de metadados um registro das alterações ocorridas nos dados da organização ao longo do tempo [Squ95].

Num ambiente de *data warehousing*, os dados são consultados e analisados, mas não são alterados e são os próprios usuários finais que propõem as consultas ao *warehouse*, utilizando ferramentas amigáveis.

Para facilitar a visualização dos dados e melhorar a performance das consultas, os dados são tipicamente modelados dimensionalmente [RM99], ou seja, os dados podem ser visualizados como um "cubo" contendo três, quatro ou até cinco ou mais dimensões, de forma a facilitar o entendimento e a navegação dos dados pelo usuário pelo software, tornando a interface do usuário simples e o desempenho das consultas aceitável [Kim96], pois as consultas dos sistemas de suporte à decisão exigem que as informações possam ser visualizadas em diversos pontos, para possibilitar o cruzamento dos dados da organização, permitir o entendimento de como os negócios estão funcionando e proporcionar um ambiente em que sejam realizadas consultas com tempo de resposta rápido.

A modelagem dimensional produz estruturas de base de dados que facilitam o entendimento do relacionamento dos dados e a proposição de consultas dos usuários finais, pois minimizam o número de tabelas e dos relacionamentos entre elas, reduzem a complexidade da base de dados e o número de junções necessários para atender às consultas propostas.

Nesta modelagem existe uma estrutura central denominada *tabela de fatos*, que armazena informações utilizadas para realizar medições numéricas do negócio (por exemplo, uma tabela que armazena o valor total da venda diária de um produto em uma filial). As informações da tabela de fatos representam um nível fundamental de dados para análise, referentes ao processo de negócio que está sendo modelado [Kim96].

A tabela de fatos possui ligações com outras tabelas que contém informações textuais sobre os dados que estão sendo trabalhados (por exemplo, dados sobre produto e filial), chamadas *tabelas de dimensões*. As chaves únicas de cada tabela de dimensão compõem a chave composta da tabela de fatos.

Cada uma das medições da tabela de fatos é obtida através da intersecção das informações de todas as dimensões, sendo assim, haverá uma linha na tabela de fatos para cada combinação única dos domínios de todas as chaves de todas as tabelas dimensionais. As tabelas dimensionais

armazenam as descrições textuais dos componentes a serem analisados no negócio.

Geralmente, as tabelas de dimensões não são normalizadas e possuem uma ou mais hierarquias embutidas[2], que especificam os níveis possíveis de agregações de dados, determinando assim a granularidade da visualização das informações, para possibilitar a realização das operações de OLAP[3] sobre os dados.

Outro recurso utilizado em *data warehousing* é o armazenamento de visões materializadas [EN94] - tabelas armazenadas na base de dados, que são derivadas da combinação entre dados de outras tabelas - para prover as necessidades de informações dos *componentes de análise e consulta*, agilizando a execução das consultas complexas mais freqüentemente propostas. A seleção do conjunto apropriado de visões deve ser feita levando-se em conta a minimização do tempo total de resposta das consultas e dos custos de manutenção das visões selecionadas [Gup97].

Quando as fontes são alteradas, as visões no *warehouse* precisam ser atualizadas de acordo, como forma de garantir a consistência dos dados, refletindo o estado real das fontes. Essas atualizações podem ser feitas através do reprocessamento de todos os dados da visão ou pela manutenção incremental da visão [WZG97, GZW97, ZGH95, GM98, Zhu99].

3.1.2 Componentes de Análise e Consultas

São os programas de *front-end*, com finalidades específicas dentro das aplicações informacionais, responsáveis por utilizar os dados disponibilizados no *warehouse* e transformá-los em informações úteis para a tomada de decisão, permitindo um gerenciamento de causas e efeitos mais eficiente e conseqüentemente, a antecipação de políticas para tal.

As ferramentas para análise e consulta, de acordo com Kimble [Kim96], podem ser um pacote de software independente, que possibilita a elaboraçao de consultas *ad-hoc* e a utilização de instruções SQL[4] para exibir e formatar relatórios tanto no modo texto como no modo gráfico

[2]Por exemplo: dia - mês - bimestre - ano são hierarquias em data

[3]On-Line Analytical Processing

[4]Structured Query Language

ou uma aplicação monolítica construída sobre um domínio específico, com uma interface de usuário personalizada.

Outra variedade de programas que trabalham neste contexto, são as ferramentas de mineração de dados (*data mining* [AZ97, BS97, MR98, Mug97]) que são softwares desenvolvidos com base em técnicas de inteligência artificial, que exploram os dados para identificar relacionamentos em variáveis que antes eram previamente independentes, de acordo com critérios pré-determinados [Oli98].

As planilhas eletrônicas também são programas que trabalham em conjunto com ferramentas de análise mais poderosas, sendo utilizadas para a apresentação dos resultados obtidos, devido a facilidade para exibição e impressão de relatórios e gráficos para a apresentação dos dados.

Os *componentes de análise e consulta* apresentam uma série de funcionalidades que facilitam a análise dimensional, permitindo a manipulação de dados que tenham sido agregados em várias hierarquias, sintetizando informações empresariais através da visualização comparativa, personalizada e também por meio da análise de dados históricos e projetados [IWG99]. Para isso, executam uma série de operações realizadas sobre os dados modelados dimensionalmente, tais como [CD97, AGS97]:

- *rollup*: diminuir o nível de detalhamento dos dados;
- *drill-down*: aumentar o nível de detalhamento dos dados;
- *drill-across*: combinar tabelas de fatos diferentes de acordo com dimensões compartilhadas;
- *slice-and-dice*: selecionar e projetar alguns sub-conjuntos de dados;
- *pivoting*: trocar o foco de visão dos dados;
- *ranking*: ordenação das consultas de acordo com os valores dos fatos;
- *selection*: selecionar dados;
- *computed attributes*: definir processamentos para atributos.

3.2 O Sistema WHIPS

Diversos trabalhos, relacionados com a criação e manutenção de um *data warehouse*, foram desenvolvidos por um grupo de pesquisadores da Universidade de Stanford [5]. O projeto principal, que engloba outros trabalhos, é o Sistema WHIPS -WareHouse Information Prototype at Stanford [WGL96, LZW97, HAM95, Zhu99].

O Sistema WHIPS coleta dados de fontes heterogêneas, transforma e sumariza-os de acordo com as especificações, e integra-os no *warehouse*. Os principais objetivos do WHIPS são:

- suportar diferentes tipos de fontes;
- manter incrementalmente as visões materializadas, sem a necessidade de interromper as consultas executadas pelos usuários finais;
- garantir a consistência dos dados.

A arquitetura do Sistema WHIPS foi projetada para atender as principais propriedades desejáveis em um sistema de *data warehousing* [WGL96]:

- *Modularidade Plug-and-Play*: permite que o sistema trabalhe com qualquer tipo de fonte de dados, e qualquer tipo de base de dados para armazenar as informações do *data warehouse*. Esta funcionalidade é obtida através da implantação de componentes Tradutores (*Wrappers*) para todos os módulos que trabalham com bases de dados.
- *Escalabilidade*: o componente Integrador (*Integrator*) deve suportar o aumento do volume de dados no *warehouse*, permitindo a distribuição de seu trabalho entre mais máquinas e módulos.
- *Operações 24 horas por dia em 7 dias da semana*: como várias organizações possuem operações em múltiplas zonas de tempo, não existe um tempo apropriado para desligar o sistema e realizar qualquer tipo de manutenção necessária. Então o sistema deve estar

[5] http://www-db.stanford.edu/warehousing.

apto à adição de novas fontes de dados, e à realização da manutenção incremental das visões materializadas sem a necessidade de interromper o processamento de consultas.

- *Consistência de dados*: quando os dados são coletados das fontes autônomas, as visões materializadas resultantes podem ser inconsistentes, ou seja, elas podem refletir um estado que nunca existiu na fonte de dados. Para evitar este problema, no Sistema WHIPS é possível especificar o nível desejado de consistência dos dados, sendo que o sistema utiliza algoritmos de manutenção para garantir diferentes níveis de consistência dos dados.

- *Suporte para diferentes tipos de fontes*: o componente Integrador (*Integrator*) deve estar habilitado a trabalhar com diversos tipos de fontes de dados, e extrair dados das mesmas de maneira eficaz.

O sistema WHIPS é composto por vários módulos distintos. Cada módulo foi implementado como um objeto CORBA [OMG95, YD96], utilizando a implementação ILU do CORBA [YD96]. A comunicação entre os objetos é executada em um ambiente de objetos distribuídos, onde cada objeto possui um identificador único que é utilizado para possibilitar a comunicação entre eles. Uma das principais vantagens de implementar-se os módulos como objetos CORBA, é que o processo de comunicação é orientado através do identificador do módulo de destino e não através da localização do módulo. Com isso, torna-se mais fácil redistribuir os módulos quando necessário.

O modelo relacional é utilizado para representar os dados no *warehouse*: as visões são definidas no modelo relacional e o *warehouse* armazena relações. Os dados das fontes são convertidos para o modelo relacional pelo Translator (*Wrapper*), antes de serem enviados a qualquer outro módulo.

O principal enfoque do Sistema WHIPS está no desenvolvimento dos componentes de integração de um sistema de *data warehousing* (figura 3.2, também obtida em [WGL96]). O sistema é composto de vários módulos distintos, mas que se comunicam mesmo residindo em máquinas diferentes. Cada componente é responsável por uma tarefa distinta, enumerada a seguir [WGL96]:

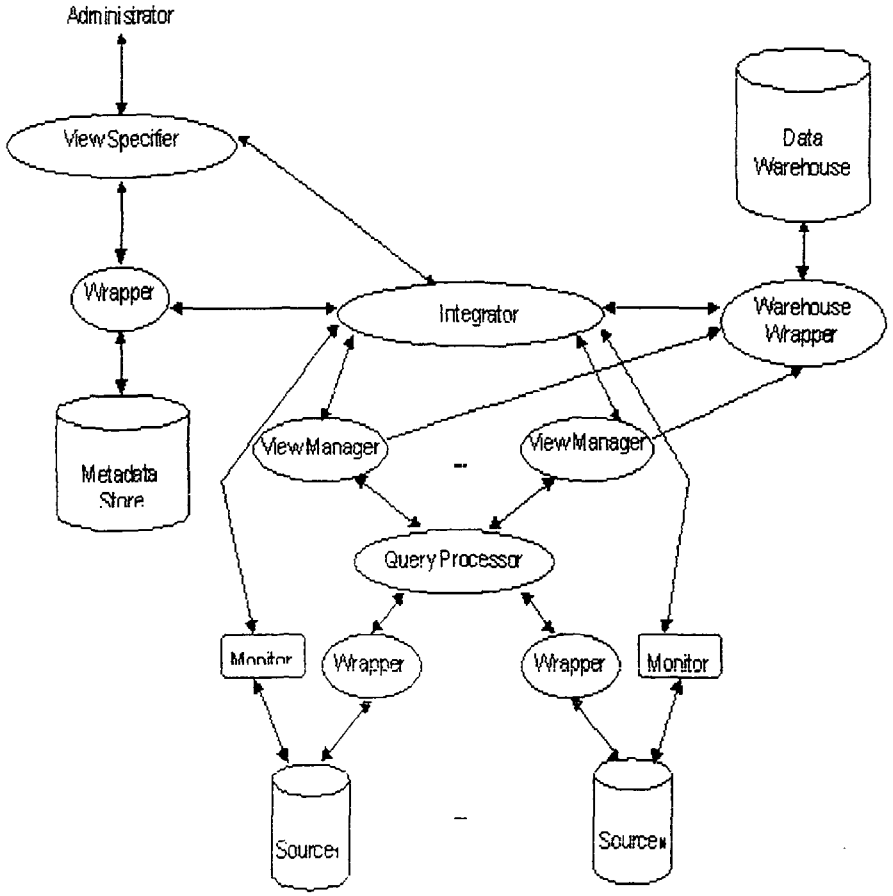

Figura 3.2: Componentes de Integração do Sistema WHIPS

- *Sources (Fontes)*: as fontes de dados são autônomas e externas ao *data warehouse.* Podem incluir tabelas relacionais, estruturas orientadas a objetos, arquivos, imagens, textos, legados, EDI ou dados provenientes da Internet. As fontes podem ser divididas em grupos, de acordo com o método utilizado para realizar a propagação de suas atualizações ao *warehouse* [Wid95]:

  - Fontes cooperativas: possuem *triggers* [EN94] ou outras capacidades de base de dados ativas [CW95], tal que quando ocorrem atualizações de interesse para o *warehouse*, as fontes são programadas para automaticamente notificar o *warehouse.*

  - Fontes que trabalham com log: mantêm um *log* [EN94] que pode ser consultado ou

inspecionado, para se extrairem as atualizações de interesse para o *warehouse*.

- Fontes consultáveis: permitem realizar uma verificação periódica para detectar e extrair as atualizações através de consultas diretas à fonte.
- Fontes que trabalham com *snapshots*[LHM97]: são aquelas que não oferecem *triggers, logs* ou disponibilidade para consultas. Então são necessários *dumps* ou *snapshots* periódicos de dados, realizados quando as fontes estão *off-line*, e através da comparação de sucessivos *snapshots* são detectadas as alterações.

- *Monitors (Monitores)*: são responsáveis por detectar as atualizações ocorridas nas fontes, que são de interesse para o *data warehouse* e relatá-las ao Integrador. Existe um Monitor específico para cada fonte de dados, sendo que as tarefas que ele desempenha são realizadas de acordo com o tipo da fonte com que esta trabalhando [Wid95]. Se as fontes são sistemas de base de dados com todas as funcionalidades, é possível simplesmente definir um conjunto apropriado de *triggers* e o Monitor aguarda até ser notificado das alterações. Outra opção para o Monitor é examinar as alterações em um *log* e extrair as atualizações de seu interesse. Nos casos dos sistemas legados, essa tarefa é mais difícil, pois muitas vezes não existem *triggers* ou *logs*, ou quando existem não são disponíveis a componentes externos. Existem então duas soluções possíveis: a aplicação que altera os dados da fonte é modificada para emitir uma mensagem de notificação de atualização apropriada ao Monitor ou um programa utilitário é escrito, para periodicamente gerar um arquivo com os dados das fontes, para que o Monitor compare sucessivas versões deste arquivo.

- *Wrappers (Tradutores)*: cada Tradutor é responsável por traduzir as informações de uma única fonte, para a representação interna usada no *data warehouse*. No caso do Sistema WHIPS, esta representação é relacional. Usando um Tradutor para cada fonte são escondidos detalhes das consultas propostas, pois todos os Tradutores suportam o mesmo método de interface, mas os seus códigos internos são específicos à linguagem nativa de cada fonte de dados.

- *Query Processor (Processador de Consultas)*: o Processador de Consultas é responsável pelo processamento de consultas distribuído, ou seja, ele recebe uma *consulta global* e de

acordo com as relações armazenadas em cada fonte, deve ser capaz de propor consultas locais aos Tradutores. O Processador de Consultas aguarda o retorno do resultado de cada consulta local proposta aos Tradutores, e então ajusta os resultados como necessário para gerar-se a resposta à consulta global. Como os Tradutores escondem a sintaxe das consultas específicas às fontes, o Processador de Consultas gera as consultas globais em um formato único, no caso do Sistema WHIPS como se os fontes fossem todas bases de dados relacionais.

- *View Manager (Gerenciador de Visão)*: existe um módulo Gerenciador de Visão responsável pela manutenção de cada visão materializada no *warehouse*, usando um algoritmo de manutenção de visões [ZGW95], para manter a consistência dos dados. No caso do projeto WHIPS é utilizado o Algoritmo Strobe [ZGW97]. Existem duas vantagens em utilizar-se um Gerenciador de Visão para cada visão: a primeira é que o trabalho de manutenção de cada visão pode ser feito paralelamente em diferentes máquinas e a outra é que cada visão pode utilizar diferentes algoritmos de manutenção para assegurar diferentes níveis de consistência de acordo com necessidades particulares.

- *Integrator (Integrador)*: coordena a inicialização do sistema de *data warehousing*, incluindo a adição de novas fontes e a inicialização das visões. A principal função do Integrador é facilitar a manutenção das visões, através da verificação de quais atualizações precisam ser propagadas para quais visões. Para isto o Integrador utiliza um conjunto de regras que especificam quais Gerenciadores de Visões estão interessados em quais atualizações. Essas regras são geradas automaticamente a partir da Árvore da Visão (*View Tree* - apresentada a seguir) quando cada visão é definida.

- *Warehouse Wrapper (Tradutor do Warehouse)*: o *warehouse* no Sistema WHIPS pode ser qualquer base de dados relacional, mas existem algumas bases que são otimizadas para as tarefas executadas em um *data warehouse* e que podem ser mais apropriadas, como por exemplo Redbrick[6]. O Tradutor do Warehouse recebe todas as definições de visões e todas as atualizações para os dados de uma visão em formato interno, e os traduz

[6]http://www.redbrick.com

em uma sintaxe específica utilizada na base de dados do *data warehouse*. O Tradutor do Warehouse esconde as particularidades do *warehouse* de todos os outros módulos do Sistema WHIPS, permitindo que qualquer sistema de base de dados possa ser usado no *warehouse*.

- *Metadata Store (Depósito de Metadados)*: o Depósito de Metadados guarda as informações sobre as fontes e como consultá-las, sobre as relações armazenadas em cada fonte e sobre o esquema de cada relação. Também guarda os caminhos de todas as definições de visões.

- *View Specifier (Especificador de Visões)*: no projeto WHIPS, as visões são definidas em um subconjunto de SQL que inclui comandos de *seleção, projeção e junção* (*select-project-join*) para visões sobre todas as fontes, sem aninhamentos. Opcionalmente, na definição da visão pode-se especificar qual algoritmo de manutenção será utilizado para manter a sua consistência. Quando uma visão é definida, o Especificador de Visões realiza uma análise sintática [Bro89] na estrutura interna da consulta, gerando uma árvore, que é chamada de Árvore da Visão (*View Tree*), adiciona informações relevantes consultadas no Depósito de Metadados (tais como atributos chaves, tipos de atributos) à Árvore da Visão e a envia para o Integrator.

Os módulos dos componentes de integração do Sistema WHIPS são responsáveis por três operações distintas que ocorrem em um sistema de *data warehousing* [WGL96]:

Inicialização do Sistema: na inicialização do sistema o Integrador publica seu identificador e cria o Processador de Consultas. Todos os outros módulos que estão inicializando contatam o Integrador para se identificarem. Os Monitores e os Tradutores também contatam o Integrador para registrar os metadados das fontes que são enviados ao Depósito de Metadados e ao Processador de Consultas. Novas fontes podem ser adicionadas ao sistema seguindo-se estes procedimentos.

Definição e inicialização de visões: as visões são definidas pelo administrador do sistema através do Especificador de Visões. O Especificador de Visões consulta o Depósito de Metadados

para realizar a verificação e validação dos dados para cada visão especificada, e passa a definição da visão para o Integrador, que cria um Gerenciador de Visão para a visão. O Integrador também notifica os Monitores de todas as fontes envolvidas na visão. Então, o Gerenciador de Visão é responsável pela inicialização e manutenção da visão. Para inicializar a visão, o Gerenciador de Visão gera uma consulta global correspondente à definição da visão e passa esta consulta ao Processador de Consultas. O Processador de Consultas gera e envia consultas aos Tradutores das fontes envolvidas, e os resultados retornados são ajustados como necessário para gerar a resposta à consulta global, que é enviada ao Gerenciador da Visão. O Gerenciador da Visão encaminha a resposta da consulta global ao Tradutor do Warehouse para inicializar a visão.

Manutenção das visões: cada Monitor de uma relação R detecta as atualizações para R que ocorrem na sua fonte e notifica o Integrador. O Integrador notifica todos os Gerenciadores de Visões que trabalham com a visão. Cada Gerenciador de Visão usa um dos algoritmos de Strobe [ZGW97] para realizar manutenção da visão e assegurar a sua consistência. Os resultados obtidos são enviados ao Tradutor do Warehouse, que aplica-os ao *warehouse*, deixando-o em um novo estado consistente.

Durante o processo de manutenção das visões as mensagens entre os módulos são enviadas de forma assíncrona, sendo que o atraso na comunicação não deve segurar o processamento de qualquer módulo.

Na arquitetura do Sistema WHIPS, as mensagens enviadas pelas fontes podem chegar ao Gerenciador de Visão através de dois caminhos:

- as atualizações são enviadas do Monitor para o Integrador que as envia para o Gerenciador da Visão;
- os resultados das consultas são enviadas do Tradutor para o Processador de Consultas que os envia para o Gerenciador da Visão.

Esta arquitetura não pode garantir que duas mensagens enviadas por caminhos diferentes cheguem ao Gerenciador da Visão na mesma ordem, podendo causar anomalias nos resultados

obtidos. Então a solução adotada no Sistema WHIPS é a utilização de números seqüênciais para identificar as mensagens passadas.

Cada Monitor tem seu próprio contador de seqüência para cada relação, e cada atualização é identificada com um número quando é enviada ao Integrador. Cada Tradutor também identifica o resultado da consulta de acordo com o número seqüêncial da última atualização enviada pelo Monitor correspondente. O Processador de Consultas constroi uma matriz de seqüência de números para cada relação, de acordo com os números retornados pelos Tradutores, como parte do resultado da consulta.

O Gerenciador da Visão também cria outra matriz de seqüência de números para cada relação, correspondente às últimas atualizações recebidas. Quando o Gerenciador da Visão recebe o resultado de uma consulta, ele compara o número identificador da mesma com os números da sua própria matriz. Se qualquer número de seqüência for maior do que o número de seqüência correspondente na sua matriz, o Gerenciador da Visão aguarda pela atualização de número anterior.

Esta solução requer que cada fonte consultada receba um número de seqüência pelo menos mais alto que as atualizações refletidas no resultado da consulta, e que a comunicação entre o Monitor e o Tradutor esteja envolvida. Mas por outro lado, a grande vantagem é que nenhum mecanismo de controle de concorrência é necessário.

3.3 O Projeto SAGU

Nas instituições de ensino e pesquisa, a qualidade do serviço prestado, o redirecionamento dos temas e currículos, o controle rigoroso e transparente dos recursos utilizados também têm como um dos alicerces a informação.

Na Universidade Federal do Paraná - UFPR[8], está em desenvolvimento um sistema de Apoio ao Gerenciamento Universitário - SAGU [Poz00], que tem como objetivo a criação de um ambiente para produzir informações relevantes visando apoiar e agilizar o processo decisório e conseqüentemente proporcionar o desenvolvimento acadêmico.

Este projeto está sendo desenvolvido pelo grupo KDD[9] - Knowledge Discovery in Databases, do Departamento de Informática da UFPR, em parceria com a UNISYS[10], e visa integrar informações heterogêneas, originárias de diferentes setores desta universidade, através da construção de um *data warehouse*, a partir do qual técnicas de análise possam ser utilizadas para extrair conhecimentos.

Diversas atividades de pesquisa do Departamento de Informática da UFPR, estão relacionadas com as tecnologias envolvidas no projeto SAGU, sendo que este estudo está sendo feito dentro de uma perspectiva teórica e prática, pois tem como base os problemas reais enfrentados pela instituição, proporcionando o ambiente de desenvolvimento ideal para colocar em prática os conceitos estudados e as soluções desenvolvidas.

Todas as funções necessárias para o funcionamento do *data warehouse* do projeto SAGU, estão sendo implementadas em componentes de software distintos, responsáveis por desempenhar tarefas específicas e interagir entre si, quando necessário.

Cada um destes componentes possui um identiticador único, conhecido por todos os outros compontentes e utilizado para a comunicação entre eles, realizada através dos protocolos de comunicação TCP/IP.

Os dados no *data warehouse* estão modelados dimensionalmente, utilizando *esquemas estrela*

[8] http://www.ufpr.br

[9] http://www.inf.ufpr.br/mestrado/projetos/kdd

[10] http://www.unisys.br

[RM99], e as bases de dados estão implementadas em Oracle[11].

A arquitetura do *data warehouse* do Sistema WHIPS [WGL96], foi escolhida como base para o desenvolvimento da arquitetura do *data warehouse* do projeto SAGU, por se tratar de uma proposta bastante genérica e aceita na literatura.

Os principais objetivos do *data warehouse* do projeto SAGU são similares aos do Projeto WHIPS, adaptados à realidade da UFPR.

Na arquitetura do *data warehouse* do projeto SAGU (figura 3.3) os dados são carregados das fontes (que são completamente autônomas) pelos Monitores. Existe um Monitor específico para cada fonte de dados, como uma forma de esconder detalhes particulares de cada fonte e possibilitar a utilização de um método de interface uniforme no *warehouse*. O Monitor é uma peça de software que periodicamente verifica a fonte, procurando por atualizações nos dados que são de interesse para o *data warehouse*.

As atualizações encontradas nas fontes são então enviadas para o módulo Tradutor, que é responsável pela tradução desses dados para o formato interno utilizado no *warehouse*. Assim como no caso dos Monitores, existe um Tradutor específico para cada fonte. Após a tradução, o Monitor envia as atualizações para o módulo Integrador.

O Integrador recebe os dados traduzidos e é responsável por notificar o módulo Carregador para transportar esses dados para a Base de Dados de Trabalho, projetada para armazenar as atualizações dos dados das fontes.

Quando o módulo Carregador finaliza o processo de transporte de dados, o Integrador notifica o módulo Consolidador que deve resolver as possíveis inconsistências dos dados armazenados na Base de Dados de Trabalho, as diferenças existentes entre os tipos de dados e definir os relacionamentos entre as múltiplas fontes de dados para combinar as atualizações recebidas com os dados existentes no *warehouse*.

Após serem resolvidas as inconsistências, o Consolidador deve atualizar a Base de Dados Espelho do *data warehouse* (que é identica à Base de Dados de Produção e armazena todos os dados utilizados no *warehouse*). Todas as atualizações são feitas na Base de Dados Espelho,

[11] http://www.oracle.com/br

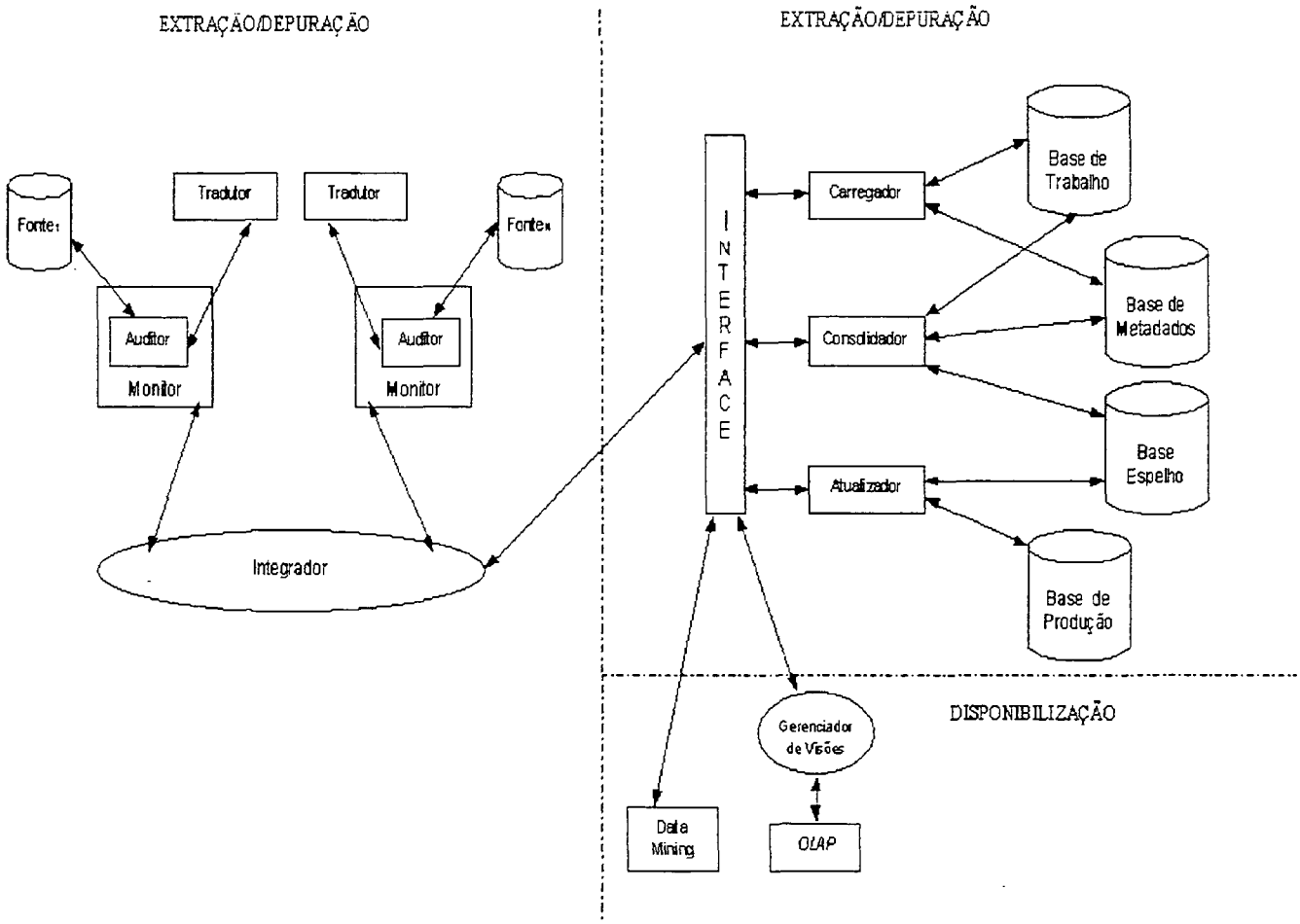

Figura 3.3: Arquitetura do Data Warehouse do Projeto SAGU

para que durante este processo, a Base de Dados de Produção não tenha que ficar *off-line*.

Estando a Base de Dados Espelho em um estado consistente com o estado das fontes, para finalizar o processo de atualização, o Atualizador deve transportar a Base de Dados Espelho para a Base de Dados de Produção do *data warehouse*.

Um exemplo prático de um tópico que está sendo abordado no Projeto SAGU é referente aos dados dos alunos que ingressaram em algum curso na UFPR, proporcionando informações sobre o histórico dos mesmos. Esses dados estão armazenados em uma fonte de dados autônoma, hierárquica e externa ao sistema de *data warehousing* do Projeto SAGU.

Existe um componente Monitor, responsável por periodicamente verificar o arquivo de log

desta fonte em busca de atualizações. No caso da existência de atualizações, elas são traduzidas pelo componente Tradutor, que neste exemplo transforma os dados do formato EBCDIC utilizado pela fonte, para o formato ASCII, para que seja possível a leitura dos dados atualizados, por componentes internos do data warehouse.

3.4 Considerações Finais

Este capítulo abordou os principais conceitos da abordagem de *data warehousing*, apresentou algumas aplicações em que esta abordagem é especialmente indicada e os componentes de integração, análise e consulta de *data warehousing*.

Também foram apresentados sistemas de *data warehousing* em desenvolvimento pelas universidades: o sistema WHIPS, da Universidade de Stanford e o Projeto SAGU, da Universidade Federal do Paraná. Este último utilizado como base para o desenvolvimento da Proposta de Especificação Formal para Data Warehousing, a que se propõe este trabalho.

Para obter informações mais detalhadas sobre a arquitetura, o processo de construção e a modelagem de dados em um *data warehousing* pode-se referir a [Dev97, Kim96].

No próximo capítulo, será apresentada a descrição formal dos componentes do Projeto SAGU.

Capítulo 4

Descrição formal dos componentes do Data Warehouse do Projeto SAGU

Este capítulo trata da especificação formal dos componentes do *data warehouse* do Projeto SAGU, acompanhada de uma explicação informal de cada um deles.

Os componentes do *data warehouse* do Projeto SAGU: Integrador, Monitor, Auditor, Tradutor, Carregador, Consolidador, Atualizador e Semáforo da Base de Dados de Trabalho estão implementados como um sistema de agentes distribuídos, sendo que cada agente é responsável por desempenhar uma tarefa específica, podendo comunicar-se com os outros agentes, enviando e recebendo mensagens, mesmo residindo fisicamente separados. Um agente representa a identidade de um processo único, embutido em um meio de comunicação comum entre agentes.

A comunicação entre esses agentes é assíncrona: para enviar uma mensagem, um agente emite a mensagem em um meio, com outro agente especificado como receptor e continua executando suas ações, não necessitando esperar até que a mensagem seja entregue ao outro agente.

Cada mensagem é vista como informação permanente, não existindo uma forma de apagar a comunicação, uma vez que a mesma tenha sido iniciada. Quando da chegada de uma mensagem

para um agente, esta é colocada no *buffer* do receptor, não interrompendo a ação que está sendo executada e permanecendo desconhecida até que o agente inspecione o *buffer*.

Toda mensagem carrega a identificação do agente remetente e a identificação do agente destinatário. Ela também possui um conteúdo, que pode ser um dado arbitrário, uma *abstração* - que é um tipo de dado que incorpora uma ação, correspondendo ao código da ação, que pode ser implementado como uma sequência de instruções de máquina ou como um ponteiro para tal sequência - a ser executada ou a identificação de outro agente. Cada mensagem transmitida possui um número serial, determinado pelo agente remetente, que a distingue de todas as outras mensagens transmitidas.

Inicialmente, somente um único agente distinto está ativo, conhecido como *user agent*. No caso do Projeto SAGU este *user agent* é o agente Integrador, responsável por toda a inicialização do processo, contratação dos outros agentes componentes do sistema e especificação das ações a serem executadas.

4.1 Componente Integrador

O Integrador é responsável pela inicialização de todos os componentes do sistema de Data Warehousing do Projeto SAGU e pela coordenação do processo de carregamento e manutenção dos dados existentes.

Na inicialização do sistema, o Integrador recebe uma lista contendo informações sobre os Monitores (componentes responsáveis pela verificação de atualizações nas fontes de dados) que fazem parte do sistema de Data Warehousing e sobre as suas fontes de dados correspondentes. Esta lista é chamada de MSList, sendo que cada elemento desta lista é uma tupla que contém os seguintes dados:

- Monitor Identification - Identificador único que o Data Warehouse utiliza para comunicar-se com o Monitor, representado como uma string.

- Source Identification - Identificador único utilizado pelo Monitor para comunicar-se com a fonte de dados, representado como um agente.

- Source Type - Especifica o tipo da fonte de dados, podendo conter uma das seguintes strings: *"Cooperative-Source"* ou *"Non-Cooperative-Source"*. As fontes cooperativas são aquelas que apresentam especificidades para automaticamente notificar o Data Warehouse quando da ocorrência de atualizações e as fontes não-cooperativas não são capazes de notificar as atualizações ao Data Warehouse [HAM95].

- ABSN - Audit Block Serial Number - Número inteiro correspondente ao último bloco de registros verificado na busca de atualizações no arquivo de log das fontes não-cooperativas. Para as fontes cooperativas esse dado é inexistente.

- Wait Time - É um número inteiro que determina a periodicidade de tempo para a checagem das atualizações nas fontes não-cooperativas. Para as fontes cooperativas esse dado é inexistente.

Juntamente com a lista MSList, o Integrador recebe um número inteiro denominado Consolidation-Time, representando o tempo relativo à periodicidade da realização da consolidação dos dados resultantes das atualizações ocorridas nas fontes de dados.

A seguir apresentamos a descrição formal da lista MSList, do inteiro Consolidation-Time e do componente *Integrador*.

- MSList = list of(string, agent, string, integer, integer).
- Consolidation-Time = integer.
- Init-System(_ , _) :: MSList, Consolidation-Time → action [diverging].

(1) Init-System(L: MSList, T: Consolidation-Time) =
    | Integrator-Init-Action(L)
    hence
    | Integrator-Action(L, T)

A ação *Integrator-Init-Action* é referente à inicialização do sistema e de todos os seus módulos componentes. Todas as associações realizadas nesta ação são passadas à ação *Integrator-Action* através do combinador hence.

A ação *Integrator-Action* é responsável pela manutenção dos dados existentes no Data Warehouse, coordenando todo o processo de carregamento, depuração e integração das atualizações aos dados armazenados no Warehouse, garantindo que o Warehouse reflita um estado consistente com o estado das fontes de dados.

A ação *Integrator-Init-Action* é descrita abaixo:

- Integrator-Init-Action(_) :: MSList → action [binding | communicating | completing]
  [using current bindings | current buffer].

(2) Integrator-Init-Action(L: MSList) =

```
| give L
then
| | | unfolding
| | | | | check not(the given list is empty-list)
| | | | and then
| | | | | | | subordinate an agent
| | | | | | and give component#1 of the head of the given list
| | | | | then bind the given agent#1 to the given string#2
| | | | and give tail of the given list
| | | | then unfold
| | | or check(the given list is empty-list)
| | and subordinate an agent then bind it to "SemaphoreWDB"
| | and subordinate an agent then bind it to "Loader"
| | and subordinate an agent then bind it to "Consolidator"
| | and subordinate an agent then bind it to "Updater"
| | before
| | | | unfolding
| | | | | | | check not(the given list is empty-list)
| | | | | | and then
| | | | | | | | give head of the given list
| | | | | | and then
| | | | | | | | | check(the given string#3 is "Cooperative-Source")
| | | | | | | | and then send a message [to the agent bound to the given string#1]
| | | | | | | |           [containning the application of closure abstraction of
| | | | | | | |           CS-Monitor-Function to the given agent#2]
| | | | | | | or
| | | | | | | | | check(the given string#3 is "Non-Cooperative-Source")
| | | | | | | | and then send a message [to the agent bound to the given string#1]
| | | | | | | |           [containning the application of closure abstraction of
| | | | | | | |           NCS-Monitor-Function to the given agent#2]
| | | | | | and give tail of the given list
| | | | | | then unfold
| | | | | or check(the given list is empty-list)
| | | and send a message[to the agent bound to "SemaphoreWDB"]
| | | |                 [containning the closure abstraction of SemaphoreWDB-Function]
| | | and send a message[to the agent bound to "Loader"]
| | | |                 [containning the closure abstraction of Loader-Function]
| | | and send a message[to the agent bound to "Consolidator"]
| | | |                 [containning the closure abstraction of Consolidator-Function]
| | | and send a message[to the agent bound to "Updater"]
| | | |                 [containning the closure abstraction of Updater-Function]
```

A ação acima é executada uma única vez, na inicialização do componente Integrador. Quando esta ação é executada, ela recebe uma lista de informações, do tipo MSList, nesta ação denominada L.

Na primeira parte desta ação (até o before), são criados novos agentes (processos) representando todos os Monitores existentes na lista L e todos os outros componentes do Data Warehouse: Carregador, Consolidador, Atualizador e Semáforo da Base de Dados de Trabalho. Todos os agentes criados possuem um identificador único, conhecido por todos os outros agentes e utilizado para a troca de mensagens entre eles. A lista L e todas as associações (*bindings*) da primeira parte da ação são passadas à segunda parte da ação, através do combinador before.

Na segunda parte da ação, o Integrador envia as ações que cada agente (processo) deverá executar. Essas ações serão apresentadas no decorrer deste capítulo e representam o funcionamento de todo o Data Warehouse do Projeto SAGU. No caso dos Monitores, as ações são dependentes do tipo de fonte com a qual ele trabalha. A identificação da fonte de dados correspondente (Source identification), existente no segundo elemento da tupla da lista MSList (representada como um agente externo) é passada juntamente com a ação a ser executada pelo Monitor.

Como as mensagens trocadas entre agentes em semântica de ações, somente podem conter dados a serem enviados e ações não são dados, então elas devem ser encapsuladas em *abstrações* para poderem ser transmitidas em mensagens. A ação subordinate cria um novo processo (agente), que estará aguardando por uma abstração para executar.

A seguir a ação *Integrator-Action* é apresentada:

- Integrator-Action (_ , _) :: MSList, Consolidation-Time → action
  [binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage]
- Consolidator.Log = list of tuples.
- Updater.Log = list of tuples.

(3) Integrator-Action(L: MSList, T: Consolidation-Time) =

```
| | give L
| then
| | unfolding
| | | | check not(the given list is empty-list)
| | | and then
| | | | | give head of the given list
| | | | then Integrate-Source
| | | and
| | | | | give tail of the given list
| | | | then unfold
| | | or check(the given list is empty-list)
and
| unfolding
| | | Sleep T
| | then send a message [to the agent bound to "Consolidator"]
| |                    [containning an "OK"]
| | then receive a message [from the agent bound to "Consolidator"]
| |                       [containning a Consolidator.Log]
| | then send a message [to the agent bound to "Updater"]
| |                    [containning an "OK"]
| | then receive a message [from the agent bound to "Updater"]
| |                       [containning an Updater.Log]
| | then unfold
```

Esta ação recebe uma lista de informações, do tipo MSList, nesta ação denominada L e o inteiro T, representando o valor do intervalo de tempo para a consolidação das atualizações dos dados e através de dois loops realiza a manutenção dos dados existentes no Warehouse. Esses loops podem acontecer simultâneamente (de acordo com as características do combinador and) pois são responsáveis por tarefas distintas.

No primeiro loop, a ação *Integrate-Source* (apresentada a seguir) é responsável por notificar os Monitores existentes na lista L, para realizarem a verificação e o carregamento das atualizações ocorridas nas fontes de dados, que são de interesse para o Data Warehouse.

A segunda parte da ação (após o and) é um loop infinito, que inicia com a ação *Sleep*, deixando o processo em estado de espera de acordo com o valor contido em T, determinando assim a periodicidade de execução da ação.

Na sequência o Integrador envia uma mensagem ao agente Consolidador - apresentado adiante - para realizar a consolidação das atualizações obtidas e integrá-las à Base de Dados

Espelho do Data Warehouse. Em seguida, aguarda o recebimento de uma mensagem de retorno do Consolidador, contendo uma lista de log das operações realizadas, denominada Consolidator.Log, indicando a finalização do processo de consolidação.

Então, o Integrador envia uma mensagem ao agente Atualizador - também apresentado adiante - para que este realize o transporte da Base de Dados Espelho, agora atualizada, para a Base de Dados de Produção e aguarda o recebimento de uma mensagem de retorno do Atualizador, contendo uma lista de log das operações realizadas, denominada Updater.Log, indicando a finalização do processo de transporte. A ação retorna ao começo do loop iniciando o processo novamente.

São realizados procedimentos não automatizados para a verificação dos arquivos de log das operações realizadas pelo Consolidador e pelo Carregador, e para os possíveis erros encontrados, são realizados procedimentos que não são especificados nesta ação, pois são feitos de forma manual e esporádica.

A ação *Sleep* deixa o sistema em estado de espera, de acordo com o valor especificado em T, determinando assim a periocidade da atualização do Warehouse. Esta função é dependente de implementação e não é especificada aqui.

- Sleep _ :: integer → action

(1) Sleep I = □

A notação □ indica que este item não é definido nesta parte da especificação.

A ação *Integrate-Source*, utilizada pela ação *Integrator-Action* apresentada acima, é especificada a seguir:

- Integrator-Source :: action [receiving a tuple | completing][using current bindings | current buffer].

(4) Integrate-Source =
    | check(the given string#3 is "Cooperative-Source") then
    | | | give the given string#1
    | | and then Integrate-Cooperative-Source
    or
    | check(the given string#3 is "Non-Cooperative-Source") then
    | | | give(the given string#1, the given integer#4, the given integer#5)
    | | and then Integrate-Non-Cooperative-Source

Na ação *Integrate-Source*, o Integrador recebe uma tupla de dados contendo cinco elementos: o identificador do Monitor, o identificador da fonte, o tipo da fonte, o último ABSN verificado e a periodicidade de verificação das atualizações.

Então de acordo com o tipo da fonte, o Integrador executa uma das seguintes ações:

- *Integrate-Cooperative-Source*: para fontes cooperativas que repassam automaticamente as atualizações ocorridas ao Monitor. O identificador do Monitor é enviado à ação.
- *Integrate-Non-Cooperative-Source*: para fontes não cooperativas, onde o Monitor é responsável por verificar e carregar as atualizações ocorridas na fonte. O identificador do Monitor, o último ABSN pesquisado e a periodicidade de verificação das atualizações são enviados à ação.

A ação *Integrate-Cooperative-Source* é apresentada a seguir:

- Integrate-Cooperative-Source :: action [receiving an string|binding|storing|diverging|communicating] [using current bindings | current buffer | current storage].
- Translated-Update-List = list of tuples.
- Log-List = list of tuples.
- Errors-List = list of tuples.

```
(5) Integrate-Cooperative-Source =
        | | allocate a cell
        then
        | | | bind the given cell to "Monitor"
        | | and store the given string#1 in it
      hence
        | unfolding
        | | | | receive a message[from the agent stored in the cell bound to "Monitor"]
        | | | |                  [containning a Translated-Update-List]
        | | | then send a message [to the agent bound to "Loader"]
        | | |                      [containning the contents of the given message]
        | | | then receive a message [from the agent bound to "Loader"]
        | | |                        [containning a (Log-List, Errors-List)]
        | | | then
        | | | | | | check not(the given list#2 is empty-list)
        | | | | | and then Carry-Errors(the contents of the given message)
        | | | | or check(the given list#2 is empty-list)
        | | and then unfold
```

A ação *Integrate-Cooperative-Source* é executada para as fontes que são cooperativas, ou seja, as fontes que automaticamente enviam para o Monitor as atualizações ocorridas em seus dados, que são de interesse para o Data Warehouse.

Esta ação recebe a identificação do agente Monitor e aloca uma célula de memória chamada "Monitor" para armazená-la para posteriores utilizações.

Na segunda parte da ação (após o hence), dentro de um loop infinito, o Integrador aguarda a chegada de uma lista chamada Translated-Update-List, enviada pelo Monitor, contendo as atualizações ocorridas nos dados da fonte. A periodicidade com que essas atualizações são enviadas é programada na própria fonte de dados.

Quando uma lista Translated-Update-List é recebida, o Integrador a envia ao componente Carregador, que será responsável por carregar as atualizações dos dados das fontes para as tabelas da Base de Dados de Trabalho.

Então o Integrador aguarda o recebimento de duas listas enviadas pelo Carregador, indicando que o mesmo finalizou suas atividades:

- Log-List: arquivo de log contendo todas as operações realizadas no processo de carregamento de dados para as tabelas da Base de Dados de Trabalho.

- Errors-List: arquivo de erros ocorridos durante o processo de carregamento de dados para as tabelas da Base de Dados de Trabalho, sendo que esta poderá ser vazia quando não existirem erros.

Após o recebimento dessas listas, o Integrador verifica se a lista de erros Errors-List recebida não esta vazia e em caso afirmativo notifica o sistema da necessidade de realizar-se a correção de erros através do procedimento *Carry-Errors*. Então a ação volta ao início do loop para recomeçar o processo.

A ação *Carry-Errors* é dependente de implementação e, no Projeto SAGU, representa um procedimento não automatizado para a correção dos erros ocorridos no carregamento das atualizações das fontes para as tabelas da Base de Dados de Trabalho, através da verificação manual da lista de erros Errors-List e da lista de log Log-List, sendo que esse procedimento não é especificado aqui.

- Carry-Errors (_ , _) :: Errors-List, Log-List → action

(1) Carry-Errors (L_1, L_2) = □

A ação *Integrate-Non-Cooperative-Source* apresentada a seguir é semelhante à ação *Integrate-Cooperative-Source*, mas nela o Integrador é quem solicita a verificação da ocorrência de atualizações nas fontes de dados, pois as fontes não cooperativas não possuem características para realizar essa função automaticamente.

- Integrate-Non-Cooperative-Source :: action
  [receiving a (string, integer, integer) | binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage].
- Translated-Update-List = list of tuples.
- Log-List = list of tuples.
- Errors-List = list of tuples.

```
(6) Integrate-Non-Cooperative-Source =
        allocate a cell
      then
        bind the given cell to "Monitor"
        and store the given string#1 in it
    and
        allocate a cell
      then
        bind the given cell to "ABSN"
        and store the given integer#2 in it
    and
        allocate a cell
      then
        bind the given cell to "Wait-Time"
        and store the given integer#3 in it
  hence
    unfolding
        send a message [to the agent stored in the cell bound to "Monitor"]
                       [containning the integer stored in the cell bound to "ABSN"]
      then receive a message [from the agent stored in the cell bound to "Monitor"]
                             [containning a (ABSN, Translated-Update-List)]
      then store the given ABSN#1 in the cell bound to "ABSN"
        and
            check not(the given list#2 is empty-list)
          and then
            send a message [to the agent bound to "Loader"]
                           [containning the given list#2]
          then receive a message [from the agent bound to "Loader"]
                                 [containning a (Errors-List, Log-List)]
          then
              check not(the given list#1 is empty-list)
              and then Carry-Errors(the contents of the given message)
            or check(the given list#1 is empty-list)
        or check(the given list#2 is empty-list)
      and then Sleep "Wait-Time"
    and then unfold
```

A ação *Integrate-Non-Cooperative-Source* é executada para as fontes que não são cooperativas, isto é, não são capazes de automaticamente notificar o Data Warehouse, quando da ocorrência de atualizações, sendo que componentes internos do Data Warehouse devem realizar esta verificação.

Inicialmente esta ação recebe uma tupla de dados contendo a identificação do Monitor, o último ABSN verificado no arquivo de log e a periodicidade para verificação das atualizações nas fontes e armazena essas informações em células de memória chamadas de Monitor, ABSN e Wait-Time respectivamente, pois essas informações serão utilizadas posteriormente.

A segunda parte da ação (após o hence), é responsável por verificar periodicamente as atualizações ocorridas na fonte de dados. A periodicidade com que essas verificações são realizadas é determinada pela variável Wait-Time, cujo valor foi recebido no início da ação.

Dentro de um loop infinito, o Integrador envia uma mensagem para o Monitor, contendo o ultimo ABSN verificado no arquivo de log da fonte, a partir do qual serão pesquisadas as atualizações na fonte de dados.

Então o Integrador aguarda a chegada de uma mensagem do Monitor, contendo o último ABSN pesquisado no arquivo de log da fonte e uma lista chamada Translated-Update-List, contendo as atualizações ocorridas nos dados da fonte.

Em seguida, o ABSN recebido é armazenado na célula de memória chamada ABSN e se a lista Translated-Update-List não estiver vazia (indicando que existem atualizações nos dados da fonte), ela é enviada ao componente Carregador, responsável por carregar as atualizações para as tabelas da Base de Dados de Trabalho.

Na sequência, o Integrador aguarda o recebimento de duas listas enviadas pelo Carregador, indicando o encerramento de suas atividades:

- Log-List: lista de log contendo todas as operações realizadas no processo de carregamento de dados para as tabelas da Base de Dados de Trabalho.

- Errors-List: lista de erros ocorridos durante o processo de carregamento de dados para as tabelas da Base de Dados de Trabalho, sendo que esta poderá ser vazia quando não existirem erros.

Após o recebimento dessas listas, o Integrador verifica se a lista de erros Errors-List recebida não está vazia e em caso afirmativo notifica o sistema da necessidade de realização da correção de erros através da ação *Carry-Errors*, a mesma utilizada na ação *Integrate-Cooperative-Source*.

apresentada acima.

Em seguida, através da ação *Sleep*, o Integrador permanece em estado de espera, de acordo com o valor armazenado no produtor Wait-Time, antes de verificar novamente as atualizações da fonte de dados. Esta ação é a mesma utilizada pelo Integrador na ação *Integrator-Action* apresentada acima.

4.2 Componentes Responsáveis pela Verificação de Atualizações em Fontes de Dados Não Cooperativas

O processo de verificação de atualizações em uma fonte de dados não cooperativa, que trabalha com arquivos de log [EN94], é coordenado pelo Integrador que em períodos de tempo pré-determinados solicita ao Monitor correspondente a checagem das fontes. Então, o Monitor consulta o arquivo de log da fonte, e passa cada registro lido ao Tradutor. Os dados obtidos da fonte são então traduzidos para o formato interno utilizado no Warehouse.

O processo é finalizado quando não existirem mais registros a serem processados. Então todas as atualizações são enviadas ao Monitor em um novo formato. O Monitor, então envia os registros atualizados ao Integrador, para que sejam integrados ao Warehouse.

4.2.1 Componente Monitor

A seguir apresentamos a descrição formal do componente Monitor para fontes de dados não cooperativas.

- NCS-Monitor-Function :: action [receiving an agent | diverging].

(1) NCS-Monitor-Function =
    | NCS-Monitor-Init-Action
    hence
    | NCS-Monitor-Action

Quando o Monitor é criado (pelo Integrador, que é responsável pela inicialização do Data Warehouse e coordenação do processo) ele recebe a identificação de sua fonte de dados correspondente.

A inicialização de um agente Monitor que trabalha com fonte de dados não cooperativa, consiste da inicialização de outros dois módulos componentes: o Auditor e o Tradutor. Todos os *bindings* conhecidos pelo Monitor são passados para esses componentes. Esta ação é descrita abaixo:

- NCS-Monitor-Init-Action :: action
  [receiving an agent | binding | completing | communicating][using current bindings | current buffer].

(2) NCS-Monitor-Init-Action =
  | | subordinate an agent then bind it to "Auditor"
  and
  | | subordinate an agent then bind it to "Translator"
  hence
  | | send a message[to the agent bound to "Auditor"]
  | | [containning the application of
  | | closure abstraction of NCS-Auditor-Function to the given agent#1]
  and
  | | send a message[to the agent bound to "Translator"]
  | | [containning the closure abstraction of NCS-Translator-Function]

Na ação acima o Monitor recebe a identificação da sua fonte de dados correspondente e cria dois novos agentes (processos), representando o componente Auditor e o Tradutor e envia a ação que cada um deles irá executar.

A ação *NCS-Monitor-Action* é especificada abaixo:

- NCS-Monitor-Action :: action [diverging | communicating][using current bindings | current buffer].
- ABSN = integer.
- Translated-Update-List = list of tuples.

(3) NCS-Monitor-Action =

```
unfolding
| | | receive a message[from the agent bound to "Integrator"]
| | |                  [containning an ABSN]
| | then send a message[to the agent bound to "Auditor"]
| |                    [containning the contents of the given message]
| | then receive a message[from the agent bound to "Auditor"]
| |                       [containning a (ABSN, Translated-Update-List)]
| | then send a message[to the agent bound to "Integrator"]
| |                    [containning the contents of the given message]
| and then unfold
```

A ação *NCS-Monitor-Action* é um loop infinito em que o Monitor aguarda o recebimento de uma mensagem do Integrador. O conteúdo dessa mensagem deve ser um ABSN, que é um número inteiro sequencial, que indica o último bloco de registros verificado no arquivo de log da fonte.

Na sequência, o Monitor passa o ABSN recebido para o componente Auditor, e aguarda até que o mesmo envie o último ABSN verificado e a lista de atualizações encontradas na fonte, denominada Translated-Update-List. As informações recebidas são então enviadas para o Integrador, que procederá com a atualização do Warehouse.

Então a ação volta ao início do loop e aguarda por uma nova mensagem do Integrador para iniciar uma nova checagem de atualizações das fontes.

4.2.2 Componente Auditor

A especificação do módulo Auditor é apresentada a seguir:

- NCS-Auditor-Function :: action
  [receiving an agent | binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage].
- ABSN = integer.
- Translated-Update-List = list of tuples.

```
(4) NCS-Auditor-Function =
        allocate a cell
      then
        bind the given cell to "Source"
        and store the given agent#1 in it
    hence
      unfolding
        receive a message[from the agent bound to "Monitor"]
                         [containning an ABSN]
        then Query-Source(the contents of the given message)
      and then
        unfolding
          Receive-Next a message[from the agent stored in the cell bound to "Source"]
                                [containning a (ABSN, tuple)]
          then give the contents of the given message
          then
            send a message[to the agent bound to "Translator"]
                          [containning the rest of the given tuple]
            and
              check not(the rest of the given tuple is empty)
              and then unfold
            or
              check(the rest of the given tuple is empty)
              and then
                give the given ABSN#1
                and
                  receive a message[from the agent bound to "Translator"]
                                   [containning a Translated-Update-List]
                  then give the contents of the given message
                then send a message[to the agent bound to "Monitor"]
                                   [containning the given ABSN#1, the given list#2]
      and then unfold
```

Inicialmente o Auditor recebe a identificação de um agente correspondendo à identificação da fonte de dados e a armazena em uma célula de memória denominada Source, para posteriores utilizações.

Na outra parte da ação (após o hence), dentro de um loop infinito, o Auditor aguarda por uma mensagem do Monitor, contendo um ABSN, para então fazer uma consulta ao arquivo de log da fonte, através da ação *Query-Source*, obtendo todos os registros posteriores ao número do ABSN recebido.

Através de um novo loop, o Auditor recebe mensagens da fonte de dados, contendo um ABSN e uma tupla de dados correspondente. Esta tupla é então enviada ao Tradutor para ser traduzida. Esse loop acontece até que não existam mais registros do arquivo de log da fonte a serem processados, ou seja, até que a mensagem recebida contenha uma tupla de dados vazia.

Então, o Auditor aguarda até que o Tradutor envie a lista de atualizações traduzidas, denominada de lista Translated-Update-List e envia uma mensagem para o Monitor contendo o último ABSN verificado no arquivo de log da fonte e a lista Translated-Update-List.

A ação *Receive-Next* é especificada a seguir:

- Receive-Next _ :: yielder → action [completing][using current buffer | giving a message].

(1) Receive-Next (Y |= yielder) =
    patiently
        choose a Y [in set of(head(Sort-ABSN(the current buffer)))]
    then
        remove the given message and give it

Aonde *Sort-ABSN* é uma função sobre a lista de mensagens encontradas no buffer do Auditor, que organiza esta lista em ordem ascendente de ABSN, resultando na escolha do primeiro valor nela encontrado. Esta função é dependente de implementação.

- Sort-ABSN _ :: buffer → action [completing][giving a buffer]

(1) Sort-ABSN (A) = □

A ação *Query-Source* é responsável por consultar o arquivo de log da fonte. Esta ação é dependente de implementação, e não é especificada aqui.

- Query-Source _ :: ABSN → Action

(1) Query-Source A = □

4.2.3 Componente Tradutor

O componente Tradutor, definido a seguir, aguarda por uma mensagem enviada pelo Auditor, para iniciar o processo de tradução dos dados. As atualizações traduzidas são armazenadas em uma lista.

- NCS-Translator-Function :: action [binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage].

```
(5) NCS-Translator-Function =
        | | allocate a cell
        | then bind the given cell to "Translated-Update-List"
        hence
        | unfolding
        | | | | store the empty-list in the cell bound to "Translated-Update-List"
        | | | and then
        | | | unfolding
        | | | | | receive a message[from the agent bound to "Auditor"]
        | | | | |                  [containning a tuple]
        | | | then
        | | | | | | check not(the given message is empty)
        | | | | and then
        | | | | | | | check(the given string#2 is in set["DSC" | "DSD" | "DSM"])
        | | | | | | and then
        | | | | | | | | Translate the given tuple
        | | | | | | | then give list of(the given tuple,
        | | | | | | |                   branches of the list stored in the cell bound
        | | | | | | |                   to "Translated-Update-list")
        | | | | | | | then store it in the cell bound to "Translated-Update-List"
        | | | | | | | then unfold
        | | | | | | or
        | | | | | | | | check not(the given string#2 is in set["DSC" | "DSD" | "DSM"])
        | | | | | | | and then unfold
        | | | | | or
        | | | | | | | check(the given message is empty)
        | | | | | and then send a message [to the agent bound to "Auditor"]
        | | | | |                         [containning the list stored in cell bound
        | | | | |                         to "Translated-Update-List"]
        | | and then unfold
```

Na primeira parte, a ação *NCS-Translator-Action* aloca uma célula de memória, chamada Translated-Update-List, para armazenar a lista das atualizações encontradas na fonte de dados.

Na segunda parte da ação (após o hence), dentro de um loop infinito é feita a inicialização da lista Translated-Update-List como vazia. Na sequência, em um novo loop o Tradutor aguarda a chegada de uma tupla de dados enviada pelo Auditor. Essa tupla de dados representa um registro lido diretamente do arquivo de log da fonte e possuiu um formato padrão para todos os arquivos existentes na fonte:

- O primeiro dado da tupla é referente ao nome do arquivo ao qual o registro pertence.
- O segundo dado da tupla é referente ao tipo da operação realizada. As únicas operações que indicam atualizações na base de dados da fonte e que conseqüêntemente devem ser traduzidas, são referentes às siglas "DSC"- inserção, "DSD"- deleção ou "DSM" - alteração da base de dados.
- Os próximos dados variam de acordo com os arquivos de dados existentes na fonte e a natureza das operações realizadas.

Após receber a mensagem do Auditor, o Tradutor verifica se a mesma não é vazia e em caso afirmativo, a função *Translate* deve ser chamada para os registros referentes à inserção ("DSC"), deleção ("DSD") ou alteração ("DSM") da base de dados. São ignorados outros registros contendo outros tipos de operações. Cada registro traduzido é adicionado à lista de atualizações Translated-Update-List.

A finalização do processo é sinalizada pelo recebimento de uma mensagem vazia, indicando que não existem mais registros a serem traduzidos. A lista de atualizações é então enviada ao Auditor e o Tradutor novamente inicializa a lista de atualizações e aguarda a chegada de uma nova tupla de dados para começar um novo processo.

A ação *Translate* é dependente de implementação, estando fora do escopo deste trabalho. No caso do Projeto SAGU, para uma fonte de dados hierárquica, a ação *Translate* recebe uma sequencia de bits em formato EBCDIC e a traduz para o formato ASCII.

- Translate _ :: tuple → action [completing][giving a tuple]

(1) Translate $(T) = \square$

4.3 Componentes Responsáveis pela Verificação de Atualizações em Fontes de Dados Cooperativas

As fontes de dados cooperativas apresentam características específicas que permitem a comunicação da fonte com componentes internos do Data Warehouse, possibilitando que as atualizações existentes sejam enviadas para o Monitor, com periodicidade e outros controles programados na própria fonte de dados.

No caso das fontes que não trabalham com um formato aceito pelo Warehouse, as atualizações devem ser traduzidas por um agente Tradutor, antes de serem integradas ao Data Warehouse.

4.3.1 Componente Monitor

A seguir apresentamos a descrição formal do componente Monitor para fontes de dados cooperativas.

- CS-Monitor-Function :: action [receiving an agent | diverging].

(1) CS-Monitor-Function =
    | CS-Monitor-Init-Action
    hence
    | CS-Monitor-Action

Quando o Monitor é criado (pelo Integrador, que é responsável pela inicialização do Data Warehouse e coordenação do processo) ele recebe a identificação de sua fonte de dados correspondente.

A inicialização de um agente Monitor que trabalha com uma fonte de dados cooperativa, consiste da inicialização do módulo Tradutor. Todos os *bindings* conhecidos pelo Monitor são passados ao Tradutor. Esta ação é descrita abaixo:

- CS-Monitor-Init-Action :: action
  [binding | commpleting | communicating][using current bindings | current buffer].

```
(2)  CS-Monitor-Init-Action =
       | subordinate an agent then bind it to "Translator"
     hence
       | send a message[to the agent bound to "Translator"]
       |                [containning the closure abstraction of CS-Translator-Function]
```

Na ação acima o Monitor cria um novo agente (processo), representando o componente Tradutor e envia a ação que o mesmo irá executar.

A ação *CS-Monitor-Action* é especificada abaixo:

- CS-Monitor-Action :: action [receiving an agent | binding | storing | diverging | communicating] [using current bindings | current buffer | current storage].
- Update-List = list of tuples.
- Translated-Update-List = list of tuples.

```
(3)  CS-Monitor-Action =
       | | allocate a cell
       | then
       | | | bind the given cell to "Source"
       | | | and store the given agent#1 in it
     hence
       | unfolding
       | | | | receive a message[from the agent bound to "Source"]
       | | | |                  [containning an Update-List]
       | | | then send a message[to the agent bound to "Translator"]
       | | |                    [containning the contents of the given message]
       | | | then receive a message[from the agent bound to "Translator"]
       | | |                      [containning an Translated-Update-List]
       | | | then send a message[to the agent bound to "Integrator"]
       | | |                    [containning the contents of the given message]
       | | | and then unfold
```

Inicialmente a ação *CS-Monitor-Action* recebe a identificação da fonte de dados correspondente, e armazena essa identificação em uma célula de memória chamada Source, para posteriores utilizações.

Na segunda parte da ação (após o hence), dentro de um loop infinito o Monitor aguarda o recebimento de uma mensagem da fonte. O conteúdo dessa mensagem deve ser uma lista contendo as atualizações ocorridas nos dados da fonte, chamada Update-List.

Na sequência, o Monitor passa a lista Update-List recebida, para o componente Tradutor, para realizar as traduções necessárias e aguarda até que o mesmo envie a lista traduzida denominada Translated-Update-List. As informações recebidas são então enviadas ao Integrador, que procederá com a atualização do Warehouse, deixando-o em um estado consistente com o estado da fonte.

O Monitor volta ao início do loop e aguarda a chegada de novas atualizações para reiniciar o processo.

4.3.2 Componente Tradutor

O componente Tradutor, definido a seguir, aguarda por uma mensagem enviada pelo Monitor, para iniciar o processo de tradução das informações. As atualizações traduzidas são armazenadas em uma lista.

- CS-Translator-Function :: action [diverging | communicating][using current buffer].
- Update-List = list of tuples.

```
(4) CS-Translator-Function =
        unfolding
        | | receive a message[from the agent bound to "Monitor"]
        | |                    [containning an Update-List]
        | | then Translate-List(the given list)
        | | then send a message [to the agent bound to "Monitor"]
        | |                     [containning the given list]
        | and then unfold
```

A ação *CS-Translator-Action* é um loop infinito no qual o Tradutor aguarda a chegada de uma lista contendo as atualizações enviadas pelo Monitor, denominada Update-List.

Após receber a mensagem do Monitor, a função *Translate* é responsável por realizar a tradução dos dados em um formato aceito pelo Warehouse.

A lista de atualizações agora traduzida é enviada ao Monitor e o Tradutor volta ao início do loop e aguarda a chegada de uma nova lista de atualizações para iniciar um novo processo.

A ação *Translate* é dependente de implementação, estando fora do escopo deste trabalho. No caso do Projeto SAGU, ela geralmente recebe uma sequencia de bits em qualquer formato e a traduz para o formato ASCII.

- Translate-List _ :: list → action [completing][giving a list]

(1) Translate-List $(T) = \square$

4.4 Componente Carregador

O componente Carregador é responsável por realizar o carregamento das atualizações encontradas nas fontes de dados. Essas atualizações são enviadas pelo Integrador, geralmente em formato ASCII e devem ser carregadas para as tabelas da Base de Dados de Trabalho no formato utilizado pelo Warehouse. A especificação do Carregador é apresentada a seguir:

- Loader-Function :: action [diverging].

(1) Loader-Function =
    | Loader-Init-Action
    hence
    | Loader-Action

A ação *Loader-Init-Action* é especificada abaixo:

- Loader-Init-Action :: action [binding | completing][using current bindings].
- Log-List = list of tuples.
- Errors-List = list of tuples.

(2) Loader-Init-Action =
    | | | allocate a cell
    | | then
    | | | bind the given cell to "Log-List"
    | and
    | | | allocate a cell
    | | then
    | | | bind the given cell to "Errors-List"

Na ação *Loader-Init-Action*, o Carregador aloca duas células de memória: uma para armazenar a lista de log das operações realizadas, denominada Log-List e outra para armazenar a lista dos possíveis erros ocorridos, chamada de Erros-List. Esses bindings são passados à ação *Loader-Action* a seguir especificada, através do combinador hence.

- Loader-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) Loader-Action =
```
unfolding
| | | store the empty-list in the cell bound to "Log-List"
| | and
| | | store the empty-list in the cell bound to "Errors-List"
| then receive a message [from the agent bound to "Integrator"]
|                        [containning a Translated-Update-List]
| then
| | | | send a message [to the agent bound to "SemaphoreWDB"]
| | | |                [containning a "Wait"]
| | | and then receive a message [from the agent bound to "SemaphoreWDB"]
| | |                            [containning an "OK"]
| | and then Carry-Work-Database(the contents of the given message)
| then send a message [to the agent bound to "SemaphoreWDB"]
|                     [containning a "Signal"]
| then send a message [to the agent bound to "Integrator"]
|                     [containning a (list stored in the cell bound to "Errors-List",
|                     list stored in the cell bound to "Log-List")]
| then unfold
```

A ação acima é um loop infinito, no qual o Carregador inicializa as listas Log-List e Errors-List e então aguarda o recebimento da lista de atualizações da fonte de dados, chamada Translated-Update-List.

Em seguida, o Carregador envia uma mensagem ao processo Semáforo da Base de Dados de Trabalho - apresentado à frente, responsável por controlar a utilização da Base de Dados de Trabalho - contendo a string "Wait" para sinalizar a utilização da Base de Dados de Trabalho para o carregamento de dados, não permitindo que nenhum outro processo utilize a mesma.

A ação aguarda o retorno de uma mensagem do Semáforo da Base de Dados de Trabalho, concedendo a liberação para que o Carregador possa executar o procedimento *Carry-Work-*

Database, responsável pelo carregamento das atualizações dos dados para as tabelas da Base de Dados de Trabalho.

Após a finalização deste procedimento, o Carregador envia uma mensagem para o Semáforo da Base de Dados de Trabalho, contendo a string "Signal" para notificar a liberação da Base de Dados de Trabalho para outros processos.

Na sequência o Carregador envia uma mensagem para o Integrador, contendo a lista de erros Erros-List (que estará vazia no caso da não ocorrência de erros durante esse processo) e a lista de log das operações realizadas Log-List. Então o Carregador volta ao início do loop reiniciar o processo.

A ação *Carry-Work-Database* é dependente de implementação, não sendo especificada aqui, e no Projeto SAGU, representa um procedimento do Oracle[1], utilizado para o carregamento de informações em formato ASCII para as tabelas da Base de Dados de Trabalho, utilizando informações adicionais existentes no Depósito de Metadados.

- Carry-Work-Database :: action [completing][giving a (Errors-List,Log-List)]

(1) Carry-Work-Database = □

4.5 Componente Consolidador

O agente Consolidador é responsável por fazer a integração das atualizações carregadas na Base de Dados de Trabalho à Base de Dados Espelho, resolvendo as possíveis inconsistências entre os dados e verificando os relacionamentos entre múltiplas fontes, para integrar as atualizações recebidas com os dados existente no Warehouse.

A Base de Dados Espelho está armazenada em um Sistema Gerenciador de Base de Dados Oracle e seus dados estão modelados dimensionalmente [RM99].

Esta base de dados está projetada para otimizar o processo de atualização dos dados, de tal forma que existem vários índices criados para garantir a integridade referencial dos dados e

[1] http://www.oracle.com

para facilitar a resolução das inconsistências entre os dados provenientes de diferentes fontes.

A Base de Dados Espelho contém os mesmos dados que a Base de Produção, mas como o processo de consolidação e atualização é bastante demorado, todas as atualizações são feitas na Base de Dados Espelho para que a Base de Dados de Produção não tenha que ficar *off-line* durante este processo.

- Consolidator-Function :: action [diverging].

(1) Consolidator-Function =
    | Consolidator-Init-Action
    hence
    | Consolidator-Action

A ação *Consolidator-Init-Action* é especificada abaixo:

- Consolidator-Init-Action :: action [binding | completing][using current bindings].

(2) Consolidator-Init-Action =
    | | allocate a cell
    | then
    | | bind the given cell to "Consolidator.Log"

Na ação *Consolidator-Init-Action*, o Consolidador aloca uma célula de memória (para armazenar a lista de log das operações realizadas) denominada Consolidator.Log, passando esse binding à ação *Consolidator-Action*, a seguir especificada.

- Consolidator-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) Consolidator-Action =

```
unfolding
| | store the empty-list in the cell bound to "Consolidator.Log"
| then receive a message [from the agent bound to "Integrator"]
|                        [containning an "OK"]
| then send a message [to the agent bound to "SemaphoreWDB"]
|                     [containning a "Wait"]
| then receive a message [from the agent bound to "SemaphoreWDB"]
|                        [containning an "OK"]
| then Integrate-Mirror-Database
| then send a message [to the agent bound to "SemaphoreWDB"]
|                     [containning a "Signal"]
| then send a message [to the agent bound to "Integrator"]
|                     [containning the list stored in the cell bound to "Consolidator.Log"]
| then unfold
```

A ação acima é um loop infinito, no qual o Consolidador inicializa como vazia a lista de log das operações realizadas, denominada **Consolidator.Log** e então aguarda o recebimento de uma mensagem do Integrador para iniciar o processo.

Em seguida, o Consolidador envia uma mensagem ao processo Semáforo da Base de Dados de Trabalho, contendo a string "**Wait**", para solicitar a liberação da Base de Dados de Trabalho, não permitindo que novas atualizações de dados sejam carregadas na mesma, durante o processo de consolidação da Base de Dados Espelho.

A ação aguarda o retorno do Semáforo da Base de Dados de Trabalho, concedendo a liberação para o Consolidador executar o procedimento *Integrate-Mirror-Database*, responsável pela integração das atualizações existentes na Base de Dados de Trabalho com os dados existente na Base de Dados Espelho do Warehouse.

Como os dados são originários de diferentes fontes, na ação *Integrate-Data-Mirror* também são resolvidas as inconsistências entre eles, através de operações de reformatação de dados, modificação de estruturas chaves dos dados, sumarização de dados e integração de dados de múltiplas fontes, com o intuito de prover uma visão unificada de todos os dados da organização, antes de serem armazenados no Warehouse.

Após a finalização deste procedimento, o Consolidador envia uma mensagem para o Semáforo da Base de Dados de Trabalho, contendo a string "**Signal**", para notificar a liberação da Base de Dados de Trabalho para outros processos.

Na sequência, o Carregador envia uma mensagem para o Integrador para notificar a finalização do processo, contendo a lista de log das operações realizadas Consolidator.Log. Então o Consolidador volta ao início do loop e inicializa a lista Consolidator.Log como vazia e aguarda até a chegada de uma nova mensagem do Integrado.

A ação *Integrate-Mirror-Database* é responsável por atualizar a Base de Dados Espelho do Warehouse, carregando as alterações ocorridas nas fontes de dados e resolvendo as possíveis inconsistências entre os dados da organização.

Para realizar essas transformações, o Consolidador consulta o Depósito de Metadados, que contém informações sobre os dados existentes no Warehouse, sobre os dados provenientes das fontes e sobre os mapeamentos entre os dados.

Esta ação é dependente de implementação, não sendo especificada aqui. No projeto SAGU, esta atualização é feita utilizando-se pacotes prontos do Oracle e também rotinas especificamente desenvolvidas de acordo com o contexto do projeto.

- Integrate-Mirror-Database :: action [completing][giving a Consolidator.Log]

(1) Integrate-Mirror-Database = □

4.6 Componente Atualizador

O agente Atualizador é responsável por carregar os dados da Base de Dados Espelho à Base de Dados de Produção, deixando o Warehouse em um estado consistente com o estado das fontes de dados.

A Base de Dados de Produção esta armazenada em um Sistema Gerenciador de Base de Dados Oracle, e está projetada visando a facilidade de utilização do usuário final, contendo índices projetados especificamente para consultas.

A modelagem dimensional de dados foi feita utilizando um misto dos modelos *Star* e *Snowflake* [RM99], que são exemplos de modelos dimensionais. O modelo *Star* consiste de uma única tabela de fatos, formando o centro da estrela, ligada a várias tabelas de dimensões

que formam os pontos da estrela, estas tabelas não são normalizadas e possuem todas as hierarquias de dados incorporadas. O modelo *Snowflake* é semelhante ao modelo *Star*, com a diferença de que as tabelas de dimensões somente incorporam as hierarquias que não são compartilhadas por mais de uma dimensão.

Devido às características da modelagem dimensional, as tabelas de dados do *warehouse* não são normalizadas e são adicionadas hierarquias e sumarizações aos dados, sendo de responsabilidade do componente Atualizador a realização das operações para realizar essas transformações de dados.

- Updater-Function :: action [diverging].

(1) Updater-Function =
    | Updater-Init-Action
    hence
    | Updater-Action

A ação *Updater-Init-Action* é apresentada abaixo:

- Updater-Init-Action :: action [binding | completing][using current bindings].

(2) Updater-Init-Action =
    | | allocate a cell
    | then
    | | bind the given cell to "Updater.Log"

Na ação *Updater-Init-Action*, o Atualizador aloca uma célula de memória para armazenar a lista de log das operações realizadas, denominada **Updater.Log**, passando esse binding à ação *Updater-Action* (especificada a seguir) através do combinador **hence**.

- Updater-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) Updater-Action =

```
unfolding
| | store the empty-list in the cell bound to "Updater.Log"
| then receive a message [from the agent bound to "Integrator"]
|                        [containning an "OK"]
| then Update-Production-Database
| then send a message [to the agent bound to "Integrator"]
|                     [containning the list stored in the cell bound to "Updater.Log"]
| then unfold
```

A ação acima é um loop infinito, no qual o Atualizador inicializa a lista de log das operações realizadas, denominada Updater.Log e então aguarda o recebimento de uma mensagem do Integrador para iniciar o processo de atualização da Base de Dados de Produção, através da ação *Update-Production-Database*.

Na sequência o Atualizador envia uma mensagem para o Integrador para notificar a finalização do processo, contendo a lista de log das operações realizadas Updater.Log. Então o Atualizador volta ao início do loop, inicializa a lista Updater.Log e aguarda até a chegada de uma nova mensagem do Integrador para iniciar o processo novamente.

A ação *Update-Production-Database* é responsável por carregar os dados da Base de Dados Espelho para a Base de Dados de Produção. Para agilizar este procedimento, somente as tabelas que contiverem atualizações em seus dados serão copiadas.

Esta ação é dependente de implementação, não sendo especificada aqui. No projeto SAGU, este procedimento é realizado utilizando-se pacotes prontos do Oracle e também rotinas especificamente desenvolvidas de acordo com o contexto do projeto.

- Update-Production-Database :: action [completing][giving an Updater-Log]

(1) Update-Production-Database = □

4.7 Componente Semáforo da Base de Dados de Trabalho

O agente Semáforo da Base de Dados de Trabalho é responsável por coordenar a utilização da Base de Dados de Trabalho, para evitar que sejam realizados carregamentos de atualizações

das fontes de dados pelo Carregador, quando o Consolidador estiver realizando a consolidação das atualizações carregadas.

Essa sincronização de processos, é utilizada para não permitir que ocorram anomalias no processo de manutenção da base de dados.

- SemaphoreWDB-Function :: action [diverging].

(1) SemaphoreDB-Function =
| SemaphoreDB-Init-Action
hence
| SemaphoreDB-Action

A ação *SemaphoreWDB-Init-Action* é especificada abaixo:

- SemaphoreWDB-Init-Action :: action [storing | completing][using current bindings | current storage].

(2) SemaphoreWDB-Init-Action =
| | allocate a cell
| then
| | bind the given cell to "User"
and
| | allocate a cell
| then
| | | bind the given cell to "Value"
| | and then
| | | store 0 in it

Na ação *SemaphoreWDB-Init-Action*, o processo aloca uma célula de memória denominada User, para armazenar a identificação do agente que atualmente está utilizando a Base de Dados de Trabalho e outra célula denominada Value, para armazenar o *status* desta base de dados, podendo conter os seguintes valores: 1 - indicando que a base não está disponível e 0 - indicando que a base está disponível para utilização. Na ação *SemaphoreWDB* é atribuído o valor 0 para a célula Value para inicializá-la.

Todos os *bindings* da ação *SemaphoreWDB-Init-Action* são passados à ação *SemaphoreDB-Action*, a seguir especificada.

- SemaphoreWDB-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) SemaphoreWDB-Action =

```
unfolding
| | | check(the value stored in the cell bound to "Value" is 1)
| | and then
| | | | receive a message[from the agent stored in the cell bound to "User"]
| | | |                  [containning a "Signal"]
| | | then store 0 in the cell bound to "Value"
| | | then unfold
| or
| | | check(the value stored in the cell bound to "Value" is 0)
| | and then
| | | | receive a message[from any agent][containning a "Wait"]
| | | then
| | | | | | store the sender of the given message in the cell bound to "User"
| | | | | and
| | | | | | store 1 in the cell bound to "Value"
| | | | and then
| | | | | send a message[to the agent stored in the cell bound to "User"]
| | | | |                [containning an "OK"]
| | | | then unfold
```

A ação acima está implementada utilizando conceitos baseados em Semáforos de Sistemas Operacionais, responsáveis por coordenar a utilização de recursos disponíveis [Tan95].

Dentro de um loop infinito, a ação verifica se o valor existente na célula de memória Value é igual ao valor 1 ou 0.

Se o valor armazenado na variável "Value" for igual a 1, significa que a Base de Dados de Trabalho está sendo utilizada por algum processo, no caso do Projeto SAGU, por um dos seguintes agentes: Carregador ou Consolidador. Na sequência a ação aguarda o recebimento de uma mensagem do processo que está utilizando a base de dados, contendo a string "Signal", indicando a liberação da Base de Dados de Trabalho e então, armazena o valor 0 na célula de memória Value.

Se o valor armazenado na variável Value for igual a 0, significa que a Base de Dados de Trabalho não está sendo utilizada por nenhum processo, então a ação aguarda o recebimento de uma mensagem do agente Carregador ou do agente Consolidador. Com a chegada da mensagem, o Semáforo da Base de Dados de Trabalho armazena o rementente da mensagem na célula de memória chamada User, armazena o valor 1 na célula de memória chamada Value e envia uma mensagem para o agente remetente liberando a utilização da Base de Dados de Trabalho.

4.8 Considerações Finais

No apêndice A é apresentada a especificação formal dos componentes do *data warehouse* do Projeto SAGU - definida neste capítulo - sem os comentários informais presentes neste capítulo.

Capítulo 5

Conclusões

A utilização de *data warehouses* está sendo considerada uma solução adequada para os problemas relacionados à construção de Sistemas de Suporte à Decisão.

Por se tratar de uma tecnologia relativamente recente, ainda existem questões referentes a *data warehousing* que devem ser resolvidas, não constituindo um concenso.

Algumas arquiteturas de *data warehouse* foram definidas [Dev97, WGL96, LZW97, HAM95, ZGH95], mas essas definições não são formais. Este trabalho visa estabelecer uma descrição formal dos componentes da arquitetura do *data warehouse* do projeto SAGU, como forma de chamar a atenção a detalhes que muitas vezes são negligenciados durante o projeto de um *data warehouse*.

O principal objetivo do nosso trabalho é possibilitar um melhor entendimento do processo de *data warehousing* como um todo, contribuindo para o processo de padronização desta tecnologia. Para isto, usamos o formalismo dado pela Semântica de Ações na especificação dos módulos componentes do *data warehouse*, visando a documentação dos mesmos.

No início do desenvolvimento do Projeto SAGU, participamos da escolha e da definição da arquitetura do *data warehouse*. Dentre várias abordagens estudadas [Dev97, HAM95, ZGH95], tomamos por base a arquitetura utilizada no Protótipo WHIPS [WGL96, LZW97, HAM95], por se tratar de uma proposta genérica e bastante aceita na literatura.

Os objetivos do *data warehouse* do Projeto SAGU são semelhantes aos do Protótipo WHIPS, porém foram adaptados à realidade da UFPR.

A especificação formal foi de extrema importância na concepção da arquitetura do Projeto SAGU, sendo feita simultâneamente à definição da mesma, proporcionando *feedback* para todas as partes envolvidas, antecipando problemas que possivelmente poderiam surgir e agilizando o desenvolvimento de algumas operações, por prover um material concreto de apoio.

A escolha do formalismo de Semântica de Ações para realizar as especificações dos módulos de software do *data warehouse* do Projeto SAGU, deve-se ao seu alto grau de modularidade, estensibilidade, reusabilidade, que são especialmente desejáveis quando utilizamos descrições semânticas durante projetos de software [MM94].

Outra característica importante que influenciou na escolha do formalismo de Semântica de Ações, é que as ações são escritas como frases em Inglês (mas completamente formais), facilitando sua leitura e entendimento (pelo menos superficial) por pessoas leigas no assunto.

A utilização de métodos formais para realizar a descrição de sistemas de software é explicável pela necessidade de se construir sistemas que ofereçam propriedades como correção, confiabilidade e segurança. Os métodos formais oferecem a vantagem de permitir a verificação formal dessas propriedades, embora nem sempre isso seja fácil.

Como os módulos componentes do Projeto SAGU são relativamente pequenos, conseguimos descrever com relativa simplicidade as propriedades de suas especificações, mas acreditamos ser imprescindível a utilização de ferramentas automatizadas, para sistemas grandes e complexos, sem as quais seria extremamente difícil a sua utilização.

Outra grande vantagem da utilização de métodos formais para especificação dos módulos componentes do sistema da *data warehouse* do Projeto SAGU, é que elas puderam ser usadas como meio de comunicação entre os participantes do projeto, aonde todos concordaram sobre o seu significado, devido ao fato de apresentarem uma semântica não ambígua e totalmente precisa. Assim como o fato da linguagem utilizada nas especificações formais ser bem definida levou a um rigor maior nas definições, facilitando o aparecimento de comportamentos errôneos de módulos do sistema especificado.

Neste trabalho, quando apresentamos a especificação de cada módulo componente, utilizamos comentários informais, de modo que os pontos mais importantes foram devidamente enfatizados. Desta forma, acreditamos que o entendimento das características das especificações fica mais rápido e mais seguro.

Na especificação formal dos Monitores para fontes de dados cooperativas e para fontes de dados não cooperativas, contamos com a colaboração de Alessandro Zamboni, aluno da graduação do curso de Informática da UFPR, e membro participante da equipe do Projeto SAGU.

Dentre as dificuldades encontradas, identificamos a especificação de tempo em Semântica de Ações. Esta característica do formalismo deve ser mudada em futuras versões do mesmo[1].

5.1 Trabalhos Futuros

Entre os trabalhos futuros que podem ser realizados podemos citar o desenvolvimento de mais modelos de Monitores para trabalhar com outros tipos de fontes, como por exemplo fontes que trabalhem com *Snapshots.*

Outro trabalho seria referente à especificação dos componentes de análise e consulta, que não foi realizado, devido ao fato de se tratarem de ferramentas prontas.

Também é necessário realizar a validação da especificação aqui realizada, usando algum método automático, de forma a provar propriedades.

Por fim, existe a possibilidade de implementar a geração automática de código implementável para Monitores e outros componentes, a partir desta especificação formal feita usando Semântica de Ações.

[1] http://www.brics.dk/projects/AS.

Bibliografia

[AGS97] R. Agrawal, A. Gupta, and S. Sarawagi. *Modeling Multidimensional Databases.* Research Report, IBM Almaden Research Center, San Jose, CA, 1997.

[AZ97] P. Adriaans and D. Zantinge. *Data Mining.* Addison-Wesley, England, first edition, 1997.

[BMW92] D. F. Brown, H. Moura, and D. A. Watt. *ACTRESS: an action semantics directed compiler generator.* Technical report, Departamental Research Report FM-1992-1, University of Glasgow, Departament of Computing Science, June 1992.

[Bro89] J. Glenn Brookshear. *Theory of Computation: Formal Languages, Automata and Complexity.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1989.

[BS97] A. Berson and S.J. Smith. *Data Warehouse, Data Mining and OLAP.* McGraw-Hill, first edition, 1997.

[CCS93] E. F. Codd, S.B. Codd, and C.T. Salley. *Providing OLAP (on-line Analytical Processing) to User-Analysts: An it Mandate.* Technical report, 1993. Available at http://www.arborsoft.com/essbase/wht-ppr/coddps.zip.

[CD97] S. Chaudhuri and U. Dayal. *An Overview of Data Warehousing and OLAP Technology.* In Proceedings of the ACM SIGMOD International Conference, 26(1):65-74, 1997.

[CW95] S. Ceri and J. Widom. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, California, 1995.

[Dev97] B. Devlin. *Data Warehouse from Arquitecture to Implementation.* Addison-Wesley, 1997.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database System.* Addison-Wesley, Redwood City, CA, 1994.

[GM98] A. Gupta and I.Mumick. *Materialized Views.* MIT Press, 1998.

[Gup97] H. Gupta. *Selection of Views to Materialize in a Data Warehouse.* In Proceedings of the International Conference on Database Theory, Delphi, Guece, January 1997.

[GZW97] H. Garcia-Molina, Y. Zhuge, and J. Wiener. *Consistency algorithms for multi-source warehouse view maintenance.* Technical report, Stanford University, 1997. Available at http://db.stanford.edu/pub/papers/strobe.ps.

[HAM95] J. Hammer, H. Garcia-Molina, J. Widom, et al. *The Stanford Data Warehousing Project.* IEEE Data Engineering Bulletin, June 1995.

[IWG99] W. H. Inmon, J. D. Welch, and K. L. Glassey. *Gerenciando Data Warehouse.* Makron Books, São Paulo, 1999.

[Kim96] R. Kimbal. *The Data Warehouse Toolkit.* John Wiley & Sons, 1996.

[Lee89] P. Lee. *Realistic Compiler Generation.* Fundation of Computing Series. The MIT Press, 1989.

[LHM97] B. Lindsay, L. M. Haas, C. Mohan, et al. Wilms. *A snapshot differential refresh algorithm.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1986.

[LZW97] W. J. Labio, Y. Zhuge, J. L. Wiener, et al. *The WHIPS Prototype for Data Warehouse Creation and Maintenance.* In Proceedings of the ACM SIGMOD Conference, Tuscon, Arizona, May 1997.

[Men98] L. C. S. Menezes. *Uso de orientação a objetos na prototipação de Semântica de Ações.* Master's thesis, Departament of Informatics, Federal University of Pernambuco, 1998.

[MM94] P. D. Mosses and M. A. Musicante. *An Action Semantics for ML Concurrency Primitives.* In FME'94: Industrial Benefit of Formal Methods, Second International Symposium of Formal Methods Europe, Barcelona, Spain, October 1994.

[Mos89] P. D. Mosses. Unified algebras and action semantics. In *STACS'89, Proc. Symp. on Theoretical Aspects of Computer Science, Paderborn,* volume 349 of Lecture Notes in Computer Science, Springer-Velag, 1989.

[Mos92] P.D. Mosses. *Action Semantics.* Cambridge University Press, Cambridge, UK, 1992.

[MR98] M. Meja-Lavalle and C. Rodrigues-Ortiz. *Obtaining Expert System Rules Using Data Mining Tools from a Power Generation Database.* Expert System with Application, 14:32-47, 1998.

[Mug97] S. Muggleton. *Declarative Knowledge Discovery in Industrial Databases.* In PADD'97: Proc. of First International Conference and Exhibition on the Practical Application of Knowledge Discovery and Data Mining, 1997.

[Mus96] M. A. Musicante. *On the Relational Semantics of Interleaving Constructors.* PhD thesis, UFPE, Dept of Computer Science, 1996.

[Oli98] A. G. Oliveria. *Data Warehouse: Conceitos e Soluções.* Advanced Editora, Florianópolis, SC, 1998.

[OMG95] Object Management Group (OMG), Framinghan, MA. *The Common Object Request Broker: Architecture and Specification.* July 1995.

[Poz00] A. T. Pozo, et al. *SAGU - Sistema de Apoio ao Gerenciamento Universitário* Relatório Técnico do Projeto, DINF - Departamento de Informática, UFPR, agosto 2000.

[RM99] M. A. Rkortink and D. L. Moody. *From Entities to Stars, Snowflakes, Clusters, Constellations and Galaxies: A Methodology for Data Warehouse Design.* In the 18th International Conference on Conceptual Modeling, Industrial Track Proceedings, Paris, France, November 1999.

[Sch86] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.

[Squ95] C. Squire. *Data Extraction and Transformation for the Data Warehouse.* In Proceedings of the ACM SIGMOD Annual Conference, San Francisco, CA, May 1995.

[Tan95] A. S. Tanenbaum. *Sistemas Operacionais Modernos.* Makron Books, São Paulo, 1995.

[YD96] Z. Yang and K. Duddy. *CORBA: A Platform for Distributed Object Computing. Operating Systems Review.* 30(2):4-31, April 1996.

[Wat91] D.A. Watt. *Programming Language Syntax and Semantics.* Prentice Hall, UK, 1991.

[Wei92] G. Wiederhold. *Mediators in the Architecture of Future Information System.* IEEE Computer, 25(3):38-49, March 1992.

[Wid95] J. Widom. *Research Problems in Data Warehousing.* In Proceedings of 4th Int'L Conference on Information and Knowledge Management (CIKM), November 1995.

[WGL96] J. L. Wiener, H. Gupta, W. J. Labio, et al. *A System Prototype for Warehouse View Maintenance.* In Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications, pages 26-33, Montreal, Canada, June 1996.

[WZG97] J. L. Wiener, Y. Zhuge, and H. Garcia-Molina. *Multiple view consistency for data warehousing.* In ICDE, Birmingham, UK, April 1997. Available at http://db.stanford.edu/pub/papers/mvc.ps.

[ZGH95] Y. Zhuge, H. Garcia-Molina, J. Hammer, et al. *View maintenance in a warehousing enviroment.* In SIGMOD, pages 316-327, San Jose, California, May 1995. Available at http://db.stanford.edu/pub/papers/anomaly-full.ps.

[ZGW95] Y. Zhuge, H. Garcia-Molina, J. Widom, et al. *View maintenance in a warehousing enviroment.* In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 316-327, San Jose, California, May 1995.

[ZGW97] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. *Consistency algorithms for multi-source warehouse view maintenance.* Technical report, Stanford University, 1997. Available at http://db.stanford.edu/pub/papers/strobe.ps.

[Zhu99] Y. Zhuge. *Incremental Maintenance of Consistent Data Warehouses.* Phd thesis, Stanford University, Departament of Computer Science, Stanford, CA 94305-2140, June 1999.

Apêndice A

Descrição Formal Completa dos Componentes do Data Warehouse do Projeto SAGU

Descrição formal do componente Integrador

- MSList = list of(string, agent, string, integer, integer).
- Consolidation-Time = integer.
- Init-System(_ , _) :: MSList, Consolidation-Time → action [diverging].

(1) Init-System(L: MSList, T: Consolidation-Time) =
    | Integrator-Init-Action(L)
    hence
    | Integrator-Action(L, T)

- Integrator-Init-Action(_) :: MSList → action [binding | communicating | completing]
  [using current bindings | current buffer].

(2) Integrator-Init-Action(L: MSList) =
```
    | give L
    then
    | | | unfolding
    | | | | | check not(the given list is empty-list)
    | | | and then
    | | | | | | subordinate an agent
    | | | | | and give component#1 of the head of the given list
    | | | | then bind the given agent#1 to the given string#2
    | | | and give tail of the given list
    | | | then unfold
    | | or check(the given list is empty-list)
    | and subordinate an agent then bind it to "SemaphoreWDB"
    | and subordinate an agent then bind it to "Loader"
    | and subordinate an agent then bind it to "Consolidator"
    | and subordinate an agent then bind it to "Updater"
    before
    | | unfolding
    | | | | | check not(the given list is empty-list)
    | | | and then
    | | | | | give head of the given list
    | | | | and then
    | | | | | | check(the given string#3 is "Cooperative-Source")
    | | | | | and then send a message [to the agent bound to the given string#1]
    | | | | |                         [containning the application of closure abstraction of
    | | | | |                         CS-Monitor-Function to the given agent#2]
    | | | | or
    | | | | | | check(the given string#3 is "Non-Cooperative-Source")
    | | | | | and then send a message [to the agent bound to the given string#1]
    | | | | |                         [containning the application of closure abstraction of
    | | | | |                         NCS-Monitor-Function to the given agent#2]
    | | | | and give tail of the given list
    | | | | then unfold
    | | | or check(the given list is empty-list)
    | | and send a message[to the agent bound to "SemaphoreWDB"]
    | |                   [containning the closure abstraction of SemaphoreWDB-Function]
    | | and send a message[to the agent bound to "Loader"]
    | |                   [containning the closure abstraction of Loader-Function]
    | | and send a message[to the agent bound to "Consolidator"]
    | |                   [containning the closure abstraction of Consolidator-Function]
    | | and send a message[to the agent bound to "Updater"]
    | |                   [containning the closure abstraction of Updater-Function]
```

- Integrator-Action (_ , _) :: MSList, Consolidation-Time → action
  [binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage]
- Consolidator.Log = list of tuples.
- Updater.Log = list of tuples.

(3) Integrator-Action(L: MSList, T: Consolidation-Time) =

```
| | give L
| then
| | unfolding
| | | | | check not(the given list is empty-list)
| | | and then
| | | | | | give head of the given list
| | | | | then Integrate-Source
| | | | and
| | | | | | give tail of the given list
| | | | | then unfold
| | | or check(the given list is empty-list)
| and
| | unfolding
| | | | Sleep T
| | then send a message [to the agent bound to "Consolidator"]
| |                    [containning an "OK"]
| | then receive a message [from the agent bound to "Consolidator"]
| |                        [containning a Consolidator.Log]
| | then send a message [to the agent bound to "Updater"]
| |                    [containning an "OK"]
| | then receive a message [from the agent bound to "Updater"]
| |                        [containning an Updater.Log]
| | then unfold
```

- Sleep _ :: integer → action

(1) Sleep I = □

- Integrator-Source :: action [receiving a tuple | completing][using current bindings | current buffer].

(4) Integrate-Source =

```
| check(the given string#3 is "Cooperative-Source") then
| | | give the given string#1
| | and then Integrate-Cooperative-Source
or
| check(the given string#3 is "Non-Cooperative-Source") then
| | | give(the given string#1, the given integer#4, the given integer#5)
| | and then Integrate-Non-Cooperative-Source
```

- Integrate-Cooperative-Source :: action [receiving an string|binding|storing|diverging|communicating] [using current bindings | current buffer | current storage].
- Translated-Update-List = list of tuples.
- Log-List = list of tuples.
- Errors-List = list of tuples.

(5) Integrate-Cooperative-Source =

```
| | allocate a cell
| then
| | | bind the given cell to "Monitor"
| | and store the given string#1 in it
hence
| unfolding
| | | | receive a message[from the agent stored in the cell bound to "Monitor"]
| | | |                  [containning a Translated-Update-List]
| | | then send a message [to the agent bound to "Loader"]
| | | |                   [containning the contents of the given message]
| | | then receive a message [from the agent bound to "Loader"]
| | | |                      [containning a (Log-List, Errors-List)]
| | | then
| | | | | | check not(the given list#2 is empty-list)
| | | | | and then Carry-Errors(the contents of the given message)
| | | | or check(the given list#2 is empty-list)
| | and then unfold
```

- Carry-Errors (_ , _) :: Errors-List, Log-List → action

(1) Carry-Errors $(L_1, L_2) = \square$

- Integrate-Non-Cooperative-Source :: action
  [receiving a (string, integer, integer) | binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage].
- Translated-Update-List = list of tuples.
- Log-List = list of tuples.
- Errors-List = list of tuples.

(6) Integrate-Non-Cooperative-Source =

```
| | | allocate a cell
| | then
| | | | bind the given cell to "Monitor"
| | | and store the given string#1 in it
| and
| | | allocate a cell
| | then
| | | | bind the given cell to "ABSN"
| | | and store the given integer#2 in it
| and
| | | allocate a cell
| | then
| | | | bind the given cell to "Wait-Time"
| | | and store the given integer#3 in it
| hence
| | unfolding
| | | | | send a message [to the agent stored in the cell bound to "Monitor"]
| | | | |                [containning the integer stored in the cell bound to "ABSN"]
| | | | then receive a message [from the agent stored in the cell bound to "Monitor"]
| | | |                        [containning a (ABSN, Translated-Update-List)]
| | | | then store the given ABSN#1 in the cell bound to "ABSN"
| | | | | and
| | | | | | | | check not(the given list#2 is empty-list)
| | | | | | | and then
| | | | | | | | | send a message [to the agent bound to "Loader"]
| | | | | | | | |                [containning the given list#2]
| | | | | | | then receive a message [from the agent bound to "Loader"]
| | | | | | |                        [containning a (Errors-List, Log-List)]
| | | | | | | then
| | | | | | | | | | | check not(the given list#1 is empty-list)
| | | | | | | | | | | and then Carry-Errors(the contents of the given message)
| | | | | | | | | | | or check(the given list#1 is empty-list)
| | | | | | | or check(the given list#2 is empty-list)
| | | | | and then Sleep "Wait-Time"
| | | | and then unfold
```

Descrição formal do componente Monitor para fontes de dados não cooperativas

- NCS-Coop-Monitor-Function :: action [receiving an agent | diverging].

(1) NCS-Coop-Monitor-Function =
    | NCS-Monitor-Init-Action
    hence
    | NCS-Monitor-Action

- NCS-Monitor-Init-Action :: action
  [receiving an agent | binding | completing | communicating][using current bindings | current buffer].

(2) NCS-Monitor-Init-Action =
    | | *subordinate an agent then bind it to "Auditor"*
    | and
    | | subordinate an agent then bind it to "Translator"
    hence
    | | send a message[to the agent bound to "Auditor"]
    | | [containning the application of
    | | closure abstraction of NCS-Auditor-Function to the given agent#1]
    | and
    | | send a message[to the agent bound to "Translator"]
    | | [containning the closure abstraction of NCS-Translator-Function]

- NCS-Monitor-Action :: action [diverging | communicating][using current bindings | current buffer].
- ABSN = integer.
- Translated-Update-List = list of tuples.

(3) NCS-Monitor-Action =
    unfolding
    | | | receive a message[from the agent bound to "Integrator"]
    | | | [containning an ABSN]
    | | then send a message[to the agent bound to "Auditor"]
    | | [containning the contents of the given message]
    | | then receive a message[from the agent bound to "Auditor"]
    | | [containning a (ABSN, Translated-Update-List)]
    | | then send a message[to the agent bound to "integrator"]
    | | [containning the contents of the given message]
    | and then unfold

Descrição formal do componente Auditor para fontes de dados não cooperativas

- NCS-Auditor-Function :: action
  [receiving an agent | binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage].
- ABSN = integer.
- Translated-Update-List = list of tuples.

(4) NCS-Auditor-Function =

```
| | allocate a cell
| then
| | | bind the given cell to "Source"
| | and store the given agent#1 in it
hence
| unfolding
| | | | | receive a message[from the agent bound to "Monitor"]
| | | | |                  [containning an ABSN]
| | | | then Query-Source(the contents of the given message)
| | | and then
| | | | unfolding
| | | | | | Receive-Next a message[from the agent stored in the cell bound to "Source"]
| | | | |                                  [containning a (ABSN, tuple)]
| | | | then give the contents of the given message
| | | | then
| | | | | | send a message[to the agent bound to "Translator"]
| | | | |                  [containning the rest of the given tuple]
| | | | and
| | | | | | | | check not(the rest of the given tuple is empty)
| | | | | | | and then unfold
| | | | | or
| | | | | | | | check(the rest of the given tuple is empty)
| | | | | | | and then
| | | | | | | | | give the given ABSN#1
| | | | | | | and
| | | | | | | | | | receive a message[from the agent bound to "Translator"]
| | | | | | | | |                  [containning a Translated-Update-List]
| | | | | | | | | then give the contents of the given message
| | | | | | | then send a message[to the agent bound to "Monitor"]
| | | | | | |                  [containning the given ABSN#1, the given list#2]
| | and then unfold
```

- Receive-Next _ :: yielder → action [completing][using current buffer | giving a message].

(1) Receive-Next (Y ¡= yielder) =
    | patiently
    | | | choose a Y [in set of(head(Sort-ABSN(the current buffer)))]
    | then
    | | | remove the given message and give it

- Sort-ABSN _ :: buffer → action [completing][giving a buffer]

(1) Sort-ABSN (A) = □

- Query-Source _ :: ABSN → Action

(1) Query-Source A = □

Descrição formal do componente Tradutor para fontes de dados não cooperativas

- NCS-Translator-Function :: action [binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage].

(5) NCS-Translator-Function =

```
| | allocate a cell
| then bind the given cell to "Translated-Update-List"
hence
| unfolding
| | | | store the empty-list in the cell bound to "Translated-Update-List"
| | | and then
| | | | unfolding
| | | | | | receive a message[from the agent bound to "Auditor"]
| | | | | |                  [containning a tuple]
| | | | | then
| | | | | | | check not(the given message is empty)
| | | | | and then
| | | | | | | | check(the given string#2 is in set["DSC" | "DSD" | "DSM"])
| | | | | | and then
| | | | | | | | Translate the given tuple
| | | | | | then give list of(the given tuple,
| | | | | |                   branches of the list stored in the cell bound
| | | | | |                   to "Translated-Update-list")
| | | | | | then store it in the cell bound to "Translated-Update-List"
| | | | | | then unfold
| | | | | or
| | | | | | | check not(the given string#2 is in set["DSC" | "DSD" | "DSM"])
| | | | | | and then unfold
| | | | or
| | | | | | check(the given message is empty)
| | | | | and then send a message [to the agent bound to "Auditor"]
| | | | | |                         [containning the list stored in cell bound
| | | | | |                         to "Translated-Update-List"]
| | and then unfold
```

- Translate _ :: tuple → action [completing][giving a tuple]

(1) Translate $(T) = \square$

Descrição formal do componente Monitor para fontes de dados cooperativas

- CS-Monitor-Function :: action [receiving an agent | diverging].

(1) CS-Monitor-Function =
    | CS-Monitor-Init-Action
    hence
    | CS-Monitor-Action

- CS-Monitor-Init-Action :: action
  [binding | commpleting | communicating][using current bindings | current buffer].

(2) CS-Monitor-Init-Action =
    | subordinate an agent then bind it to "Translator"
    hence
    | send a message[to the agent bound to "Translator"]
    | [containning the closure abstraction of CS-Translator-Function]

- CS-Monitor-Action :: action [receiving an agent | binding | storing | diverging | communicating]
  [using current bindings | current buffer | current storage].
- Update-List = list of tuples.
- Translated-Update-List = list of tuples.

(3) CS-Monitor-Action =
    | | allocate a cell
    | then
    | | | bind the given cell to "Source"
    | | and store the given agent#1 in it
    hence
    | unfolding
    | | | | receive a message[from the agent bound to "Source"]
    | | | | [containning an Update-List]
    | | | then send a message[to the agent bound to "Translator"]
    | | | [containning the contents of the given message]
    | | then receive a message[from the agent bound to "Translator"]
    | | [containning an Translated-Update-List]
    | | then send a message[to the agent bound to "Integrator"]
    | | [containning the contents of the given message]
    | | and then unfold

Descrição formal do componente Tradutor

- CS-Translator-Function :: action [diverging | communicating][using current buffer].
- Update-List = list of tuples.

(4) CS-Translator-Function =
unfolding
| receive a message[from the agent bound to "Monitor"]
| [containning an Update-List]
| then Translate-List(the given list)
| then send a message [to the agent bound to "Monitor"]
| [containning the given list]
| and then unfold

- Translate-List _ :: list → action [completing][giving a list]

(1) Translate-List $(T) = \square$

Descrição formal do componente Carregador

- Loader-Function :: action [diverging].

(1) Loader-Function =
| Loader-Init-Action
hence
| Loader-Action

- Loader-Init-Action :: action [binding | completing][using current bindings].
- Log-List = list of tuples.
- Errors-List = list of tuples.

(2) Loader-Init-Action =

```
| | | allocate a cell
| | then
| | | bind the given cell to "Log-List"
| and
| | | allocate a cell
| | then
| | | bind the given cell to "Errors-List"
```

- Loader-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) Loader-Action =

```
unfolding
| | | store the empty-list in the cell bound to "Log-List"
| | and
| | | store the empty-list in the cell bound to "Errors-List"
| then receive a message [from the agent bound to "Integrator"]
|                        [containning a Translated-Update-List]
| then
| | | | send a message [to the agent bound to "SemaphoreWDB"]
| | | |                [containning a "Wait"]
| | | and then receive a message [from the agent bound to "SemaphoreWDB"]
| | |                            [containning an "OK"]
| | and then Carry-Work-Database(the contents of the given message)
| then send a message [to the agent bound to "SemaphoreWDB"]
|                     [containning a "Signal"]
| then send a message [to the agent bound to "Integrator"]
|                     [containning a (list stored in the cell bound to "Errors-List",
|                     list stored in the cell bound to "Log-List")]
| then unfold
```

- Carry-Work-Database :: action [completing][giving a (Errors-List,Log-List)]

(1) Carry-Work-Database = □

Descrição formal do componente Consolidador

- Consolidator-Function :: action [diverging].

(1) Consolidator-Function =
    | Consolidator-Init-Action
    hence
    | Consolidator-Action

- Consolidator-Init-Action :: action [binding | completing][using current bindings].

(2) Consolidator-Init-Action =
    | | allocate a cell
    | then
    | | bind the given cell to "Consolidator.Log"

- Consolidator-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) Consolidator-Action =
    unfolding
    | | store the empty-list in the cell bound to "Consolidator.Log"
    | then receive a message [from the agent bound to "Integrator"]
    | [containning an "OK"]
    | then send a message [to the agent bound to "SemaphoreWDB"]
    | [containning a "Wait"]
    | then receive a message [from the agent bound to "SemaphoreWDB"]
    | [containning an "OK"]
    | then Integrate-Mirror-Database
    | then send a message [to the agent bound to "SemaphoreWDB"]
    | [containning a "Signal"]
    | then send a message [to the agent bound to "Integrator"]
    | [containning the list stored in the cell bound to "Consolidator.Log"]
    | then unfold

- Integrate-Mirror-Database :: action [completing][giving a Consolidator.Log]

(1) Integrate-Mirror-Database = □

Descrição formal do componente Atualizador

- Updater-Function :: action [diverging].

(1) Updater-Function =
    | Updater-Init-Action
    hence
    | Updater-Action

- Updater-Init-Action :: action [binding | completing][using current bindings].

(2) Updater-Init-Action =
    | | allocate a cell
    then
    | | bind the given cell to "Updater.Log"

- Updater-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) Updater-Action =
    unfolding
    | | store the empty-list in the cell bound to "Updater.Log"
    then receive a message [from the agent bound to "Integrator"]
                           [containning an "OK"]
    then Update-Production-Database
    then send a message [to the agent bound to "Integrator"]
                        [containning the list stored in the cell bound to "Updater.Log"]
    then unfold

- Update-Production-Database :: action [completing][giving an Updater-Log]

(1) Update-Production-Database = □

Descrição formal do componente Semáforo da Base de Dados de Trabalho

- SemaphoreWDB-Function :: action [diverging].

(1) SemaphoreDB-Function =
| SemaphoreDB-Init-Action
hence
| SemaphoreDB-Action

- SemaphoreWDB-Init-Action :: action [storing | completing][using current bindings | current storage].

(2) SemaphoreWDB-Init-Action =
| | allocate a cell
| then
| | bind the given cell to "User"
and
| | allocate a cell
| then
| | bind the given cell to "Value"
| | and then
| | | store 0 in it

- SemaphoreWDB-Action :: action
  [storing | diverging | communicating][using current bindings | current buffer | current storage].

(3) SemaphoreWDB-Action =

```
unfolding
| | | check(the value stored in the cell bound to "Value" is 1)
| | and then
| | | | receive a message[from the agent stored in the cell bound to "User"]
| | | |                  [containning a "Signal"]
| | | then store 0 in the cell bound to "Value"
| | | then unfold
| | or
| | | check(the value stored in the cell bound to "Value" is 0)
| | and then
| | | receive a message[from any agent][containning a "Wait"]
| | | then
| | | | | store the sender of the given message in the cell bound to "User"
| | | | and
| | | | | store 1 in the cell bound to "Value"
| | | and then
| | | | send a message[to the agent stored in the cell bound to "User"]
| | | |              [containning an "OK"]
| | | then unfold
```