Universidade Estadual de Campinas - UNICAMP
Faculdade de Engenharia Elétrica e de Computação - FEEC
Departamento de Telemática - DT

O uso combinado da técnica de modelagem baseada em objetos OMT com a linguagem de especificação formal SDL como metodologia alternativa para o desenvolvimento do ambiente de software AIDA

Carla Geovana do Nascimento Macário

Banca Examinadora:

Prof. Dr. Walter da Cunha Borelli(Orientador)
Dr. Kleber X. S. de Souza (CNPTIA/Embrapa)
Prof. Dr. Eleri Cardozo (DCA/FEEC/UNICAMP)
Prof. Dr. Shusaburo Motoyama (DT/FEEC/UNICAMP - Suplente)

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para obtenção do título de Mestre.

Campinas - SP Brasil
22 de dezembro de 1997

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

M118u

Macário, Carla Geovana do Nascimento
O uso combinado da técnica de modelagem baseada em objetos OMT com a linguagem de especificação formal SDL como metodologia alternativa para o desenvolvimento do ambiente de software AIDA / Carla Geovana do Nascimento Macário.-- Campinas, SP: [s.n.], 1997.

Orientador: Walter da Cunha Borelli
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Engenharia de software. 2. Software - Manutenção 3. Software - Validação. 4. SDL (Linguagem de programação de computador). 5. Simulação (Computadores). I. Borelli, Walter da Cunha. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Dedico este trabalho aos meus pais,

a quem devo tudo que sou.

Obrigada por sempre terem confiado em mim

e por me ensinarem a nunca desistir.

Agradecimentos

AO CNPTIA/Embrapa, na pessoa de seu chefe, Dr. Moacir Pedroso Júnior, pela chance de realizar este trabalho.

Aos meus pais que sempre investiram em mim, e à minha irmã, a "Carlinha", que eu adoro e que nunca me esquece.

Ao Prof. Borelli, pelas oportunidades oferecidas e por me acompanhar no desenvolvimento da minha tese.

Ao meu marido Alexandre, que sempre esteve ao meu lado, me apoiando em tudo e me incentivando nas horas mais difíceis.

Aos meus colegas do projeto AIDA e da Embrapa, por todo o apoio e incentivo para que eu conseguisse terminar. Em especial à Fernanda, ao Roberto, ao Chaim, à Adriana, ao Zé Ruy, à Sílvia, ao Paulinho, ao Kléber e à Magda, que foi quem sugeriu este trabalho. Um abraço para todos!

Aos professores do DT Ivanil, Pedro, Akebo, Paulo, Motoyama e também ao Jesus, que de alguma forma contribuiram para que eu realizasse o meu trabalho.

E a todas as outras pessoas que me ajudaram na elaboração da tese.

Índice

Resumo

Este trabalho propõe uma metodologia alternativa para o desenvolvimento e evolução do AIDA, um software para o gerenciamento e análise de dados experimentais, em desenvolvimento na Embrapa. Esta nova metodologia consiste no uso combinado da técnica OMT (Object Modeling Technique) com a linguagem de especificação formal SDL (Specification and Description Language), e apresenta facilidades que produzem ganhos no processo de desenvolvimento de software, como a possibilidade de validação e de simulação do sistema, e também a geração de código para sua prototipação. A partir do modelo de classes proposto pela OMT, passa-se à especificação do sistema em SDL, levando-se em conta na elaboração de ambos os modelos, conceitos como reuso, herança e evolução de sistemas. A validação, a simulação e a geração de código do sistema tornam-se possíveis com o uso da ferramenta CASE SDT[1] (SDL Design Tool - Telelogic, Suécia). É apresentada a evolução do ambiente AIDA centralizado até uma versão distribuída, considerando uma arquitetura cliente-servidor com mecanismo CORBA, bem como exemplos de simulação e de validação destes sistemas.

Palavras-Chave: Engenharia de Software, Evolução de Software, Object Modeling Technique - OMT, Specification and Description Language - SDL, Message Sequence Chart - MSC, Especificação Formal, Validação, Simulação, Prototipação, Ambiente de Software AIDA/Embrapa, Common Object Request Broker Arquitecture - CORBA, Interface Definition Language - IDL.

[1] SDT Pacote SDT, versão 3.02 (SDT Base, MSC Editor, Simulator e Validator) e versão 3.2 (SDT Base, MSC Editor, Simulator, Validator e OMT Editor): adquirido pelo DT/FEEC/UNICAMP através de Projeto Temático - FAPESP (Proc. 91/3660-0).

Abstract

The present work proposes an alternative methodology for the development and evolution of AIDA, a software environment for the management and analysis of experimental data, being developed at Embrapa, Brazil. This new technique consists of the combined use of OMT (Object Modeling Technique) with the formal specification language SDL (Specification and Description Language), in its 1992 version SDL-92, presenting strong facilities on the software development process, allowing the validation, simulation and eventual code generation for software prototyping. From the system object model and after mapping to SDL-92, the formal specification is generated considering concepts such as reuse, inheritance, and system evolution. The validation, the simulation and code generation of the system is allowed through the use of the CASE tool SDT[2] (SDL Design Tool - Telelogic, Sweden). The specification of the centralized AIDA to its evolution to distributed version are presented, validated and simulated. This is followed up with a specialization and validation of thedistributed AIDA, considering an arquitecture client-server with CORBA mechanism.

Key-words: Software Engineering, Software Evolution, Object Modeling Technique - OMT, Specification and Description Language - SDL, Message Sequence Chart - MSC, Formal Specification, Validation, Simulation, Prototyping, AIDA Software Environment/Embrapa, Common Object Request Broker Arquitecture - CORBA, Interface Definition Language - IDL.

[2] SDT Package, version 3.02 (SDT Base, MSC Editor, Simulator and Validator) e version 3.2 (SDT Base, MSC Editor, Simulator, Validator and OMT Editor): purchased by DT/FEEC/UNICAMP through a grant from the State of Sao Paulo Foundation for Research - FAPESP (Proc. 91/3660-0).

Publicações

Macário, C.G. N.; Pedroso Júnior, M.; Borelli, W. C. OMT+SDL: An alternative methodology for the AIDA development. In: III CONGRESO INTERNACIONAL DE INGENIERIA INFORMATICA, Buenos Aires. ICIE 96-97: Proceedings. Buenos Aires:Universidad de Buenos Aires - Faculdad de Ingenieria - Departamento de Computacion, [1997]. p.351-365.

Macário, C.G. N.; Pedroso M. J; Borelli, W. C. Designing a multi-user software environment for development and analysis using a combination of OMT and SDL92. In:SDL'97 TIME FOR TESTING SDL, MSC and Trends, Eighth SDL Forum: Proceedings. Evry-France Elsevier Science Publishers, [1997]. p.03-17.

Lista de Siglas e Acrônimos

CNPTIA - Centro Nacional de Pesquisa Tecnológica em Informática para a Agricultura

Embrapa - Empresa Brasileira de Pesquisa Agropecuária

AIDA - Ambiente Integrado para Desenvolvimento e Análise

CentralizedAIDA - versão centralizada do AIDA

ConcurrentAIDA - versão concorrente do AIDA

SingleUserAIDACORBA - nova versão centralizada do AIDA

DistributedAIDACORBA - versão concorrente do AIDA em ambiente CORBA

OMT - Object Modeling Technique

SDL - Specification and Description Language

MSC - Message Sequence Chart

ASN.1 - Abstract Syntax Notation One

SDT - SDL Design Tool

CORBA - Common Object Request Broker Architecture

ORB - Object Request Broker

IDL - Interface Definition Language

OMG - Object Management Group

OMA - Object Management Architecture

Initial - iniciador do AIDA

AnalysisEnv - gerenciador do ambiente de análise

EInterf - interface gráfica do gerenciador do ambiente de análise

Tool - ferramenta de análise ou formatador de relatórios

TInterf - interface gráfica da ferramenta de análise ou do formatador de relatórios

Calculation - executa cálculos específicos da ferramenta de análise

Controller - controlador dos servidores de processamento

Server - servidor de processamento

FConv - conversor de formatos de arquivos externos

FCInterf - interface gráfica do conversor de formatos

DataFileManip - manipula o arquivo de dados

DataFile - arquivo de dados

VirtDataDef - define o arquivo virtual de dados

VirtFile - definição do arquivo virtual de dados

Manager - agrupa módulos ou processos que fazem o gerenciamento do AIDA

ToolExecution - agrupa módulos ou processos para a execução da ferramenta de análise ou formatação de relatório

FileConvertion - agrupa módulos ou processos para a conversão de formatos de arquivos externos

DataManipPack - possui processos para a manipulação de arquivos de dados

1. Introdução

Este trabalho propõe uma metodologia alternativa para o desenvolvimento e evolução do AIDA, um ambiente de software para o gerenciamento e análise de dados experimentais, em desenvolvimento na Embrapa. Esta metodologia consiste no uso combinado da técnica OMT (Object Modeling Technique), muito utilizada no desenvolvimento de software, com a linguagem de especificação formal SDL (Specification and Description Language), muito usada na área de telecomunicações, unindo as vantagens apresentadas por ambas[Macário, et al., 1997a] [Macário, et al., 1997b].

A técnica OMT [Rumbaugh et al., 1991] é uma das mais bem sucedidas metodologias de desenvolvimento orientado a objetos, estruturando o sistema em termos de objetos e conceitos relacionados, consistindo numa boa alternativa para representar a arquitetura do sistema. Já a linguagem SDL[ITU-T, 1993a] é uma linguagem de especificação formal bem estabelecida na área de telecomunicações, permitindo a definição das características dinâmicas dos sistemas, ou seja, do seu comportamento. Além disso, por ser formal, possibilita a validação, a simulação e a prototipação dos sistemas ainda nas fases iniciais de desenvolvimento.

O AIDA é um ambiente de software composto por ferramentas para a análise matemática e estatística de dados experimentais em desenvolvimento no CNPTIA/Embrapa[3], consistindo numa evolução do Software NTIA [Macário et al., 1991], um outro ambiente também desenvolvido pela Embrapa, largamente utilizado pelas unidades da empresa e do SNPA (Sistema Nacional de Pesquisa Agropecuária), mas cujas manutenção e evolução haviam se tornado muito difíceis.

Vários esforços têm sido feitos na elaboração de técnicas para o desenvolvimento de software de qualidade, já que um software de maior qualidade necessita de uma menor manutenção, e ocorrendo a manutenção, ela é muito mais rápida. O consenso é que a melhoria da qualidade só é obtida pela maturidade do processo de desenvolvimento de software. Procedimentos como a adoção de metodologias de desenvolvimento, a melhoria na documentação do sistema, a verificação e a validação do software desde as suas fases iniciais, reduzem o tempo gasto com testes, e conseqüentemente, com o desenvolvimento como um todo, produzindo um software de melhor qualidade.

[3] Centro Nacional de Pesquisa Tecnológica em Informática para a Agricultura, uma das unidades de pesquisa da Empresa Brasileira de Pesquisa Agropecuária.

Assim, visando a produção rápida de um software funcionalmente equivalente, mais fácil de evoluir e de manter que o Software NTIA, em 1995 foi proposto pelo CNPTIA o ambiente AIDA, que seria desenvolvido adotando-se a metodologia orientada a objetos OMT, buscando a sistematização do processo de desenvolvimento, bem como a produção de um software de qualidade. Entretanto, na sua primeira versão [Chaim et al., 1996], alguns problemas foram encontrados, principalmente decorrentes do fato da OMT não ser formal, não permitindo a validação do sistema desde suas fases iniciais de desenvolvimento.

Com a disponibilidade da linguagem SDL e da ferramenta CASE SDL Design Tool (SDT)[4] no Departamento de Telemática (DT), foi proposta esta metodologia alternativa combinada que faz uso da técnica OMT em conjunto com a linguagem SDL, apresentando facilidades que produzem ganhos no processo de desenvolvimento de software, como a possibilidade de validação e de simulação do sistema, e também a geração de código para sua prototipação.

Esta metodologia mostrou-se excelente na especificação do sistema desenvolvido na Embrapa, tendo sido adequada às necessidades do sistema na sua evolução de um ambiente simples (centralizado) [Macário, et al., 1997a] para outras configurações, centralizada e concorrente, até a sua versão distribuída em um ambiente CORBA (Common Object Request Broker Arquitecture), visando uma eventual implementação numa configuração cliente-servidor presente atualmente em diversas redes de computadores existentes, característica comum a vários ambientes integrados de desenvolvimento.

1.1. Estruturação

Esta trabalho divide-se em cinco capítulos e dois apêndices. O capítulo 2 traz um breve histórico do AIDA com suas funcionalidades principais, e propõe a metodologia combinada OMT + SDL a ser utilizada em detalhes nos capítulos 3 e 4.

O capítulo 3 apresenta a aplicação da metodologia proposta numa versão centralizada do ambiente AIDA, assim como a sua evolução para uma versão concorrente. Estas versões tratam da visão geral do AIDA, mostrando os modelos OMT e as especificações SDL correspondentes. Através da introdução da versão centralizada são apresentados os conceitos básicos de SDL-92 para a elaboração da sua especificação SDL.

[4] SDT Pacote SDT, versão 3.02 (SDT Base, MSC Editor, Simulator e Validator) e versão 3.2 (SDT Base, MSC Editor, Simulator, Validator e OMT Editor): adquirido pelo DT/FEEC/UNICAMP através de Projeto Temático - FAPESP (Proc. 91/3660-0).

Já a versão concorrente mostra as estruturas presentes nesta linguagem para o reuso e a evolução de especificações. Ao final apresentam-se as validações de ambas as versões, incluindo exemplos de erros encontrados na atividade de validação, e um exemplo de simulação da versão AIDA concorrente, através do diagrama MSC (Message Sequence Chart) [ITU-T, 1993b] correspondente.

O capítulo 4 apresenta uma versão distribuída do AIDA em arquitetura CORBA, e algumas das funcionalidades existentes no sistema, como o tratamento de arquivos de dados, são detalhadas. São mostradas as construções utilizadas para adequar a metodologia ao novo sistema, apresentando os modelos OMT e SDL das versões centralizada e distribuída do AIDA_CORBA, bem como a especificação de suas interfaces em linguagem IDL. Os resultados da validação do sistema AIDA_CORBA distribuído e um exemplo de sua simulação são exibidos ao final do capítulo.

No capítulo 5 são apresentadas as conclusões principais e sugestões para eventuais trabalhos futuros que podem dar continuidade a esta tese de mestrado.

Os apêndices A e B apresentam, respectivamente, resumos sobre a linguagem SDL e a arquitetura CORBA.

2. AIDA: Uma Evolução do Ambiente de Software NTIA

2.1. Introdução

Este capítulo apresenta um breve histórico e a descrição das funcionalidades do AIDA [Chaim et al., 1994] [Embrapa, 1996], um ambiente de software em desenvolvimento na Embrapa, para o gerenciamento e a análise de dados experimentais gerados em pesquisas agropecuárias. Em seguida é proposta uma metodologia alternativa para o seu desenvolvimento, que apresenta o uso combinado da técnica OMT (Object Modeling Technique) [Rumbaugh et al., 1991] e da linguagem formal SDL-92 (Specification and Description Language) [ITU-T, 1993a].

2.2. Histórico

A Embrapa - Empresa Brasileira de Pesquisa Agropecuária, tem como missão, gerar e promover conhecimento e tecnologia para o desenvolvimento sustentado do complexo agro-industrial, em benefício da sociedade, sendo formada por 39 unidades de pesquisa distribuídas pelo país, e uma unidade central. As unidades de pesquisa, cuja distribuição pelo Brasil é apresentada na Figura 2.1, dividem-se em quatro tipos:

- centros de pesquisa ecorregionais, que trabalham com pesquisa em áreas ecorregionais, como a floresta amazônica e o pantanal, entre outras;
- centro nacionais de produtos, que constituem a grande maioria das unidades e trabalham na pesquisa de produtos específicos como soja, trigo, gado de leite e gado de corte;
- centros de serviços, que constituem unidades especiais para a execução de serviços, como a produção de sementes e a produção de informação;
- centros nacionais temáticos, que realizam pesquisas em temas específicos, como tecnologia de alimentos, solos e informática.

O Centro Nacional de Pesquisa Tecnológica em Informática para a Agricultura - CNPTIA, localizado em Campinas-SP, é um dos centros temáticos da Embrapa e tem como principal missão o desenvolvimento de ferramentas, métodos e técnicas para o apoio à pesquisa agropecuária da Embrapa e dos demais centros de pesquisa do SNPA - Sistema

Nacional de Pesquisa Agropecuária. Atualmente o CNPTIA apresenta diversos projetos de pesquisa, sendo um deles o AIDA, o objeto de estudo desta tese.

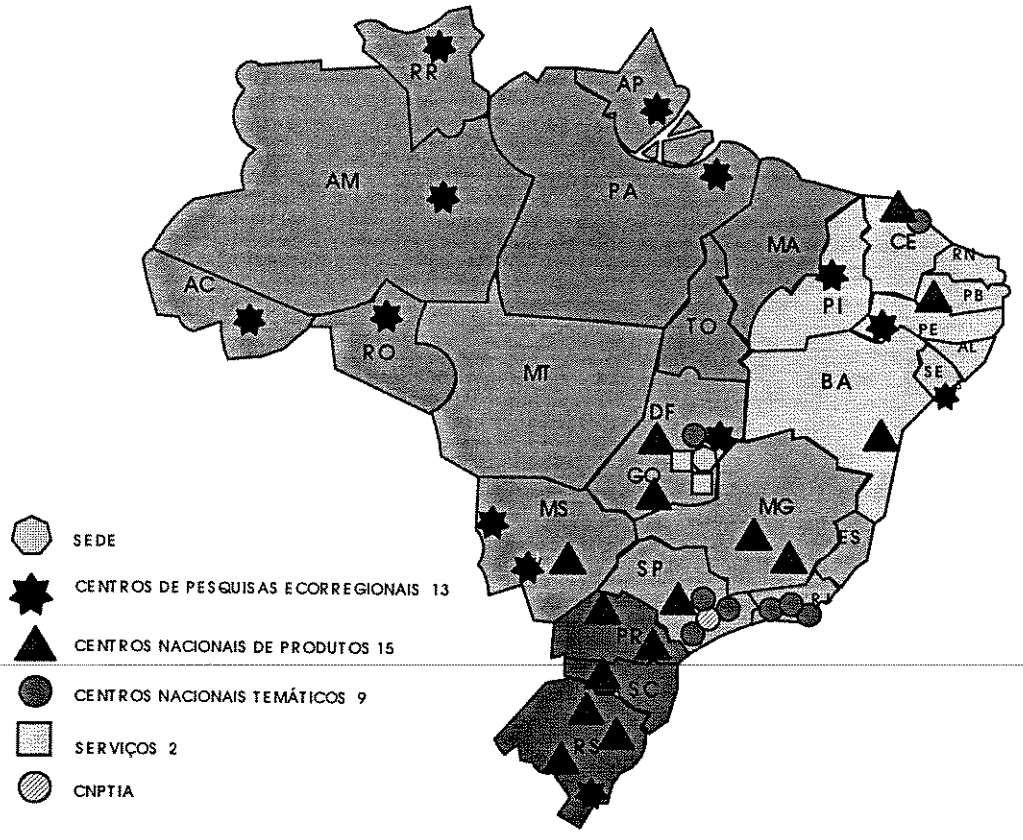


Figura 2.1 - Unidades de Pesquisa da Embrapa

O AIDA - Ambiente Integrado para Desenvolvimento e Análise, consiste numa evolução do Software NTIA - SW NTIA [Macário et al., 1991], um ambiente de software que começou a ser desenvolvido em 1986 no CNPTIA para suprir uma carência que existia na Embrapa: a inexistência de produtos voltados para a análise matemática e estatística de dados que pudessem ser executados em microcomputadores.

As pesquisas agropecuárias geram um grande volume de dados a serem analisados utilizando-se modelos matemáticos e estatísticos. Até 1986, todas as análises de dados da empresa eram realizadas num computador de grande porte, localizado na unidade central da Embrapa em Brasília, e desta forma, para a realização das análises necessárias, os pesquisadores das unidades descentralizadas enviavam seus dados experimentais para a sede da empresa, num procedimento sujeito a atrasos e a dificuldades, como a ocorrência de erros nos resultados, decorrentes de preenchimento errado dos formulários que

continham os dados, ou mesmo na sua digitação para o processamento no computador. Diante disto, quando os microcomputadores começaram a tornar-se disponíveis nas unidades de pesquisa, teve início o desenvolvimento do SW NTIA, cujo principal objetivo era prover um software para a análise matemática e estatística de dados, que pudesse ser executado em microcomputadores, viabilizando a realização das análises pelos próprios pesquisadores nas unidades descentralizadas.

Assim, teve início o seu desenvolvimento, que foi baseado no sistema estatístico em uso no computador central da Embrapa, o Statistical Analysis System - SAS [SAS, 1985]. Sua larga utilização por pesquisadores da Embrapa e do SNPA levou a uma evolução constante, passando a incluir funcionalidades que não haviam sido originalmente previstas e permitindo a sua execução também em ambientes UNIX. Entretanto, com o seu crescimento, a sua manutenção tornou-se trabalhosa e após uma breve análise percebeu-se que não seria fácil a incorporação de novas tecnologias, como por exemplo a adoção de uma interface gráfica baseada em janelas e um sistema de armazenamento de dados mais eficiente do que o utilizado.

Foi então proposto, em meados de 1994, o desenvolvimento do AIDA [Chaim et al., 1994], um novo ambiente de software voltado para a análise, que apresentasse as vantagens do SW NTIA, mas com uma linguagem de programação única, e utilizasse algumas das tecnologias disponíveis, como Sistema Gerenciador de Banco de Dados - SGBD e interface gráfica. A idéia básica foi a de reestruturar o SW NTIA, incorporando novos conceitos que facilitassem a sua futura evolução, aproveitando todo o conhecimento acumulado no seu desenvolvimento.

Vários esforços têm sido feitos para o desenvolvimento de software de qualidade, como a proposição do modelo CMM (Capability Maturity Model) [Taurion, 1997][Belloquim, 1997], um modelo de qualidade de software que se baseia na maturidade organizacional da empresa para desenvolver software, qualificando-a através de cinco níveis diferentes de maturidade de processos. O avanço da empresa nestes níveis indica que ela possui uma capacidade maior de produzir software de qualidade, já que a melhoria da qualidade só é obtida pela maturidade dos processos de desenvolvimento de software. Neste sentido, um outro objetivo do projeto AIDA era funcionar como projeto piloto para a utilização do documento denominado Cartilha Azul: Guia do Processo de Desenvolvimento de Software do CNPTIA [Pacheco et al., 1997], no qual são descritas diretrizes para o processo de desenvolvimento de software. Pretendia-se provar que a utilização de um processo

sistemático para desenvolvimento de software aumenta a produtividade e qualidade desta atividade e facilita futuras manutenções e evoluções.

2.3. AIDA: funcionalidades básicas

O AIDA é um ambiente de software multiplataforma voltado para o domínio de análise de dados. Dentre outras coisas, deve ser provido de interface simples, clara e eficiente, permitir a fácil incorporação de novas funcionalidades, viabilizar a incorporação de tecnologias e ferramentas emergentes, e integrar facilmente novos métodos de análise de dados.

As ferramentas que realizam as análises de dados são organizadas em grupos:

- estatísticas descritivas: estatísticas básicas, freqüências e correlações;
- modelos lineares: análise de variância uni e multivariada, regressão linear e stepwise;
- pesquisa operacional: programação linear usando o algoritmo simplex revisado;
- gráficas: gráficos de barra, coluna, setorial, 2D, 3D e curvas de níveis.

A execução destas ferramentas gera um relatório padrão, que pode ser exibido na tela ou enviado para a impressora. Em alguns casos também são gerados arquivos de saída. Os relatórios gerados podem ser editados através do Formatador de Relatórios, que oferece comandos para a personalização dos relatórios produzidos.

Uma execução completa de uma ou mais ferramentas de análises de dados e/ou de formatação de relatórios no ambiente AIDA é definida como uma sessão de trabalho. Todos os comandos executados durante esta sessão são armazenados em um arquivo, permitindo a sua recuperação num outro momento e viabilizando ao usuário manter um histórico de suas análises. Este arquivo contém os relatórios gerados e também informações sobre as ferramentas utilizadas para realizar as análises de dados durante a sessão com suas opções de execução. Os arquivos de dados de entrada escritos num formato diferente do reconhecido pelo AIDA podem ser utilizados no ambiente após serem convertidos pelo Conversor de Formatos.

No início da execução do AIDA é apresentada a tela principal do sistema, onde o usuário pode começar uma sessão de trabalho, converter um arquivo de dados de entrada, ou finalizar a execução do ambiente. A sessão de trabalho pode ser uma nova ou uma já

existente, e no caso de uma nova o ambiente solicita um resumo com uma breve descrição do que vai ser feito. Uma vez iniciada uma sessão, o ambiente apresenta a tela de sessão com os grupos de ferramentas de análise disponíveis e também opções para formatar um relatório gerado por uma ferramenta, finalizar a sessão de trabalho ou terminar a execução do ambiente. Ao selecionar um grupo, é exibida a lista de ferramentas de análise deste grupo, e o usuário deve informar qual delas será utilizada. Quando a ferramenta selecionada é iniciada, o sistema exibe uma tela com as opções específicas para a execução da análise correspondente, e o usuário deve definir os arquivos sobre os quais será feita esta análise. Essas opções são validadas, e não havendo erros, o cálculo do algoritmo é realizado sobre os arquivos definidos. A execução deste cálculo gera um relatório de saída, e em algumas das ferramentas também um arquivo de dados. Ao final da execução da ferramenta, o sistema retorna para a tela de sessão.

A opção de formatação de relatório exibe a lista de relatórios disponíveis em uma sessão de trabalho. Selecionado um relatório, o Formatador de Relatórios apresenta as opções de execução que são: formatar um item do relatório, salvar o relatório ou terminar a formatação. Os resultados da análise nunca ficam disponíveis para formatação Apenas itens como cabeçalho, título e rótulos (*labels*) podem ser alterados. Quando um deste itens é alterado, ele é validado e o seu resultado é exibido para o usuário. A opção de terminar a formatação pergunta se o usuário deseja ou não salvar o relatório, o qual é atualizado no arquivo de sessão de trabalho.

A opção de finalizar uma sessão de trabalho salva todas as informações disponíveis no arquivo de sessão, retornando à tela principal e permitindo ao usuário iniciar uma outra sessão de trabalho. Já a opção de terminar a execução do ambiente finaliza todas as execuções, bem como a sessão de trabalho atual, saindo do ambiente AIDA.

A opção de conversão de arquivos exibe uma tela que requisita o nome do arquivo a ser formatado. Utilizando um conversor adequado, o arquivo é convertido para o formato do AIDA e em seguida exibido ao usuário. Este arquivo é salvo com o novo formato e com o nome determinado pelo sistema.

A estrutura geral do ambiente AIDA segundo o seu relacionamento com elementos externos é composta por três partes: o(s) usuário(s), o sistema AIDA em si, e os arquivos de dados de entrada e saída. A representação desta estrutura pode ser vista na Figura 2.2, onde:

usuário utiliza o sistema AIDA para realizar análises matemáticas e estatísticas sobre arquivos de dados;

sistema AIDA representa o ambiente de software AIDA, com suas funcionalidades (ferramentas de análise, formatador de relatórios e conversor de formatos);

arquivos de dados podem ser de entrada ou de saída; os de entrada são utilizados pelas ferramentas análise, e os de saída são aqueles gerados pelo sistema com o resultados destas análises.

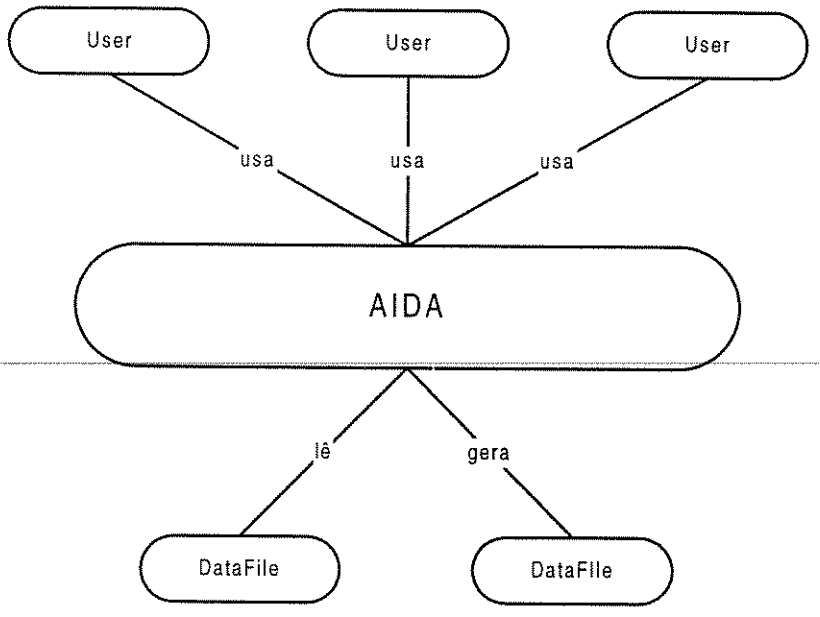


Figura 2.2 - Modelo Contextual do Sistema AIDA

Em uma análise sobre como construir este novo produto, e visando atender aos requisitos estabelecidos, chegou-se à conclusão de que seria importante a adoção de uma metodologia de engenharia de software que permitisse o desenvolvimento rápido do novo ambiente, bem como facilitasse sua evolução. Desta forma, optou-se por uma metodologia baseada em orientação a objetos, já que esta abordagem, através dos conceitos que carrega, induz ao reuso e à modularidade, e facilita o desenvolvimento e a manutenção de software.

Assim, adotando-se a Object Modeling Technique - OMT [Rumbaugh et al., 1991], foi apresentada, em meados de 1996 no CNPTIA, uma versão de demonstração do sistema AIDA centralizado para o ambiente MS-Windows, que contemplava apenas ferramentas

atísticas [Chaim et al., 1996]. Nesta versão centralizada, apenas uma única pessoa pode cutar o sistema, executando somente uma ferramenta por vez. Basicamente, esta são consistiu numa restruturação do SW NTIA, com a aplicação de conceitos de objetos.

2.4. Problemas Encontrados na 1ª. Versão

Nesta experiência inicial foram identificadas algumas dificuldades que levaram a um aso no seu cronograma de execução [Chaim et al., 1996]. Em primeiro lugar a dificuldade especificação e no entendimento comum do software, em parte pela falta de uma amenta que facilitasse a utilização da técnica OMT. Em segundo lugar, e principalmente, mpossibilidade de simulação e de validação do sistema ainda nas fases iniciais de senvolvimento, o que permitiria a identificação de erros e garantiria um custo menor no senvolvimento do sistema como um todo.

Neste sentido, apesar da OMT se mostrar como uma excelente opção para a scrição de software, não consiste numa técnica formal de especificação, e assim não mite a geração de código a partir da especificação para a simulação ou a validação do ema em desenvolvimento, ou mesmo a sua prototipação. Desta forma, os ganhos venientes destas atividades não são possíveis apenas com o uso desta metodologia.

A utilização de uma metodologia orientada a objetos permite a descrição do sistema diferentes níveis de abstração, iniciando pelo sistema como um todo e chegando até as ções em si. Esta descrição facilita o entendimento do sistema pelos seus senvolvedores. Entretanto, num projeto onde eles são muitos, como era o caso do AIDA, extremamente importante que este entendimento seja o mesmo para todos. Isto não é antido apenas pela adoção da OMT, que apesar de prover um certo formalismo ao cesso de produção, permite a ocorrência de ambigüidades. Assim, a utilização de uma juagem não ambígua na descrição do sistema, pode levar a um entendimento único do tema, facilitando a comunicação entre os desenvolvedores.

Os modelos formais, por utilizarem uma notação em termos matemáticos, consistem linguagens não ambíguas. A sua utilização, combinada a uma metodologia de senvolvimento de software, visa garantir o desenvolvimento adequado de um software no o AIDA. Além disso, uma especificação formal, por ser uma linguagem com semântica intaxe bem definidas, permite que a descrição do sistema seja validada, possibilitando a ntificação de erros ainda na fase de análise [Gehani, 1982]. Isto constitui um ganho para projetos, já que a identificação de erros apenas na fase de implementação, implica num pacto maior para o desenvolvimento.

Dentre as diversas metodologias existentes, uma das mais difundidas é a técnica de análise estruturada, que dá ênfase à transformação dos dados. Entretanto, apesar desta técnica levar a uma boa documentação do sistema, o fato de centrar-se nos processos e não nos dados, acaba por tornar a evolução dos produtos uma atividade trabalhosa. A cada mudança de requisitos, que geralmente dizem respeito a novos procedimentos ou à modificação dos antigos, as alterações dos programas se tornam, na maioria das vezes, imensas.

A técnica de orientação a objetos procura resolver este problema, centrando-se nos objetos que irão compor o sistema. Um objeto é definido como um conceito ou uma abstração de alguma coisa, permitindo um mapeamento direto dos elementos que compõem o sistema no mundo real com os componentes da aplicação, e oferecendo uma base real para a sua implementação nos computadores. A definição de um sistema em termos de objetos levando-se em conta conceitos de encapsulamento, abstração, modularidade e herança [Booch, 1991], torna a atividade futura de manutenção menos complexa, além de proporcionar ganhos de produtividade através do reuso de software.

Uma metodologia de desenvolvimento orientada a objetos enfatiza a identificação e a organização dos conceitos, ou entidades, do sistema em questão, no lugar da sua representação final em linguagem de programação [Rumbaugh et al., 1991]. Com isso, torna-se mais fácil introduzir as alterações nos processos, uma vez que geralmente elas ficam restritas apenas a alguns objetos. Uma outra vantagem, é que, por permitir uma representação direta do sistema no mundo real, facilita o seu entendimento pelo usuário, e mesmo por uma equipe inteira de desenvolvimento.

Existem diversas metodologias de desenvolvimento de software baseadas em objetos, como por exemplo as apresentadas por [Meyer, 1988] e [Booch, 1991], entre outras. A utilizada neste trabalho é a Object Modeling Technique - OMT, proposta por Rumbaugh [Rumbaugh et al., 1991]. Esta metodologia utiliza três tipos diferentes de modelos para a descrição do sistema:

- modelo de classes: descreve a estrutura estática do sistema em termos das classes de objetos e seus relacionamentos, capturando do mundo real os conceitos que são importantes para a aplicação;
- modelo dinâmico: representa os aspectos de controle temporal e comportamental dos objetos; utiliza diagramas de estados e de eventos;

- modelo funcional: descreve as transformações dos dados no sistema; contém os Diagramas de Fluxos de Dados - DFDs.

Estes modelos são complementares e devem ser usados durante todos os estágios do processo de desenvolvimento numa forma evolutiva. Uma descrição completa do sistema requer o seu uso em conjunto.

2.5.1.1. O modelo de classes em OMT

Na técnica de orientação a objetos, o modelo de classes é o modelo mais importante, descrevendo as classes dos objetos, seus relacionamentos, seus atributos e suas operações.

Uma *classe* descreve um grupo de objetos com propriedades semelhantes (atributos), com o mesmo comportamento (operações) e mesmo relacionamento com outros objetos, e com a mesma semântica. As *classes dinâmicas* são aquelas que exercem alguma ação sobre outra classe, e as *passivas* são aquelas que apenas sofrem uma ação.

Os *atributos* são valores de dados guardados pelos objetos de uma classe, com um nome único. Uma *operação* é uma função ou transformação que pode ser aplicada a objetos ou por estes a uma classe. Um *método* é a implementação de uma operação. Classes diferentes podem apresentar operações com o mesmo nome. Esta característica denomina-se *polimorfismo*.

A Figura 2.3 apresenta a notação gráfica utilizada para representar uma classe de nome *Classe*, onde *atrib1* e *atrib2* correspondem aos seus atributos, e *op1* e *op2* às suas operações, sendo que *op2* apresenta um argumento de entrada (*arg1*) que é do tipo *tipo1*, e tem como valor de retorno *result*.

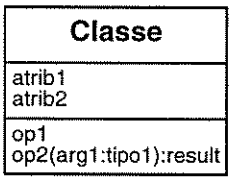


Figura 2.3 - Representação gráfica de classe em OMT

Um *relacionamento* representa uma associação (ligação) entre duas classes.

Uma *associação* pode envolver um ou mais objetos de uma classe, e esta informação deve estar presente no modelo. As associações podem ser *agregação*, quando uma classe é composta por outras; *herança*, quando uma classe herda características de outra, acrescentando algumas mais; ou podem apenas indicar algum tipo de relacionamento

ou dependência entre as classes. Neste caso podem ser utilizadas descrições das associações e das funções de cada classe, facilitando o entendimento das relações existentes.

As associações e as agregações podem apresentar *multiplicidade*, especificando quantas instâncias de uma classe podem estar relacionadas a uma instância de uma classe associada. A Figura 2.4 apresenta os tipos de multiplicidade possíveis.

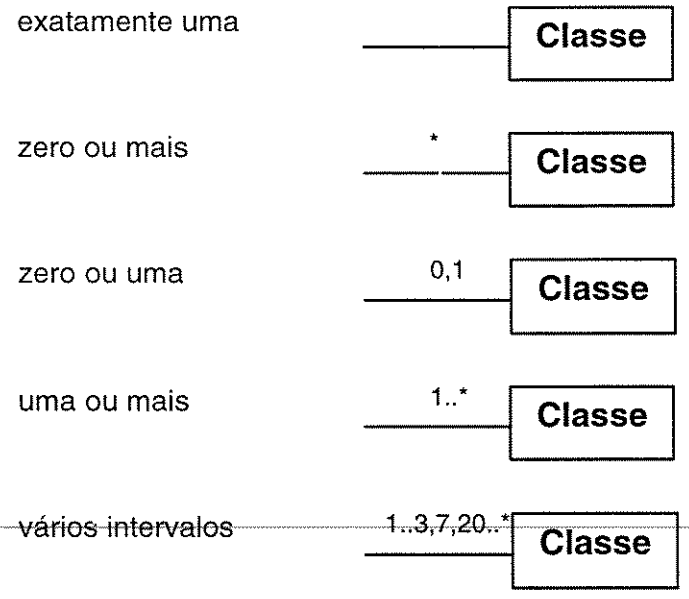
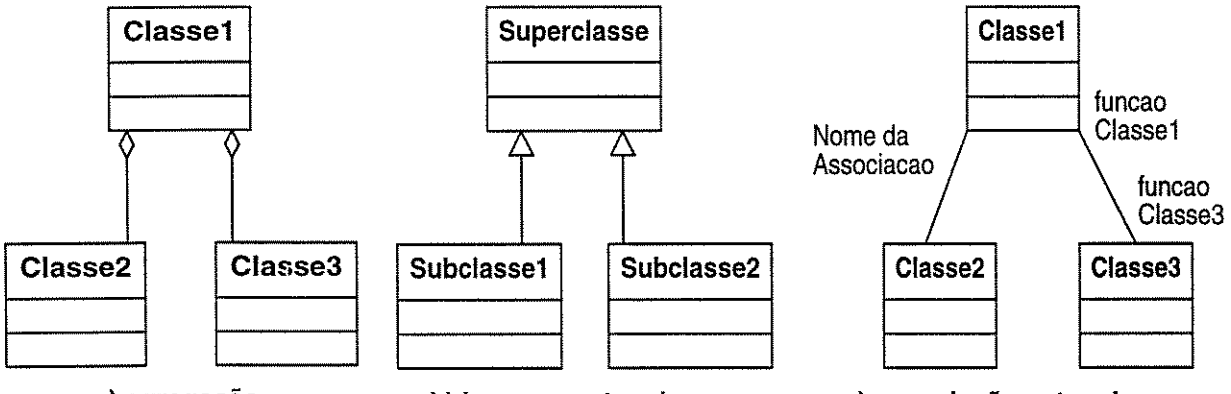


Figura 2.4 - Tipos de multiplicidade em OMT

A herança é uma das associações mais utilizadas, levando ao reuso de classes. Uma classe que herda propriedades de outra, consiste numa *especialização* daquela classe.

São adotados símbolos específicos para cada um dos tipos de associação, os quais podem ser vistos na Figura 2.5.



a) agregação **b) herança entre classes** **c) associação entre classes**

Figura 2.5 - Representação gráfica de associações em OMT

Para facilitar o gerenciamento da complexidade das classes, o sistema pode ser dividido em *módulos* que correspondem a uma unidade lógica.

A elaboração do modelo de classes proposto pela OMT parte de uma descrição informal do sistema, contendo o máximo de detalhes possível. Com base nesta descrição, é feita a identificação das classes que irão compor o software, com a descrição do seu escopo, de seus requisitos e de suas restrições. Em seguida são identificadas as associações entre as classes e os seus atributos. Durante todo este processo deve-se explorar ao máximo os conceitos de reuso e de herança de classes, procurando-se elaborar um modelo enxuto e conciso.

2.5.2. A linguagem de especificação formal SDL

A linguagem formal Specification and Description Language - SDL [ITU-T, 1993a], começou a ser desenvolvida em 1972 pela CCITT, hoje ITU-T para a especificação e a descrição de sistemas de telecomunicações. Sua primeira versão foi lançada em 1976, seguida por novas versões em 1980, 1984, 1988 e 1992 [Belina et al., 1991]. Sua versão mais recente, a SDL-92, incorporou conceitos de objetos, como tipos, reuso e herança, ampliando a sua abrangência, com a tendência de ser utilizada na especificação dos mais diversos produtos de diferentes áreas, principalmente em sistemas de tempo real, interativos ou distribuídos. Esta tese, e os artigos técnicos produzidos [Macário, et al., 1997a] e [Macário, et al., 1997b], constituem exemplos de uma nova utilização desta linguagem no desenvolvimento de ambiente de software fora da área de telecomunicações.

A linguagem SDL serve tanto para representar o comportamento do sistema, como sua estrutura, usando diferentes níveis de abstração, desde o nível mais alto até o mais detalhado. Uma de suas características fortes é que, diferente da maioria das linguagens de especificação formal que apresentam apenas uma representação textual, a SDL possui também uma representação gráfica, denominada SDL-GR, o que facilita bastante o seu entendimento pelo usuário, já que a descrição do sistema é feita numa maneira muito próxima àquela a que se está acostumado. Além disso, por ser uma linguagem formal, possui uma sintaxe e uma semântica bem definidas, possibilitando a simulação e a validação automática do sistema sendo especificado, mesmo nas fases iniciais do desenvolvimento, através de uma ferramenta CASE como o ambiente SDT[TELELOGIC AB, 1995a][TELELOGIC AB, 1996c][TELELOGIC AB, 1997].

A especificação de um software em SDL apresenta uma descrição geral do sistema, que é feita utilizando-se blocos, canais e tipos abstratos de dados. A especificação dos blocos se dá através de processos e de sinais. Por fim, a especificação dos processos descreve o comportamento do sistema, em termos de estados e transições, podendo ser utilizados também procedimentos. Neste nível são definidas as variáveis necessárias e os sinais são apresentados já com os seus parâmetros, caso existam. Dependendo do nível de abstração utilizado e da fase em que o desenvolvimento se encontra, as tarefas podem ser descritas através de um texto informal, ou chegar ao detalhamento da manipulação de dados.

2.5.2.1. SDL-92

A versão mais recente da linguagem SDL, a SDL-92 já prestes a ser substituída pela SDL-96 [Reed, 1997], passou a incorporar conceitos de objetos, que vem a ser a sua grande diferença em relação às versões anteriores, e a ser mantida e evoluída em versões futuras, permitindo dentre outras coisas, o reuso das especificações. A partir desta versão existem duas formas de declarações: tipos e instâncias. Os tipos definem um grupo de entidades (sistemas, blocos ou processos) com propriedades semelhantes e mesmo comportamento, e a sua utilização na descrição dos sistemas se dá através das instâncias. As principais vantagens da utilização de tipos são permitir a criação de tantas instâncias quantas forem necessárias, evitando a repetição das especificações, como acontecia com a versão de 1988, e principalmente tornar possível o uso de conceitos de herança, especialização e reuso, comuns a qualquer metodologia orientada a objetos [Færgemand et. al. 1997][TELELOGIC AB, 1995b].

Na especificação das instâncias dos processos que compõem um bloco, devem ser definidos o número inicial e o número máximo de instâncias que cada processo poderá apresentar simultaneamente durante uma execução do sistema. Estes números são indicados após o nome da instância. Durante a execução do sistema, cada uma das instâncias de processos apresentará um identificador único, do tipo Pid.

Além da definição de tipos e instâncias, outras funcionalidades foram incorporadas à linguagem, visando o uso de conceitos de orientação a objetos. Dentre elas destacam-se:

- mecanismo de PACKAGE, que permite reusar todas as definições de um sistema já especificado, ou apenas algumas delas, num propósito similar às bibliotecas de funções. A inclusão de um package num sistema é feita através da cláusula USE;

- construção `INHERITS`, que indica que o sistema herda as definições de um outro sistema, e que apenas as diferenças entre eles são especificadas;
- construção `REDEFINED`, que indica que a especificação apresentada de um bloco ou processo é uma redefinição de um bloco ou processo do sistema herdado. Como existe uma hierarquia de definições, uma *redefined procedure* é descrita em um *redefined process type*, que por sua vez faz parte de um *redefined block type*. Para que um *block type* ou um *process type* possa ser redefinido, o mesmo deve ser declarado como `VIRTUAL` na sua especificação original (*virtual block type* ou *virtual process type*). Em especial nos processos, as transições a serem redefinidas também devem ser definidas originalmente como `VIRTUAL`.

O Apêndice A apresenta detalhes das principais construções SDL, e mais algumas construções presentes na versão SDL-92.

2.5.3. O uso combinado da técnica OMT com a linguagem formal SDL-92

A metodologia alternativa proposta neste trabalho para o desenvolvimento do AIDA, apresenta o uso combinado da técnica OMT com a linguagem formal SDL-92, doravante denominada SDL, buscando prover um certo formalismo ao seu desenvolvimento. Para tanto, utiliza-se o modelo de classes proposto pela OMT nas fases de análise de requisitos e análise do sistema, e a linguagem SDL nas fases de desenho do sistema e dos objetos. O seu uso combinado, dentre outras coisas induz ao reuso, à herança e à modularização, possibilitando também a validação e a simulação do sistema sendo especificado ainda nas fases iniciais do processo de desenvolvimento [Macário et al., 1997a][Macário, et al., 1997b].

O modelo de classes é usado para descrever o sistema e os seus elementos externos, bem como para descrever a sua arquitetura, mapeando o sistema em termos de objetos do mundo real, o que torna mais fácil a sua definição. O modelo contendo a sua arquitetura é gerado a partir de uma descrição informal do sistema, devendo ser identificadas suas classes, seus requisitos e suas restrições, seus atributos e seus relacionamentos. Este modelo deve evoluir durante o processo de desenvolvimento, passando a incluir os detalhes de implementação.

A partir do modelo de classes, passa-se à elaboração da especificação do sistema em SDL, definindo-se a estrutura de implementação do sistema e o comportamento de cada

classe. Com a existência na linguagem SDL de conceitos de tipos, instâncias, reuso e herança, o mapeamento do modelo OMT para a especificação SDL torna-se uma atividade simples e quase automática. Além disso, a possibilidade de especificar o sistema com diferentes níveis de abstração, permite a sua utilização em diversas fases do processo de desenvolvimento, chegando-se até a sua implementação.

A Tabela 2.1 apresenta o mapeamento genérico de OMT para SDL:

Modelo de Classes	Representação SDL
Sistema	System type
Módulos	Block type
Classes dinâmicas	Process type
Classes passivas	SORTS (tipos abstratos de dados em SDL)
Objetos das classes dinâmicas	instâncias de cada process type
Objetos das classes passivas	instâncias de SORTS, definidas através de variáveis nos process types

Tabela 2.1 - Mapeamento de OMT para SDL

Os capítulos 3 e 4 apresentam a aplicação desta metodologia no desenvolvimento e evolução do AIDA, partindo de uma versão centralizada até uma versão distribuída, executando em ambiente CORBA.

3. AIDA Centralizado e Concorrente usando OMT e SDL

3.1. Introdução

Este capítulo apresenta a utilização da metodologia proposta no capítulo 2, que consiste em uma combinação da técnica orientada a objetos OMT com a linguagem de especificação formal SDL, para o desenvolvimento do AIDA e a sua evolução de uma versão centralizada para uma versão concorrente, onde vários usuários podem executar várias ferramentas de análise ao mesmo tempo em um ambiente compartilhado como em uma rede de PCs ou estações de trabalho. Além disso, apresenta as funcionalidades básicas do AIDA, mostrando como elas foram mapeadas em OMT e SDL-92, e a utilização de construções que permitem o reuso e a evolução de sistemas.

Fazendo uso do ambiente SDL Design Tool (SDT), são apresentados os resultados das validações das duas versões, com exemplos de falhas detectadas nas especificações e suas conseqüentes correções. Ao final do capítulo alguns exemplos de simulação serão comentados, com a apresentação de um diagrama MSC (Message Sequence Chart) relativo a uma simulação de uma configuração do AIDA concorrente.

3.2. Modelo OMT do AIDA Centralizado

O modelo de classes apresentado na Figura 3.1 consiste numa evolução do proposto em [Macário, et al., 1997a], onde ainda não se previa a sua evolução para um ambiente concorrente. Este modelo é o resultado da análise do software, representando a estrutura geral do AIDA, num alto nível de abstração. Na verdade, cada uma das classes apresentadas poderiam ser descritas em outros diagramas. Por exemplo, as classes *EInterf* (Interface geral do ambiente), *TInterf* (Interface da ferramenta de análise e do formatador de relatórios) e *FCInterf* (Interface do conversor de formatos) são especializações de uma classe genérica *Interface*, não representada neste diagrama, da qual herdam características comuns.

Durante a análise do sistema, foram identificadas as classes principais do AIDA para representarem as suas funcionalidades mais importantes: gerenciamento do ambiente, ferramentas para análise matemática e estatística, formatador de relatórios e conversor de arquivos.

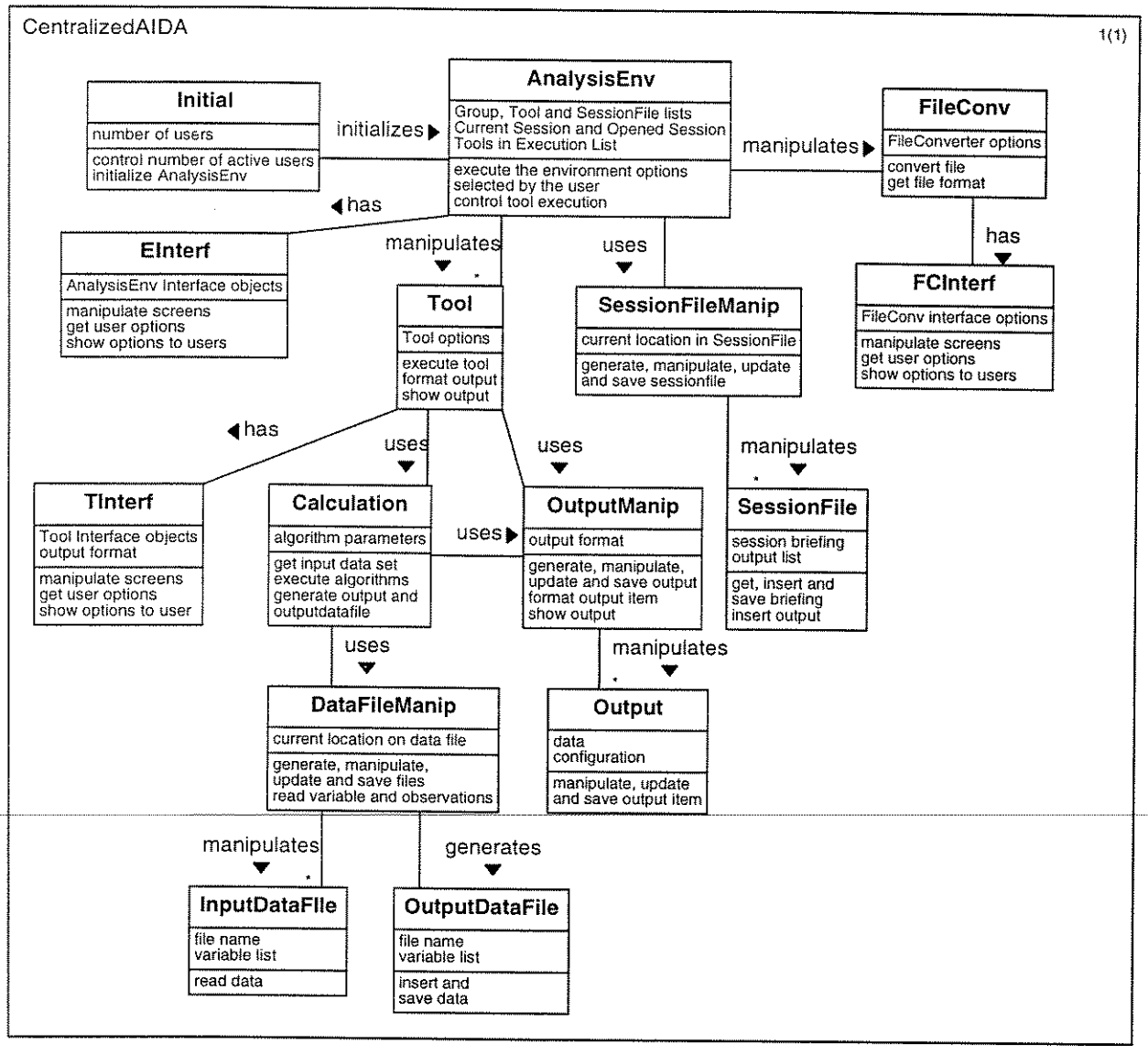


Figura 3.1- Modelo OMT do CentralizedAIDA

Para facilitar o entendimento do sistema, bem como diminuir a sua complexidade, o modelo OMT do AIDA foi divido em três módulos que agrupam as classes logicamente segundo suas funcionalidades. O módulo *AnalysisEnv* é formado pelas classes *Initial*, *AnalysisEnv*, *EInterf*, *SessionFileManip* e *SessionFile* utilizadas para o gerenciamento do sistema AIDA como um todo. O módulo *Tool* agrupa as classes *Tool*, *TInterf*, *Calculation*, *OutpManip*, *Output*, *DataFileManip*, *InputDataFile* e *OutputDataFile* responsáveis pela execução de uma ferramenta de análise e formatação de relatório. O módulo *FileConv* apresenta as classes para conversão de arquivos de dados externos ao AIDA.

A classe *Initial*, que não existia na primeira versão do sistema centralizado [Macário, et al., 1997a], inicia o ambiente AIDA e controla o número de usuários executando o

sistema. Esta classe é responsável por instanciar a classe *AnalysisEnv* (Gerenciador do ambiente de análise) para cada usuário que iniciar a execução do ambiente. Na versão centralizada, esta classe permite o uso do sistema por apenas uma pessoa por vez.

A classe *AnalysisEnv* possui relacionamentos com outras quatro classes: *EInterf*, *Tool* (Ferramenta de análise e formatador de relatórios), *SessionFileManip* (Manipulador do arquivo de sessão de trabalho) e *FileConv* (Conversor de arquivos externos). Ela é responsável por gerenciar, para cada usuário, a execução das ferramentas e a formatação de relatórios. Além disso, também controla as sessões de trabalho do usuário, e as opções selecionadas por ele. A interação entre o usuário e o gerenciador do ambiente de análise do AIDA é feita através da classe *EInterf*, que controla todos os elementos da interface do ambiente de análise. A classe *SessionFileManip* possui as funções específicas para a manipulação dos arquivos de sessões (*SessionFile*).

A classe *FileConv*, que realiza a conversão dos arquivos externos para o formato do ambiente AIDA, e *FCInterf*, que faz a interação do conversor com o usuário.

A classe *Tool* representa uma especificação genérica para todas ferramentas de análise incorporadas ao ambiente AIDA. Esta classe é responsável tanto pelo gerenciamento da execução da ferramenta em si, como da formatação de um relatório que foi gerado por ela, já que apenas ela conhece o seu formato. Esta classe usa três outras classes: *TInterf*, que representa a interação entre o usuário e a ferramenta ou com o formatador de relatórios; *OutputManip* (Manipulador do relatório), para a formatação de um relatório (*Output*); e *Calculation* (Cálculo dos algoritmos), que executa algoritmos específicos à ferramenta, acessando os arquivos de dados, se necessário. A classe *Calculation* também se relaciona com a classe *OutputManip* na medida em que gera relatórios ao final da execução dos algoritmos. A classe *DataFileManip* manipula e gera arquivos de dados de entrada (*InputDataFile*) e saída (*OuputDataFile*), respectivamente.

3.3. Especificação SDL do AIDA Centralizado

A especificação SDL de um sistema pode ser elaborada através de um mapeamento direto do seu modelo OMT. A definição do sistema é feita através de um *system*, ou de um *system type,* em termos de seus módulos, que em SDL são mapeados como *block types.* Cada bloco é formado por um ou mais *process type*, usado para representar cada classe dinâmica identificada no modelo de classes. O mapeamento OMT para SDL da versão centralizada é apresentado na Tabela 3.1.

O *system type CentralizedAIDA* define o sistema AIDA em si, representando o modelo de classe da Figura 3.1. Os blocos *AnalysisEnv*, *Tool* e *FileConv* correspondem aos módulos lógicos criados no modelo OMT para agrupar classes relacionadas, e os processos existentes em cada um destes blocos representam as funcionalidades das classes dinâmicas identificadas.

Classes OMT	Representação em SDL
AIDA	System Type CentralizedAIDA, composto pelos block types AnalysisEnv, FileConv e Tool
Módulo AnalysisEnv	Block Type AnalysisEnv, composto pelos process types Initial, AnalysisEnv e EInterf
Módulo FileConv	Block type FileConv, composto pelos process types FileConv e FCInterf
Módulo Tool	Block type Tool, composto pelos process types Tool, TInterf e Calculation
Output, SessionFile, DataFile	Classes passivas a serem representadas por SORTS
OutputManip, SessionFileManip, DataFileManip	Classes a serem representadas como operações sobre os SORTS

Tabela 3.1 - Mapeamento de OMT para SDL do sistema AIDA Centralizado

Como neste processo de desenvolvimento tinha-se em mente a evolução do sistema AIDA centralizado para um ambiente concorrente, bem como a geração de elementos reusáveis para este ou para outros sistemas, optou-se por utilizar estruturas *system type*, *block type* e *process type* para representar as classes identificadas para o AIDA (Figura 3.1).

Também foram usadas outras das funcionalidades presentes na linguagem SDL-92, como o mecanismo de `PACKAGE`, que permite reusar integralmente as definições de um sistema já especificado, ou partes dele. Neste sentido, na especialização de sistemas, é necessário apenas a evolução de alguns dos processos existentes, através da sua redefinição, e em alguns casos, a inclusão de outros. Para que um *block type* ou um *process type* possa ser redefinido, o mesmo deve ser declarado como `VIRTUAL` (*virtual block type* ou *virtual process type*) na sua especificação original a ser herdada. Em especial

nos processos, as transições a serem redefinidas também devem ser definidas originalmente como `VIRTUAL`. Assim, para que o sistema centralizado pudesse ser reusado e evoluído para a especificação do sistema concorrente, foi criado o package *Centralized* (Figura 3.2), que contém todas as definições do *system type CentralizedAIDA*. Como algumas destas definições serão redefinidas na evolução do sistema, muitos dos diagramas apresentados nesta sessão apresentam a construção `VIRTUAL`, com exceção do bloco *FileConverter*, pois na evolução para a versão concorrente ele se manteve inalterado, sem ter sido necessária a sua redefinição.

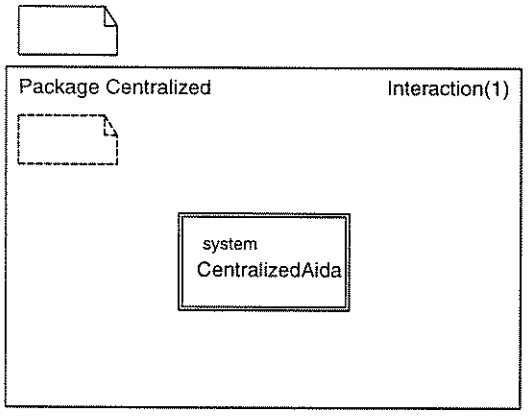


Figura 3.2 - Package Centralized

O system type *CentralizedAIDA* (Figura 3.3), é composto por três block types, *AnalysisEnv*, *Tool* e *FileConv* representados por figuras com bordas duplas, e cujas instâncias (figuras com borda simples) apresentam os respectivos nomes: *AE*, *T* e *FC*. Os *block types* agrupam classes funcionalmente relacionadas, e a interação entre elas é representada pelas suas instâncias. A comunicação entre as instâncias dos blocos ou entre elas e o ambiente externo ao sistema é feita através de canais de sinais, onde são indicadas as trocas de mensagens (sinais). Os *gates* são usados para conectar os blocos a estes canais. Por exemplo, a instância *AE* do block type *AnalysisEnv* se comunica com a instância *FC* do block type *FileConv* pelo canal *AEFC*, conectando-se a este canal através do *gate C*.

No nível do sistema são definidos todos os sinais e estruturas de dados que são visíveis aos blocos. Na definição dos sinais, através do comando `SIGNAL`, devem ser especificados os tipos dos parâmetros que eles carregam, como o sinal *ExeTool*, que possui os parâmetros *Pid*, *Integer*, *Charstring* e *Pid*. A construção `SIGNALLIST` define listas de sinais que são transportados pelos canais, facilitando a definição do sistema. No system

type *CentralizedAIDA* foram definidas as listas de sinal L1, L2, L3, L4 e L5. Todos os sinais especificados nestas listas já devem ter sido declarados.

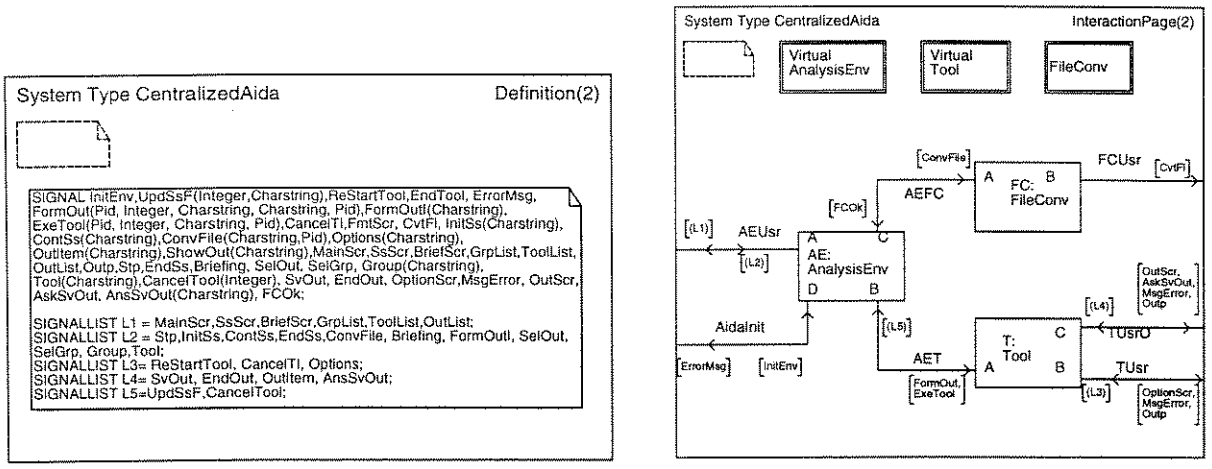


Figura 3.3- System Type CentralizedAida

A especificação dos blocos é feita através dos *process types*, que representam as classes dinâmicas no modelo OMT. O relacionamento destes processos é descrito por suas instâncias, que comunicam-se entre si e com os canais do sistema através das rotas de sinais. A conexão dos processos a estas rotas se dá pelos gates.

O block type *AnalysisEnv* (Gerenciador do AIDA - Figura 3.4) é composto por três process types: *Initial*, *AnalysisEnv* e *EInterf*, cuja interação é descrita através das instâncias *Init*, *AE* e *EI*, onde por exemplo, a instância *EI* comunica-se com a instância *AE* pela rota *AEEI* e com o *gate A* do bloco pela rota *EIUsr*. Durante a execução do sistema, é possível a criação de tantas instâncias do *process types* quantas forem necessárias. O número inicial de instâncias de um processo, e o número máximo que poderão ser criadas simultaneamente durante uma execução devem ser definidos ao lado do seu nome. Desta forma, a declaração *Init(1,1)* na especificação deste bloco, indica que existirá no início da execução do sistema uma instância *Init* do processo *Initial*, não sendo permitida a criação simultânea de nenhuma outra instância deste processo durante esta execução. Todos os sinais, listas de sinais e tipos de dados usados neste nível e que não tenham sido declarados no nível do sistema, devem ser especificados. Por exemplo, o sinal *QuitScrOk*, na rota *AEEI*, que é um sinal local ao bloco, deve ser declarado no bloco. As classes *OutpManip*, *Output*, *SessionFileManip* e *SessionFile* não foram mapeadas no bloco e são representadas no processo *AnalysisEnv* através de tarefas como por exemplo "*update session file*".

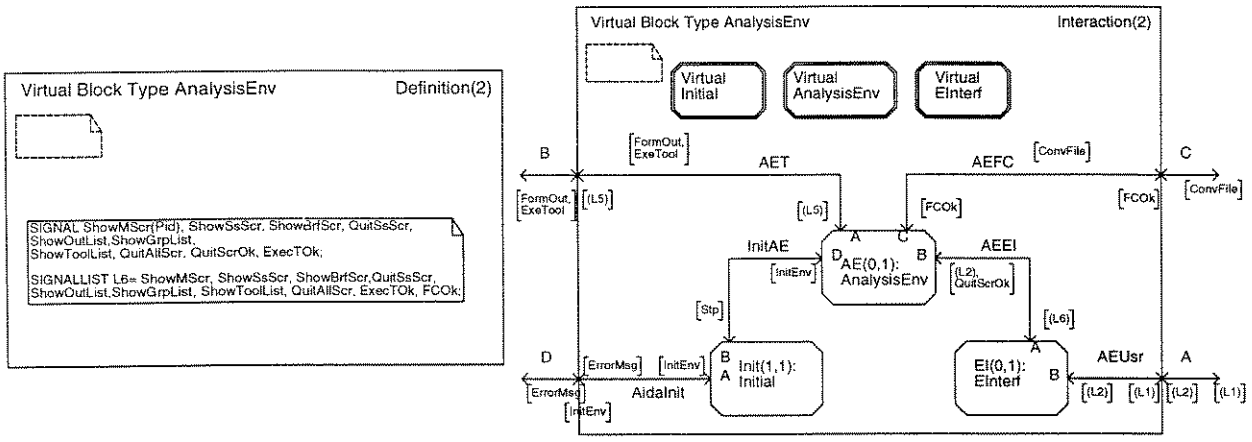


Figura 3.4- Virtual Block Type AnalysisEnv / CentralizedAida

O bloco *Tool* (Execução de ferramenta de análise e Formatação de relatório - Figura 3.5) é composto pelos processos *Tool*, *TInterf* e *Calculation*, que apresentam as mesmas funcionalidades descritas pelas suas respectivas classes no modelo OMT (Figura 3.1). Suas instâncias *T*, *TI* e *Calc*, comunicam-se através das rotas *AET*, *TCalc*, *TTIO*, *TTI*, *TIUsr* e *TIUsrO*, e no início da execução do sistema existirá uma instância *T* e nenhuma instância *TI* e *Calc*, não podendo ser criada simultaneamente mais que uma de cada.

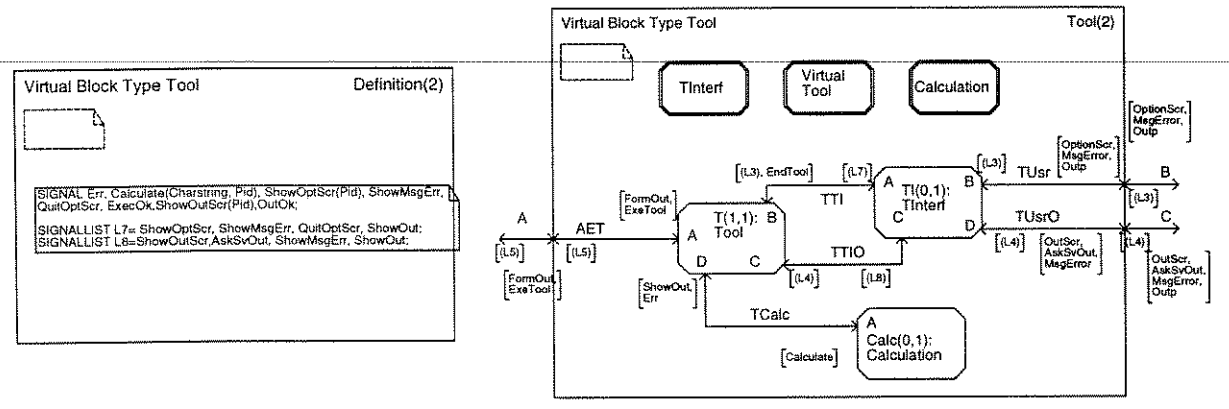


Figura 3.5- Virtual Block Type Tool / CentralizedAida

O bloco *FileConv* (Conversor de formatos - Figura 3.6), contém os processos *FileConv* e *FCInterf* para conversão de arquivos, cujas instâncias são *FC* e *FCI* respectivamente. Este bloco apresenta os sinais *ShowConvFile* e *EndFC* que são locais e por esta razão são declarados apenas neste nível. Os demais sinais utilizados no bloco já foram definidos no nível do sistema (Figura 3.3).

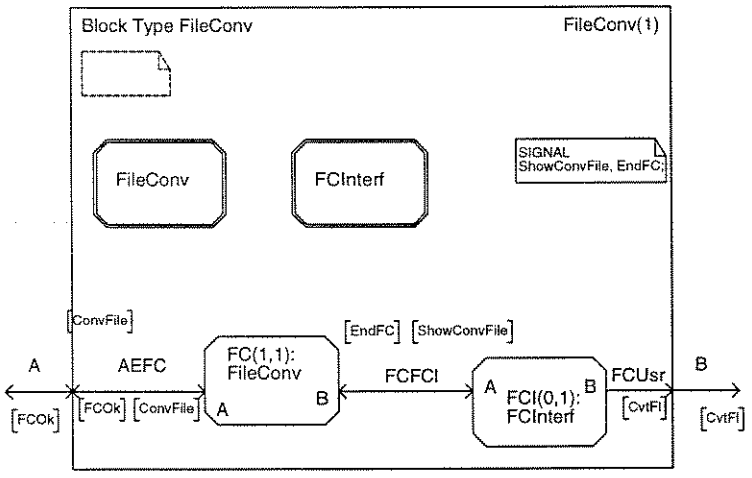


Figura 3.6 - Block Type FileConv / CentralizedAida

Uma especificação de processos em SDL representa principalmente a troca de mensagens e o seu comportamento. Sem perda de generalidades, as tarefas, neste e nos outros processos do AIDA, foram abstraídas e representadas em SDL pelo símbolo `TASK`, com um texto descrevendo a ação correspondente. Por exemplo: "*quit all executions*" e "*update executing tools table*", entre outras na Figura 3.7.

- **Process type AnalysisEnv (Figura 3.7) / Bloco AnalysisEnv (Figura 3.4)**

O process type *AnalysisEnv* é responsável pelo gerenciamento da execução das ferramentas, da formatação de relatórios e do conversor de formatos, controlando também as sessões de trabalho. Além disso, é ele quem cria uma instância *EI* do processo *EInterf*, usando para isso a construção `CREATE REQUEST`. Isto ocorre quando ele se encontra no estado *idle* e recebe o sinal *InitEnv*, que indica a iniciação do ambiente por um usuário. Uma vez iniciado o AIDA, o usuário pode sair do ambiente (sinal *Stp*), iniciar uma nova sessão de trabalho (sinal *InitSs*) ou continuar executando uma sessão existente (sinal *ContSs*), ou converter um arquivo externo (sinal *ConvFile*). Uma requisição de execução das ferramentas de análise (sinal *SelGrp*) ou de formatação de um relatório (sinal *SelOut*) só pode ser feita quando uma sessão de trabalho tiver sido iniciada. O recebimento dos sinais *UpdSsF* ou *CancelTool* indicam o término da execução requerida pelo usuário, permitindo que uma nova ferramenta de análise ou um novo relatório sejam selecionados para execução. Na definição de um processo devem ser especificados todos os seus gates e os sinais que entram e saem por eles. Por exemplo, o processo *AnalysisEnv* na definição do block type *AnalysisEnv* (Figura 3.4), apresenta quatro gates *A*, *B*, *C* e *D*, que o conectam às rotas de sinais *AET*, *AEEI*, *AEFC* e *InitAE* respectivamente.

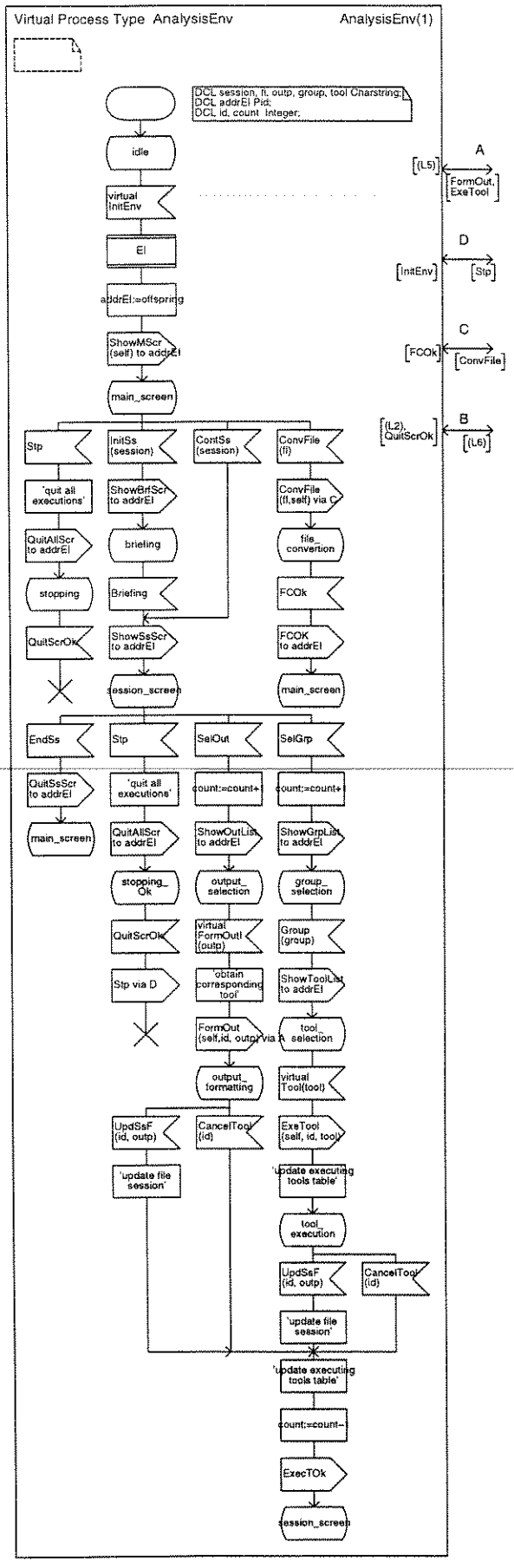


Figura 3.7 - Virtual Process Type AnalysisEnv

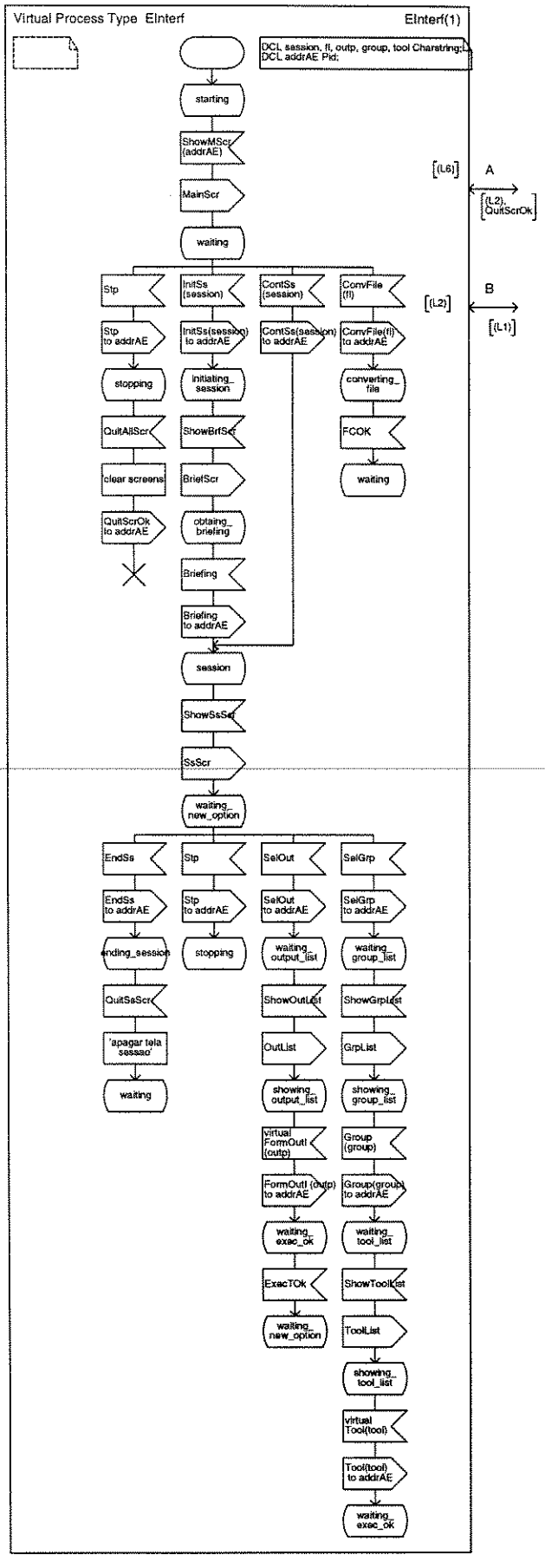


Figura 3.8 - Virtual Process Type EInterf

Na especificação do processo, o *gate A* do tem como entrada os sinais especificados na lista de sinais *L5*, e como saída os sinais *FormOut* e *ExeTool (*rota *AET* do bloco *AnalysisEnv).*

- **Process type EInterf (Figura 3.8) / Bloco AnalysisEnv (Figura 3.4)**

O processo *EInterf* é responsável pela interface entre o usuário e o processo *AnalysisEnv.* No início de sua execução, encontra-se no estado *starting,* e ao recebimento da requisição para exibir a tela principal do sistema (sinal *ShowMScr(addrAE)*), exibe a tela principal para o usuário (sinal *MainScr*), exibindo as opções de disponíveis. Quando uma nova sessão de trabalho é iniciada, o usuário pode fornecer um resumo do que será feito durante esta sessão (sinal *Briefing* - estado *obtaining_briefing*), e após iniciada a sessão (nova ou não), a interface exibe a tela de sessão (sinal *SsScr*) e vai para o estado *waiting_new_option,* permitindo ao usuário selecionar um relatório para formatação (sinal *SelOut to addrAE*), ou um grupo de ferramentas de análise para execução (sinal *SelGrp to addrAE).* A construção `TO` no sinal de saída é usada para identificar a instância de processo que vai receber o sinal. Assim, a opção selecionada é enviada ao processo *AnalysisEnv*, identificado por *addrAE*, através dos sinais correspondentes. Na execução de uma ferramenta (sinal *Tool*) ou formatação de relatório (sinal *FormOut*), o processo vai para o estado *waiting_exec_ok*, e o recebimento do sinal *ExecTOk*, indica que a execução selecionada terminou, o processo volta para o estado *waiting_new_option*, possibilitando ao usuário selecionar uma nova opção.

- **Process type Initial (Figura 3.9) / Bloco AnalysisEnv (Figura 3.4)**

O processo *Initial*, que não existia na primeira especificação do sistema centralizado [Macário, et al., 1997a], consiste numa das evoluções decorrentes de uma posterior validação do sistema. A identificação da necessidade deste processo é apresentada na seção 3.6.3, que trata alguns erros encontrados na validação do sistema anterior. Este processo inicia o ambiente AIDA e controla o número de usuários executando o sistema, criando uma instância *AE* do processo *AnalysisEnv* (Gerenciador do ambiente de análise) para cada um deles, ao recebimento do sinal *InitEnv.* Como o sistema é mono-usuário, não pode existir mais de uma instância *AE* simultaneamente durante a sua execução. Além disso, este processo só deixa o sistema ser executado por apenas uma pessoa por vez, e portanto, qualquer tentativa de acesso ao sistema será recusada até que o sinal *Stp* seja recebido, indicando que a execução do AIDA foi finalizada. Como na versão concorrente este processo será redefinido para permitir a execução do sistema por mais de um usuário ao mesmo tempo, na versão centralizada ele foi definido como *virtual.*

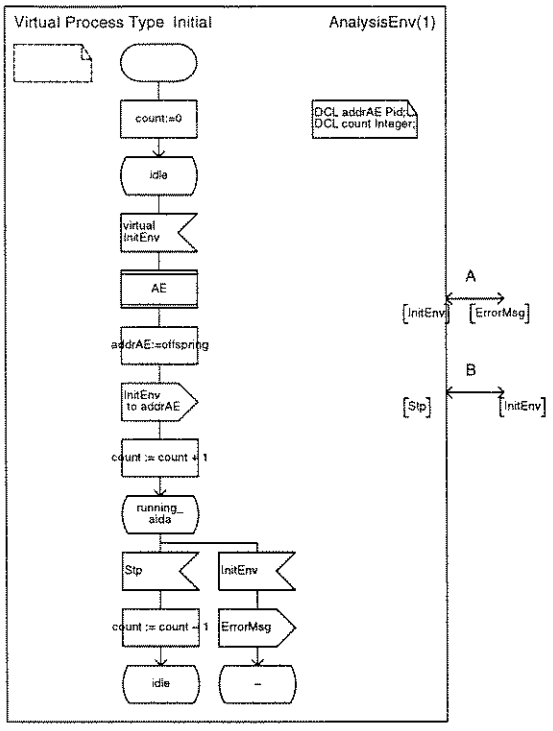


Figura 3.9 - Virtual Process Type Initial

O block type *Tool* (Execução de ferramenta de análise e Formatação de relatório - Figura 3.5) é formado por três process types: *Tool*, *TInterf* e *Calculation*. Na evolução deste sistema para uma versão concorrente, apenas o processo *Tool* precisou ser redefinido, e assim, apenas ele e algumas de suas transições foram definidos originalmente como *virtual*.

- **Process type Tool (Figura 3.10) / Bloco Tool (Figura 3.5)**

O processo *Tool* é responsável pelo gerenciamento da execução da ferramenta análise e da formatação de relatório. A especificação destas funcionalidades é feita em dois diagramas distintos, denominados páginas, que fazem parte do mesmo processo. A página *Tool* apresenta a funcionalidade de gerenciamento da execução de ferramenta, e é acionada com a recepção do sinal *ExeTool(user, id, tool)*, cujos parâmetros indicam o identificador da instância que enviou o sinal, o índice desta ferramenta na tabela de ferramentas em execução mantida pelo processo *AnalysisEnv* e o nome da ferramenta a ser executada. Em seguida cria uma instância *TI* do processo *TInterf* , que especifica sua interface com o usuário, requisitando a ela a exibição da tela de opções de execução (sinal *ShowOptScr*).

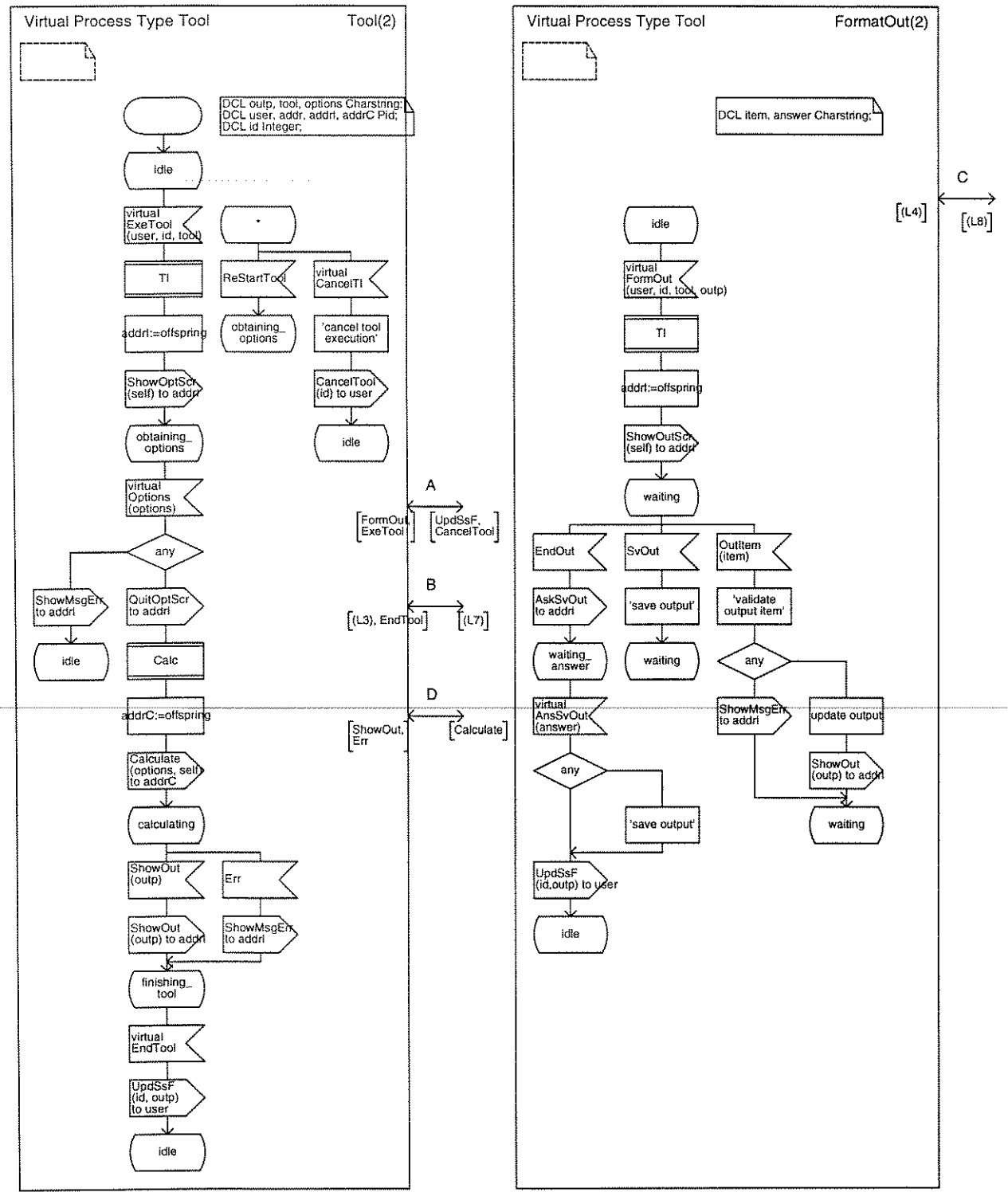


Figura 3.10- Virtual Process Type Tool

No recebimento do sinal *Options(options)* com as opções selecionadas pelo usuário, este processo cria uma instância *Calc* do processo *Calculation* para realizar o cálculo do algoritmo específico da ferramenta de análise invocada, cuja saída será o sinal *ShowOut(outp) to addrI*, solicitando a exibição do relatório de execução pela interface

identificada por *addrI*, ou o sinal *Err*, indicando que ocorreu algum erro de execução do algoritmo durante o cálculo. A segunda página do processo *Tool* (*FormOut* na Figura 3.10) corresponde ao gerenciamento da formatação de relatório que especifica a formatação de um relatório gerado por esta ferramenta. O recebimento do sinal *FormOut(user, id, tool, outp)*, cujos parâmetros indicam o identificador da instância que enviou o sinal, o índice desta ferramenta na tabela de ferramentas em execução mantida pelo processo *AnalysisEnv*, o nome da ferramenta que gerou este relatório e o nome do relatório, leva à criação da instância *TI* do processo *Interf*, responsável também pela interface do formatador de relatórios, que retorna o identificador desta interface (*addrI*). Durante a formatação, o usuário pode formatar um item do relatório (sinal *OutItem*), salvar o relatório (sinal *SvOut*) ou terminar a sua formatação (sinal *EndTool*). Em ambas as páginas aparece o símbolo `any` , usado na construção `DECISION` para permitir uma escolha randômica de um de seus caminhos, durante a execução do processo. Esta construção foi utilizada várias vezes na especificação do sistema para simular os resultados gerados. Por exemplo, na página *Tool*, esta construção foi usada para deixar a cargo das ferramentas utilizadas para a validação e a simulação do sistema AIDA a escolha do caminho a ser testado: a ocorrência, ou não, de erros na validação das opções fornecidas pelo usuário.

- **Process type TInterf (Figura 3.11) / Bloco Tool (Figura 3.5)**

O processo *TInterf* também apresenta duas páginas: uma para a especificação da interface da ferramenta de análise, e outra para a interface do formatador de relatórios. A página *TInterf* apresenta a interface da ferramenta, e sua execução tem início com a recepção do sinal *ShowOptScr(addT)*, que informa o identificador da instância *T* do processo *Tool*, indicando que deve ser exibida a tela de opções(sinal *OptScr*) para que o usuário informe as opções de execução da ferramenta. As opções selecionadas são repassadas para o processo *Tool* - página *Tool* através do sinal *Options(options) to addrT.* Apesar deste ser um sistema mono-usuário, onde apenas uma ferramenta pode ser executada por vez, quando existem sinais iguais de entrada e saída, ou sinais iguais saindo por mais de um gate, deve ser especificado quem recebe o sinal ou o seu gate de saída. Por esta razão é usada a construção *to addrT* no envio de alguns sinais para o processo *Tool.* Após o envio das opções de execução, este processo fica esperando pelo resultado do cálculo do algoritmo (sinal *ShowOut* ou sinal *ShowMsgErr*). O símbolo de estado com um asterisco (*) significa que este processo estando em quaisquer dos seus estados, podem ser recebidos o sinal *ReStartTool*, que indica o reinício da execução da ferramenta, ou o sinal *CancelT*, que cancela a sua execução.

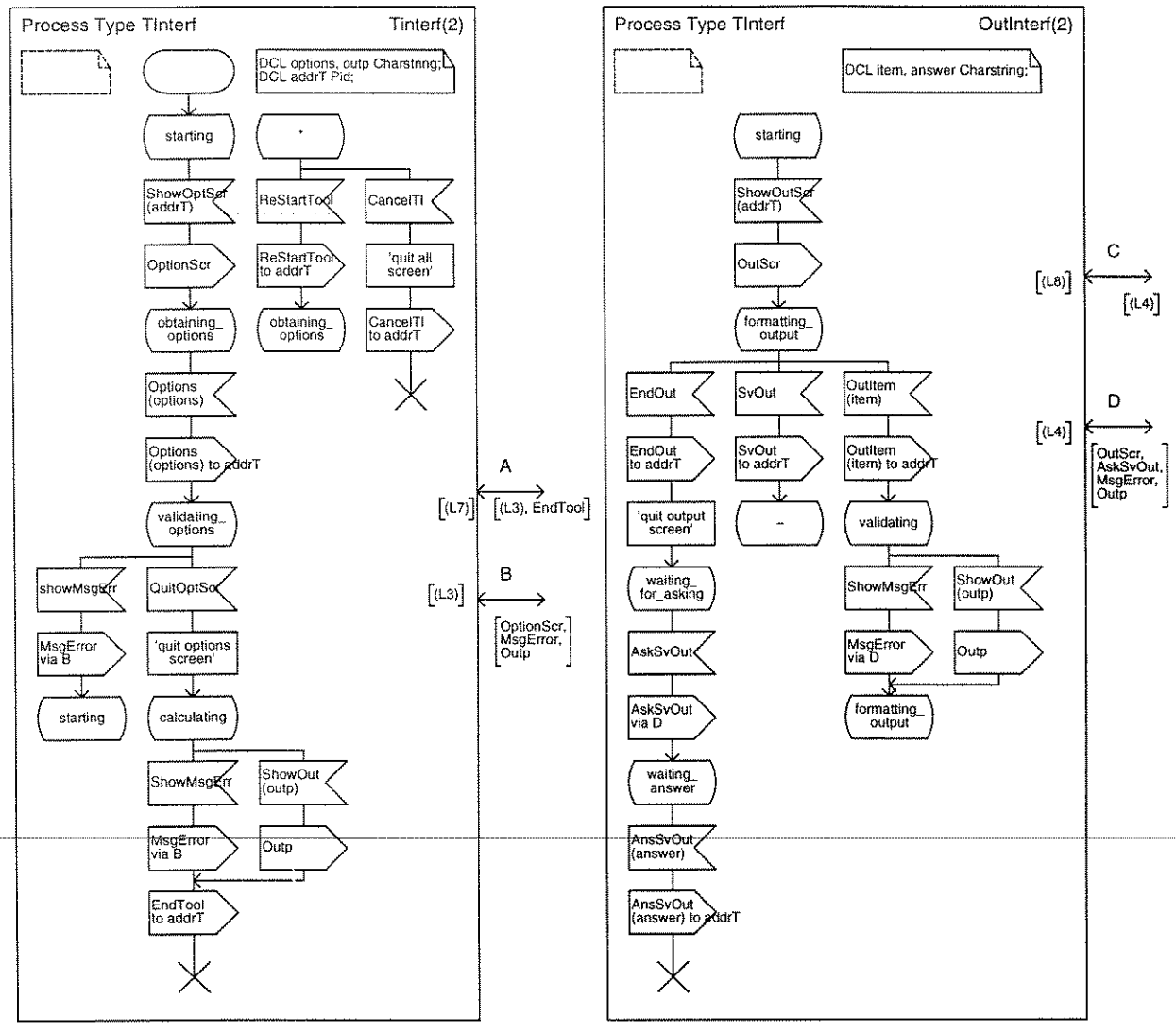


Figura 3.11- Process Type TInterf

A página *OutInterf* especifica a interface do formatador de relatórios, e é iniciada pela recepção do sinal *ShowOutScr(addrT)*, cujo parâmetro informa o identificador da instância *T* do processo *Tool*, solicitando a exibição para o usuário da tela do formatador. Esta tela (sinal *OutScr* enviado para o usuário) tem como opções a formatação de um item de relatório (sinal *OutItem*), salvar o relatório sendo formatado (sinal *SvOut*) ou terminar a sua formatação (sinal *EndOut*). Quando o usuário seleciona a opção para terminar a formatação, ele recebe uma mensagem deste processo (sinal *AskSvOut*) sobre a possibilidade de salvar o seu relatório. Através do sinal *AnsSvOut(answer)*, o usuário informa se quer ou não salvar o relatório, repassando esta resposta ao processo *Tool*, e terminando a execução do processo.

- **Process type Calculation (Figura 3.12) / Bloco Tool (Figura 3.5)**

O processo *Calculation* realiza o cálculo do algoritmo específico à ferramenta de análise, recebendo do processo *Tool* o sinal *Calculate* com as opções de execução selecionadas pelo usuário.

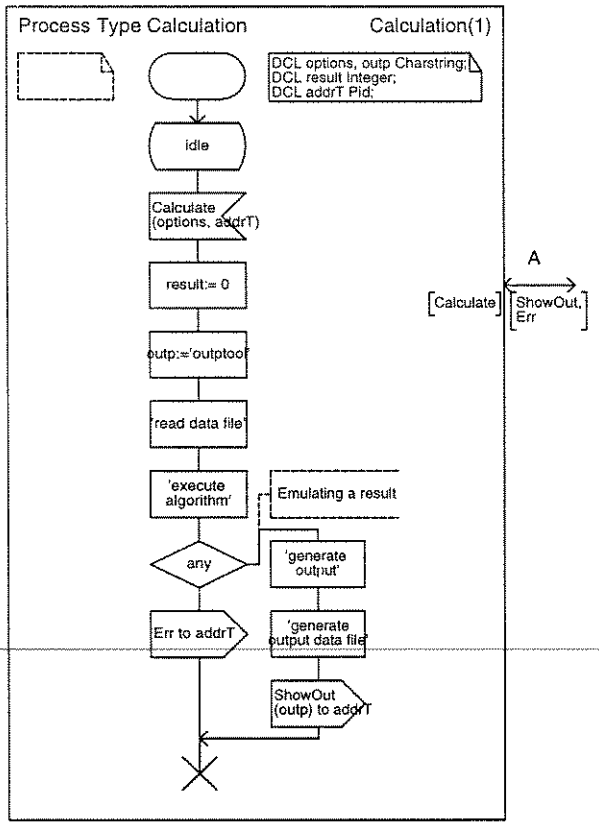


Figura 3.12- Process Type Calculation

Baseado nestas opções, ele lê o(s) arquivo(s) de dados e executa o cálculo do algoritmo correspondente. Caso ocorra algum erro durante este cálculo, retorna o sinal *Err* para o processo *Tool*, e caso contrário, gera o relatório de execução, e o arquivo de dados de saída, retornando o sinal *ShowOutp* para o processo *Tool*, cuja instância é identificada por *addrT*.

O bloco *FileConv* (Conversor de formatos - Figura 3.6) é composto pelos processos *FileConv* e *FCInterf* para conversão de arquivos externos.

- **Process type FileConv (Figura 3.13) / Bloco FileConv (Figura 3.6)**

O processo *FileConv* é responsável pela conversão de arquivos externos para o formato do AIDA, recebendo o sinal *ConvFile(file, addrAE),* cujos parâmetros indicam o

nome do arquivo a ser convertido e o identificador da instância *AE* do processo *AnalysisEnv* (Figura 3.7) que enviou o sinal, e cria então, a instância *FCI* do processo *FCInterf*, que corresponde à sua interface e que exibirá o arquivo de dados convertido.

- **Process type FCInterf (Figura 3.14) / Bloco FileConv (Figura 3.6)**

O processo *FCInterf* recebe o sinal *ShowConvFile* , recebido do processo *FileConv*, e exibe para o usuário o arquivo de dados convertido (sinal *CvtFl*).

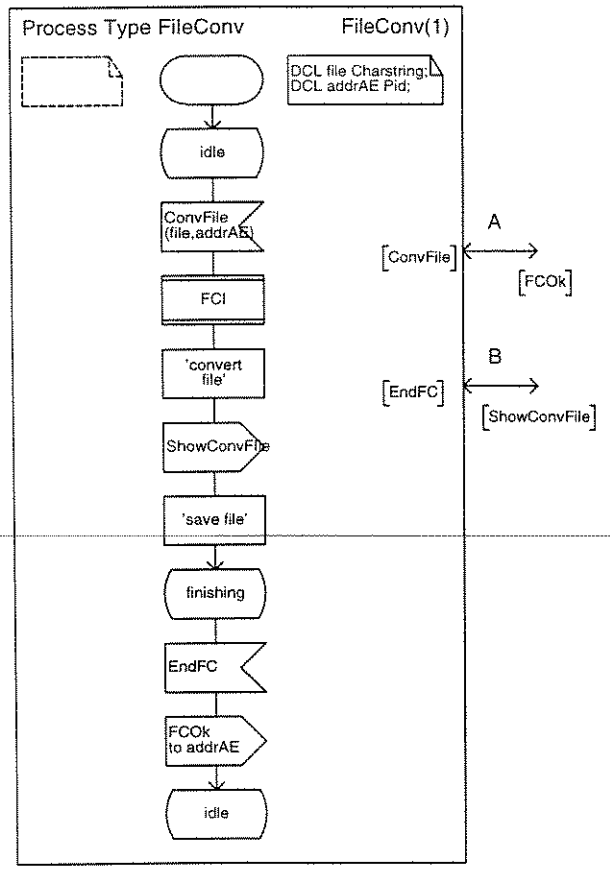


Figura 3.13- Process Type FileConv

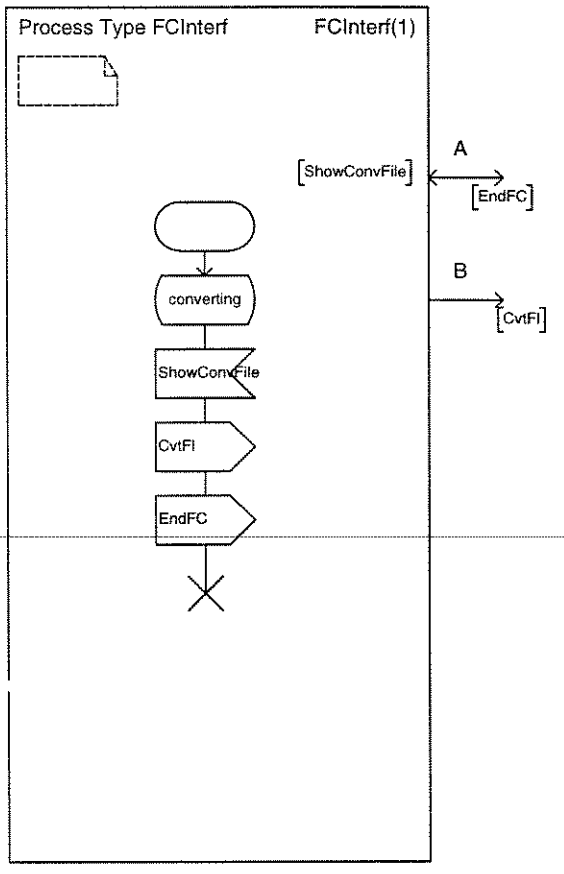


Figura 3.14- Process Type FCInterf

3.4. Modelo OMT do AIDA Concorrente

O modelo OMT da versão AIDA concorrente é mostrado na Figura 3.15 e consiste numa evolução do modelo centralizado (Figura 3.1) para um ambiente concorrente onde o processamento das ferramentas pode ser feito por servidores diferentes, como por exemplo em uma rede de PC´s ou de estações de trabalho, de maneira transparente ao usuário.

Esta versão tem como requisitos o controle do número de usuários que utilizam o AIDA, bem como das diversas requisições de execução de ferramenta de análise ou de

formatação de relatório que podem ser emitidas simultaneamente pelos diversos usuários do sistema. Além disso, também deve existir um controle sobre os servidores de processamento disponíveis.

Para atender a estes requisitos foram incluídas as classes *Controller* e *Server*, e as classes *Initial*, *AnalysisEnv* e *EInterf* do modelo centralizado foram evoluídas, passando a apresentar novas funcionalidades.

A classe *Initial* foi evoluída para permitir a execução do sistema por mais de um usuário, realizando o controle sobre o número máximo de usuários num determinado momento e a classe *EInterf* foi evoluída para prover suporte de interface gráfica às novas operações da classe *AnalysisEnv*.

A classe *AnalysisEnv*, além das funcionalidades que apresentava, passa a controlar as diversas requisições de execução que cada usuário pode emitir simultaneamente. Quando um usuário emite uma requisição, ela é repassada à classe *Controller* (Controlador de servidores), definindo um tempo máximo de espera pelo atendimento desta requisição. Caso este tempo seja atingido e nenhuma resposta tenha chegado, a classe *AnalysisEnv* solicita ao usuário manter ou cancelar a requisição. Na evolução do sistema centralizado para o sistema concorrente, esta classe passa a se relacionar com a classe *Controller*, em lugar da classe *Tool*.

A nova classe *Controller* (Controlador de servidores) é responsável por controlar as requisições dos usuários para a execução de ferramentas de análise ou formatação de relatórios. Ao receber uma requisição, esta classe verifica se existe alguma instância de servidor de processamento (*Server*) disponível, e havendo disponibilidade, envia a requisição para o servidor correspondente. Caso contrário, as requisições não atendidas são armazenadas numa fila de requisições até que algum dos servidores seja liberado.

A classe *Server* (Servidor de processamento) representa o servidor que recebe as requisições do controlador e as envias à classe *Tool*.

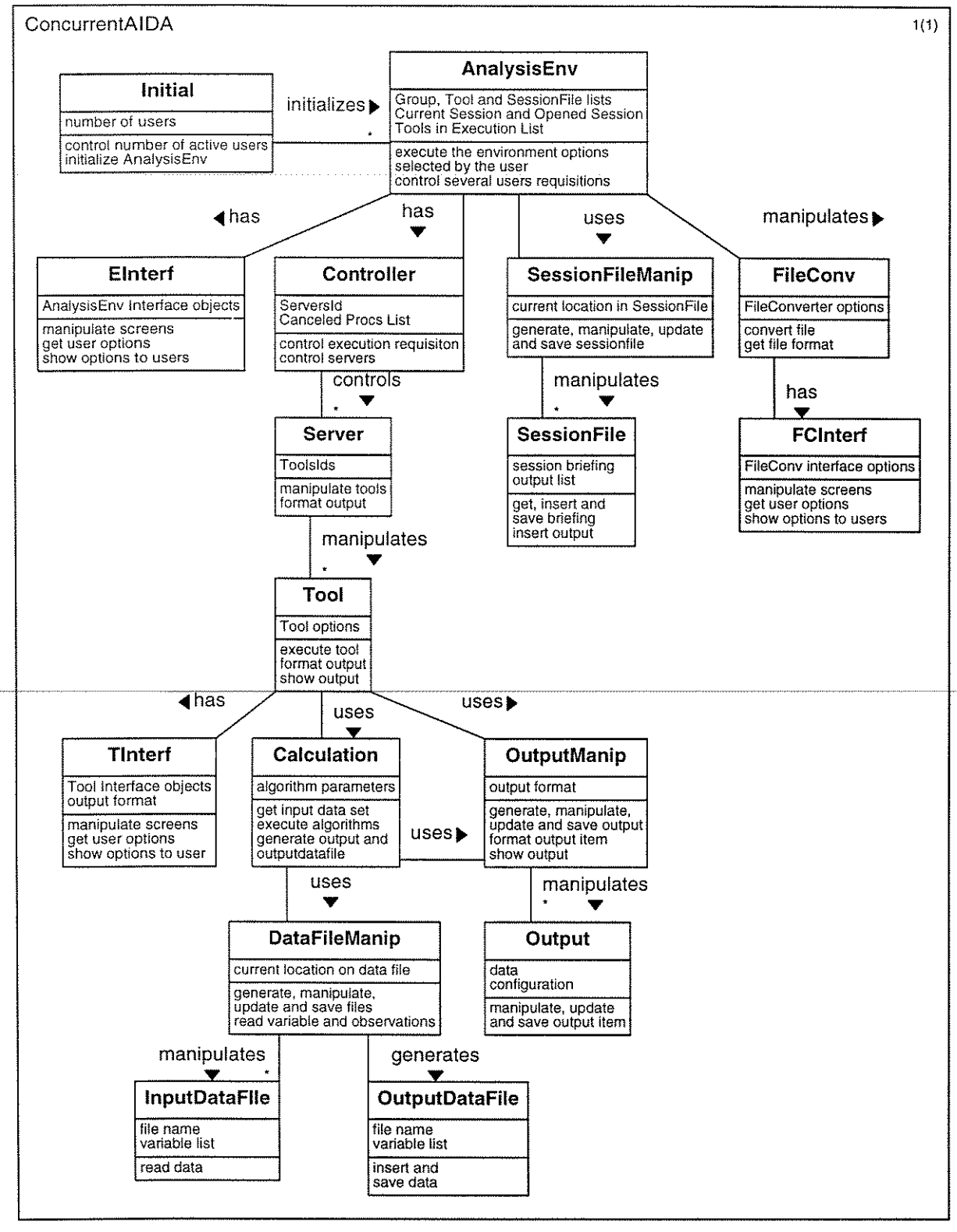


Figura 3.15- Modelo OMT do ConcurrentAIDA

3.5. Especificação SDL do AIDA Concorrente

Na versão centralizada (modelo OMT - Figura 3.1 e especificação SDL - Figura 3.3), existe apenas um usuário que pode executar localmente uma única ferramenta, seja para cálculo ou para formatar um relatório. Já na sua evolução para um ambiente concorrente, vários usuários podem executar o sistema de maneira concorrente, submetendo requisições de execução de ferramentas ou de formatação de relatórios, via *Controller*, às diversas instâncias do processo *Server*. Cada *Server*, por sua vez, pode executar, ao mesmo tempo, um número determinado de ferramentas, utilizando uma política de processamento do tipo "time-sharing".

Como a versão concorrente é uma especialização da versão centralizada, ela reusa as suas definições com a inclusão de novas funcionalidades e redefinição de algumas existentes. Assim, a elaboração de seu modelo SDL deve reutilizar todas as definições do sistema centralizado, redefinindo algumas de suas funcionalidades ou incluindo outras. O mecanismo de `PACKAGE` permite o reuso de definições de um sistema, tornando-as disponíveis para o sistema sendo especificado; entretanto, isto não é suficiente. Para que seja possível redefinir as propriedades do sistema reusado, deve ser utilizada a construção `INHERITS`, indicando a herança de tipos, em conjunto com a construção `REDEFINED`, usada para a redefinição de um block type ou de um process type da especificação herdada. Quando um sistema herda outro, devem ser especificadas apenas as diferenças existentes entre eles, com o reuso das instâncias definidas anteriormente ou com a definição de novas. Assim, a versão concorrente herda todas as definições do sistema centralizado (Figura 3.3) definindo apenas as diferenças entre eles.

A Tabela 3.2 apresenta o mapeamento de OMT-SDL somente das diferenças entre os sistemas. Assim, com relação ao sistema centralizado, as classes *Initial*, *AnalysisEnv*, *EInterf* e *Tool* foram redefinidas para incorporar novas funcionalidades do sistema concorrente, sendo mapeadas em SDL como *redefined process types*, e as novas classes *Controller* e *Server* do módulo *Tool*, foram representadas como *process types* no block type *Tool*. Como as demais classes do modelo OMT da versão centralizada apresentam as mesmas funcionalidades na versão concorrente (Figura 3.17), elas serão herdadas, não devendo ser especificadas novamente.

Analogamente ao que foi feito na versão centralizada, constrói-se um package contendo as definições do sistema concorrente, possibilitando o seu reuso em outras especificações. Na definição do package *Concurrent*, o package *Centralized* foi usado através da construção *USE* (Figura 3.16). Já na definição do system type *ConcurrentAIDA*

(Figura 3.17), o sistema *CentralizedAIDA* é herdado, através da construção *INHERITS* , e apenas as diferenças entre eles serão especificadas. Por exemplo, novos sinais que não existiam na especificação herdada devem ser definidos.

Classes OMT	Representação em SDL
AIDA	System Type ConcurrentAIDA que herda as definições do system type CentralizedAIDA, e é composto pelos redefined block types AnalysisEnv e Tool.
Módulo AnalysisEnv	Redefined block type AnalysisEnv, composto pelos redefined process types Initial, AnalysisEnv e EInterf.
Módulo Tool	Redefined block type Tool, composto pelo redefined process type Tool e pelos process types Controller e Server.

Tabela 3.2 - Mapeamento de OMT para SDL do sistema AIDA Concorrente

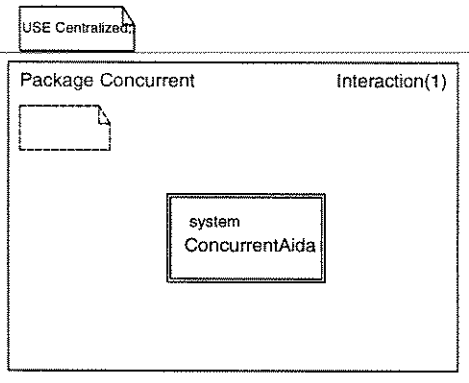


Figura 3.16- Package Concurrent

Como na versão concorrente são vários usuários executando mais de uma ferramenta ao mesmo tempo, tornou-se necessária a definição de sinais indicando que uma ferramenta de análise, ou o formatador de relatórios, teve seu processamento iniciado ou foi cancelada, identificando quem requisitou esta execução. Na linguagem SDL cada instância de um processo possui uma identificação única, do tipo Pid. Assim, para que o controle de ferramentas e usuários seja correto, os novos sinais possuem parâmetros deste tipo, carregando as identificações correspondentes. Na Figura 3.17 tem-se o sinal *InitExec(Pid)*, que indica que a ferramenta identificada por *Pid* teve seu processamento iniciado, e o sinal

CancelExec(Pid, Integer), que indica o cancelamento da execução da ferramenta identificada por *Pid*. O parâmetro *Integer* indica o seu índice na tabela que o processo *AnalysisEnv* mantém sobre as execuções de ferramentas que ele solicitou.

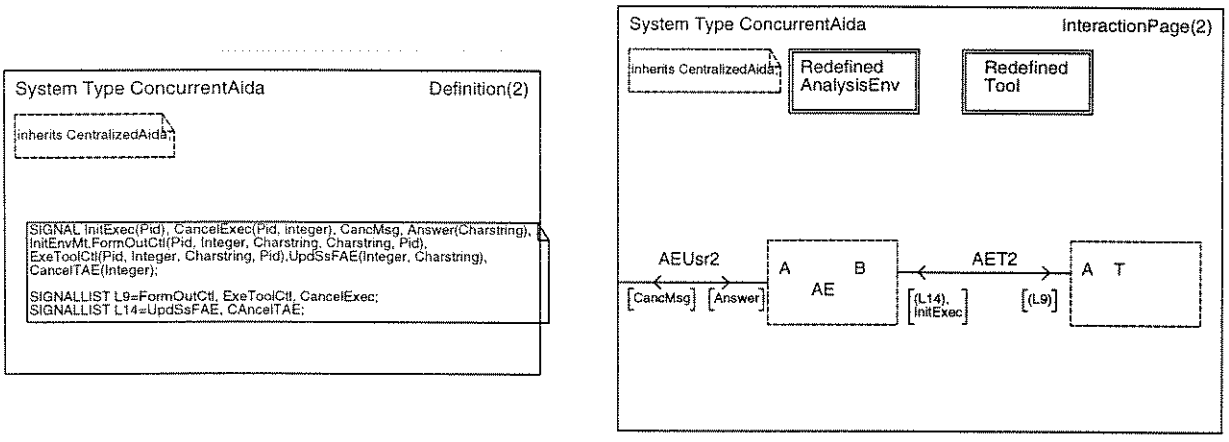


Figura 3.17- System Type ConcurrentAIDA

Os blocos *AnalysisEnv* e *Tool* foram redefinidos para a inclusão de novos sinais, e os elementos tracejados indicam que as instâncias *AE* e *T* definidas na versão herdada (Figura 3.3) estão sendo reusadas, mantendo as suas características originais, ou seja, são mantidos os seus processos, suas rotas de sinais e seus sinais. Neste reuso novos processos ou sinais podem ser adicionados às instâncias, devendo, entretanto, ser especificados novos canais para o transporte dos novos sinais, pois canais não podem ser redefinidos. Por esta razão, na Figura 3.17 os canais *AEUsr2* e *AET2* foram definidos para transportar os novos sinais do sistema. O bloco *FileConv* não foi alterado e como apenas as diferenças entre os sistemas devem ser especificadas, nenhuma referência a ele é feita.

Na redefinição do block type *AnalysisEnv* (Figura 3.18), os processos *AnalysisEnv* e *EInterf* passaram a apresentar as novas instâncias *AE2* e *EI2* que receberão os novos sinais, além dos originais. A princípio poderiam ter sido reusadas as instâncias *AE* e *EI* da especificação herdada, mas era necessário mudar o limite máximo de usuários do sistema, que na versão centralizada é igual a um. Assim, as novas instâncias *AE2(0,)* e *EI2(0,)* apresentam um limite indeterminado de usuários, já que o número máximo de instâncias que podem ser criadas não é fixado, ficando o processo *Initial* responsável por controlar este número.

Como *AE2* e *EI2* não são instâncias reusadas da versão centralizada, é necessária a declaração de todos os sinais que são recebidos ou enviados por elas, e não apenas os novos. O processo *Initial* foi redefinido para permitir a execução do sistema por mais de

uma pessoa. A sua instância *Init* foi reusada, já que os números inicial e máximo de instâncias deste processo para uma execução do sistema não foi alterado.

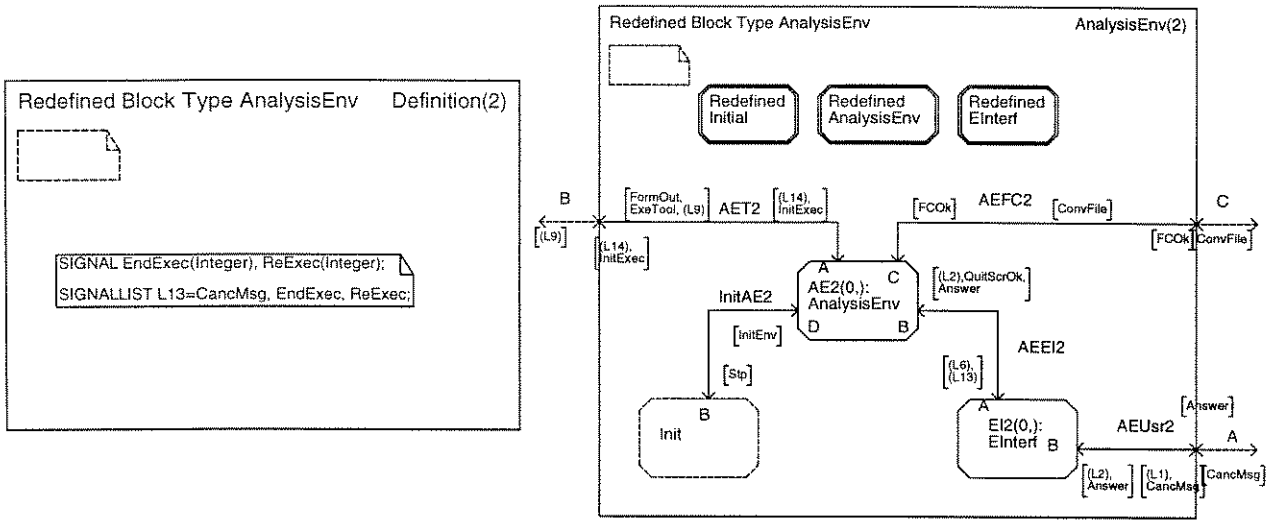


Figura 3.18- Redefined Block Type AnalysisEnv / ConcurrentAIDA

O block type *Tool* (Figura 3.19) foi redefinido para incluir os novos process types *Controller* e *Server*. O processo *Controller* é responsável por gerenciar as requisições de execução de ferramentas de análise ou de formatação de relatórios feitas pelos usuários, repassando-as ao servidor de processamento. Pela especificação de sua instância *Ctl (1,1)*, no início da execução do sistema existe apenas um controlador, não podendo ser criado nenhum outro. Já o processo *Server* se encarrega de enviar estas requisições para o processo *Tool*, que efetivamente executa a ferramenta ou formata o relatório. A especificação de sua instância *Sv (0,2)* indica que no início da execução do sistema não existe nenhuma instância deste processo, e que só poderão existir duas delas simultaneamente. Na especificação deste bloco no Cap. 4 o limite se servidores passa a ser ilimitado. Além da inclusão dos novos processos, foram definidas novas instâncias *T2*, *TI2* e *Calc2* do processos *Tool*, *TInterf* e *Calculation* para definir novos números inicial e máximo de instâncias criadas durante uma execução. Assim, diferentemente da especificação do bloco *Tool* na versão centralizada (Figura 3.5), no início do sistema não existe nenhuma instância destes processos, e também deixa de haver limites com relação ao número de ferramentas em execução ao mesmo tempo. O processo *Tool* foi redefinido para criar as novas instâncias definidas.

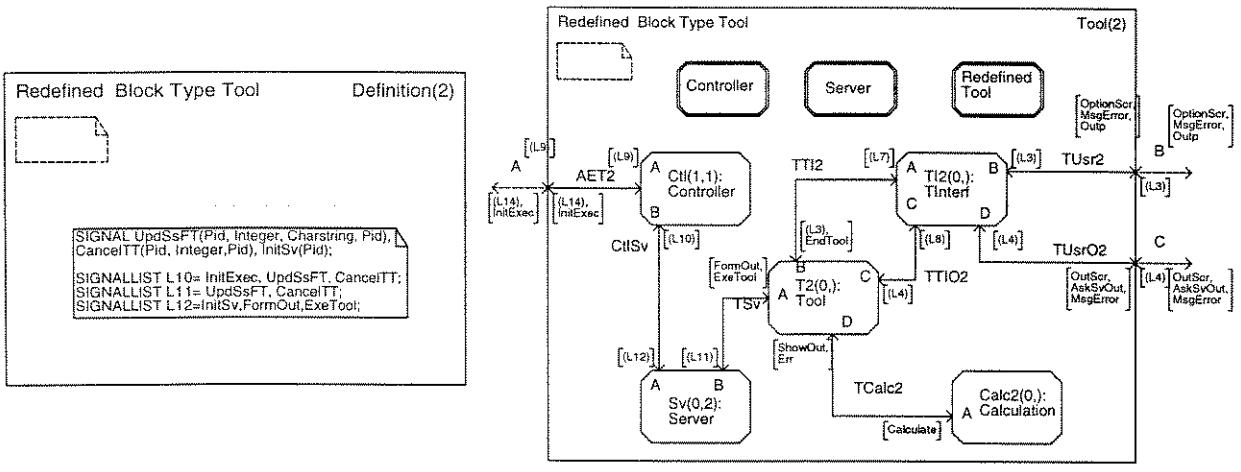


Figura 3.19- Redefined Block Type Tool / ConcurrentAIDA

Uma transição em um processo SDL corresponde a um estado, um sinal de entrada, que faz o processo sair daquele estado, e um conjunto de tarefas seguidas de um outro estado, onde cada sinal de entrada em um estado indica uma transição diferente. Para a redefinição de transições é necessário que o sinal de entrada da transição a ser redefinida, seja definido como *virtual* no processo original. A sua redefinição se dá no processo através da construção *redefined* onde o estado inicial desta transição, em conjunto com o seu sinal de entrada, agora *redefined,* são definidos. As tarefas que o seguem podem ou não levar ao estado final da transição especificada no processo original. Na execução do sistema, são inicialmente executados os símbolos do processo original, e encontrando-se uma transição definida como *virtual*, a execução é desviada para o processo *redefined,* que contém redefinição desta transição.

- **Redefined process type *AnalysisEnv* (Figura 3.20) / Bloco AnalysisEnv (Figura 3.18)**

A principal diferença do processo *AnalysisEnv* redefinido com relação à sua versão centralizada (Figura 3.7), é que no sistema concorrente ele passa a controlar as diversas requisições de execução das ferramentas de análise e de formatação de relatórios emitidas por um usuário, informando-o sobre os seus atendimentos. Na versão centralizada este processo enviava um sinal para execução da ferramenta, ou formatação de relatórios, e nada era feito até que chegasse algum resultado desta execução. Na versão concorrente, passa a existir um temporizador *T*, que é ativado através da construção *set(now+100,T)*, toda vez que o usuário fizer uma requisição de execução. Esta construção ativa o temporizador *T*, atribuindo a ele um tempo absoluto (*now*) mais um tempo relativo (*100*).

Quando o tempo resultante (*now +100*) é atingido, o temporizador envia um sinal *T* para o mesmo processo que o definiu, no caso *AnalysisEnv*. Assim, se o tempo máximo de espera por uma resposta é atingido, este processo envia o sinal *CancelMsg* para o processo de interface (*EInterf*), solicitando ao usuário manter ou excluir a sua requisição da fila de processos a serem executados. Caso o usuário decida por manter a requisição, o temporizador *T* é novamente ativado e o sinal *ReExec(1)* é enviado para o processo de interface, indicando que a execução foi mantida. Caso contrário, ele é desativado, através da construção *reset(T)*, o sinal *CancelExec* é enviado para o processo *Controller* e o sinal *ReExec(0)* para o processo *EInterf*, ambos indicando que a requisição foi cancelada.

Na recepção do sinal *initEnv*, é criada a instância *EI2* do processo *EInterf* e não mais *EI* como era na versão centralizada.

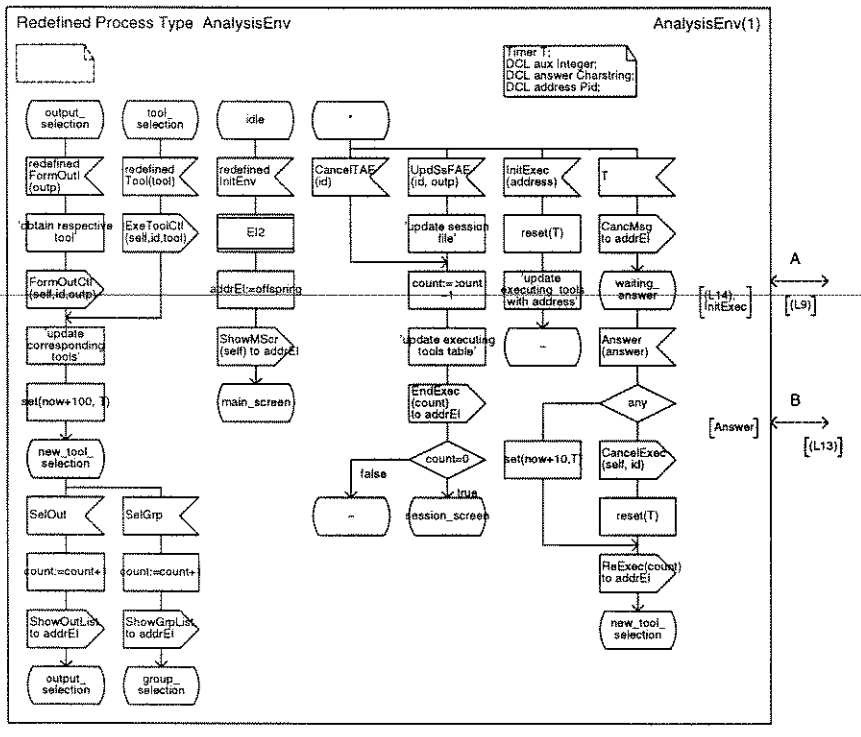


Figura 3.20 - Redefined Process AnalysisEnv

- **Redefined process type EInterf (Figura 3.21) / Bloco AnalysisEnv (Figura 3.18)**

O processo *EInterf* foi redefinido para prover o suporte de interface às novas funcionalidades do processo *AnalysisEnv*. Por exemplo, foi incluída uma transição para o tratamento do sinal *CancMsg*, que indica que em qualquer estado que o processo se encontre, na recepção deste sinal deve ser emitida uma mensagem para o usuário

solicitando excluir ou manter um processo na fila de requisições do processo *Controller*. Além disso, foi estabelecido que usuário não pode finalizar o ambiente enquanto existir alguma ferramenta de análise ou formatador de relatório em execução, e por esta razão, quando o usuário emite uma requisição de execução, as únicas opções que ficam disponíveis para ele são a seleção de um novo relatório para formatação, ou de um novo grupo de ferramentas de análise (estado *waiting_execution* - sinais *SelOut* e *SelGrp*).

- **Redefined process type Initial (Figura 3.22) / Bloco AnalysisEnv (Figura 3.18)**

O processo *Initial* foi redefinido para permitir a execução do sistema por mais de um usuário simultaneamente. A cada usuário que inicia o sistema, é criada uma nova instância *AE2* do processo *AnalysisEnv* (Figura 3.20). Para que a validação do sistema (seção 3.6) não se tornasse muito extensa, foi definido o número máximo de dois usuários executando o AIDA simultaneamente, os quais são controlados no processo pela variável *count*.

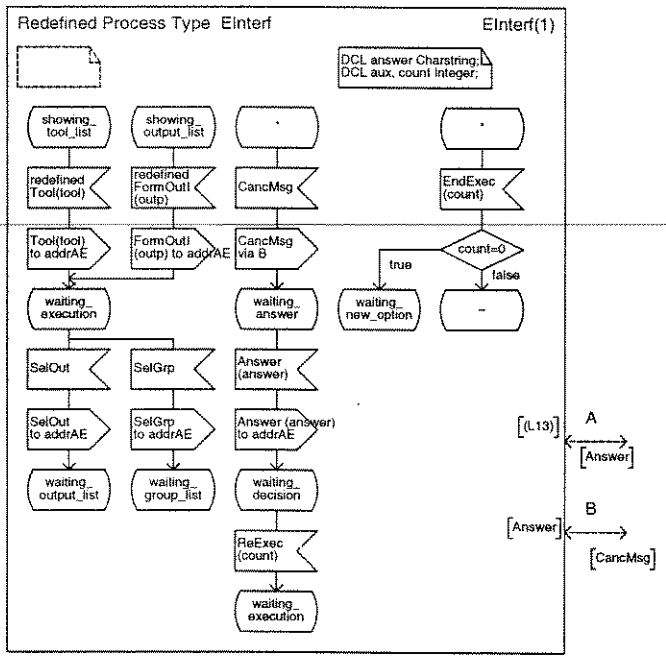


Figura 3.21- Redefined Process Type EInterf

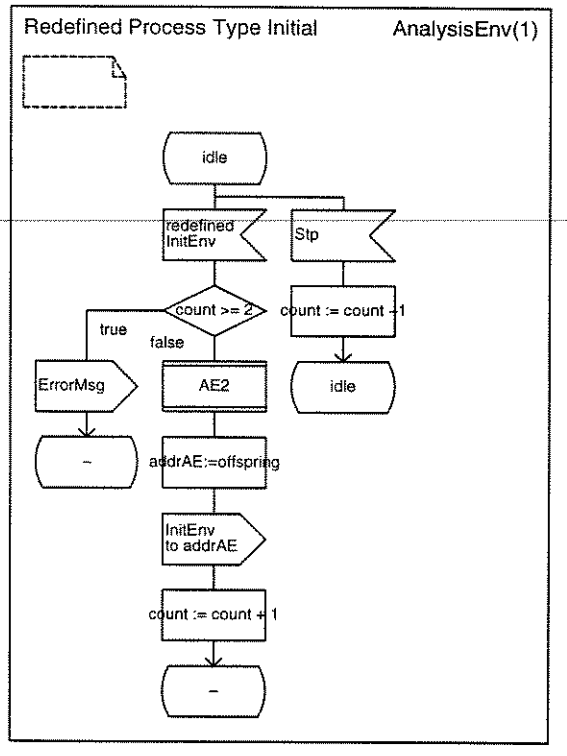


Figura 3.22- Redefined Process Type Initial

- **Process type Controller (Figura 3.23) / Bloco Tool (Figura 3.19)**

O novo processo *Controller*, a partir do recebimento de uma requisição de execução de ferramenta (sinal *ExeToolCtl*) ou de formatação de relatório (sinal *FormOutCtl*), verifica a disponibilidade de servidores para atendê-la.

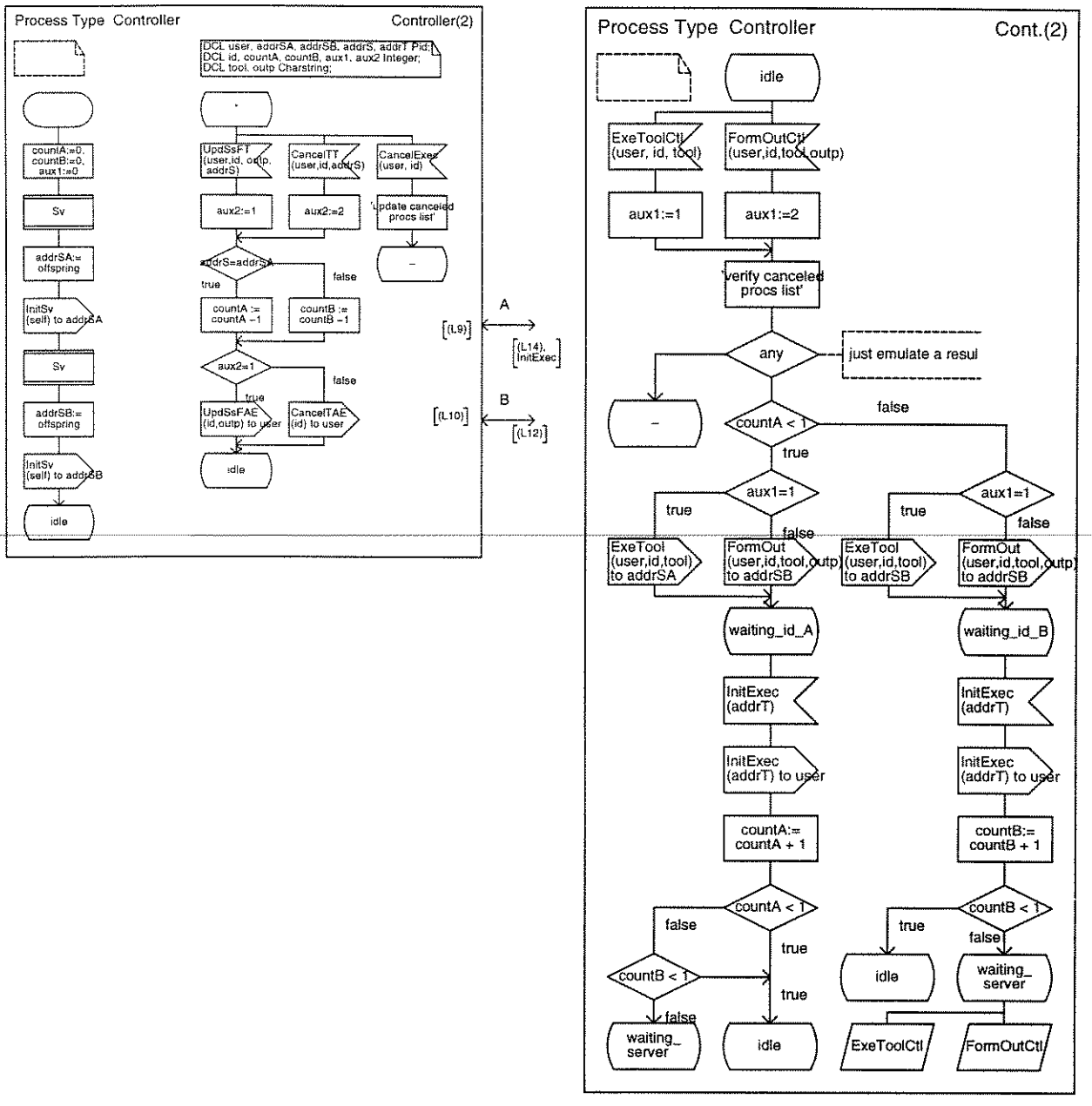


Figura 3.23- Process Type Controller

Havendo uma das duas (nesta versão concorrente) instâncias supostas de *Server* disponível, o processo envia a requisição de execução para a instância correspondente, identificada por *addrSA ou addrSB*, esperando o recebimento do sinal *InitExec*, que representa o início do processamento daquela ferramenta. Este sinal, com o identificador

(*addrT*) da ferramenta (*Tool*) em execução, é repassado ao processo *AnalysisEnv* (Figura 3.20) do usuário requisitante, indicando que o seu pedido foi atendido. As eventuais requisições que não podem ser atendidas são colocadas numa fila FIFO, utilizando para isso o símbolo *SAVE* (estado *waiting_server*, sinais *ExeToolCtl* e *FormOutCtl*), que salva, nesta fila, o sinal recebido sem consumi-lo. Assim que um servidor de processamento é liberado (sinais *UpdSsFT* ou *CancelTT* na outra página do processo *Controller*) e estiver pronto para executar uma nova requisição, o controlador obtém a primeira solicitação da fila e envia para este servidor.

- **Process type Server (Figura 3.24) / Bloco Tool (Figura 3.19)**

O novo processo *Server* ao receber uma requisição de execução (sinal *ExeTool* ou *FormOut*), cria uma instância *T2* do processo *Tool*, inicia a sua execução, retorna ao controlador o sinal *InitExec* com o Pid da ferramenta iniciada (*addrC*), e incrementa o contador de ferramentas (*count*) de 1.

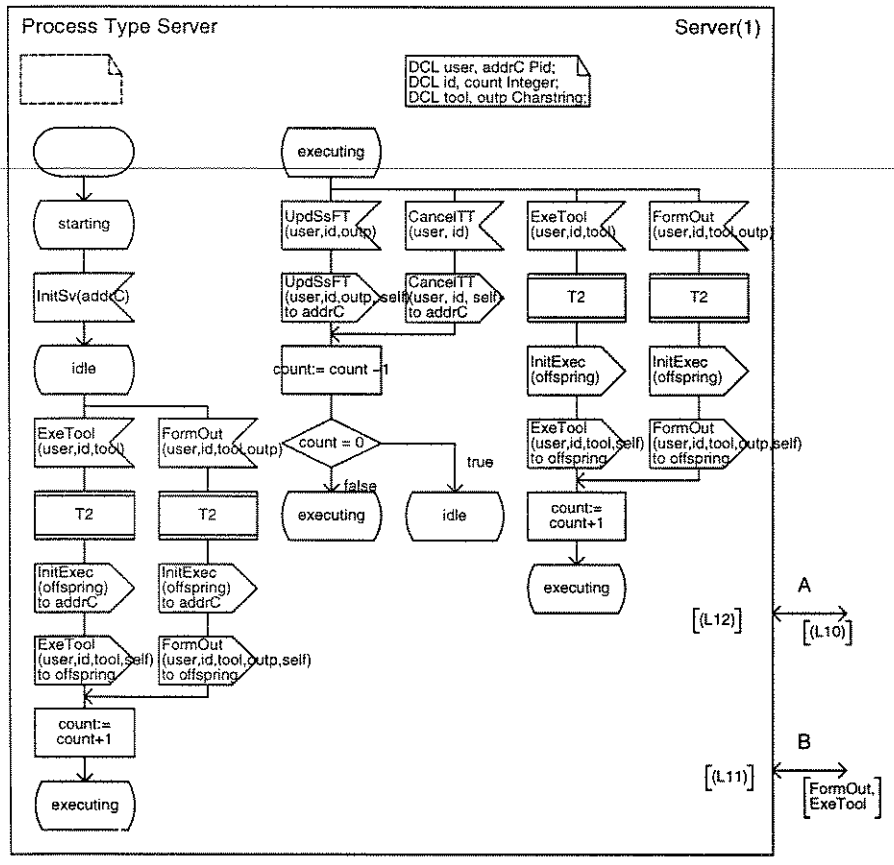


Figura 3.24- Process Type Server

Na recepção de um sinal que indica o fim de execução de uma ferramenta de análise ou da formatação de um relatório (sinais *UpdSsFIT* ou *CancelTT*), o servidor decrementa o

contador e repassa este sinal para o controlador, indicando que está liberado para atender mais uma requisição.

- **Redefined process type Tool (Figura 3.24) / Bloco Tool (Figura 3.19)**

O processo *Tool* foi redefinido para criar as novas instâncias *TI2* e *Calc2* dos process types *TInterf* e *Calculation* (*TI* e *Calc* na versão centralizada - Figura 3.5), e para enviar os sinais *CancelT* e *UpdSsFT* (estados *finishing_tool*, * e *waiting_answer*) para o processo *Server*, já que na versão centralizada estes sinais eram enviados para o processo *AnalysisEnv* (Figura 3.7).

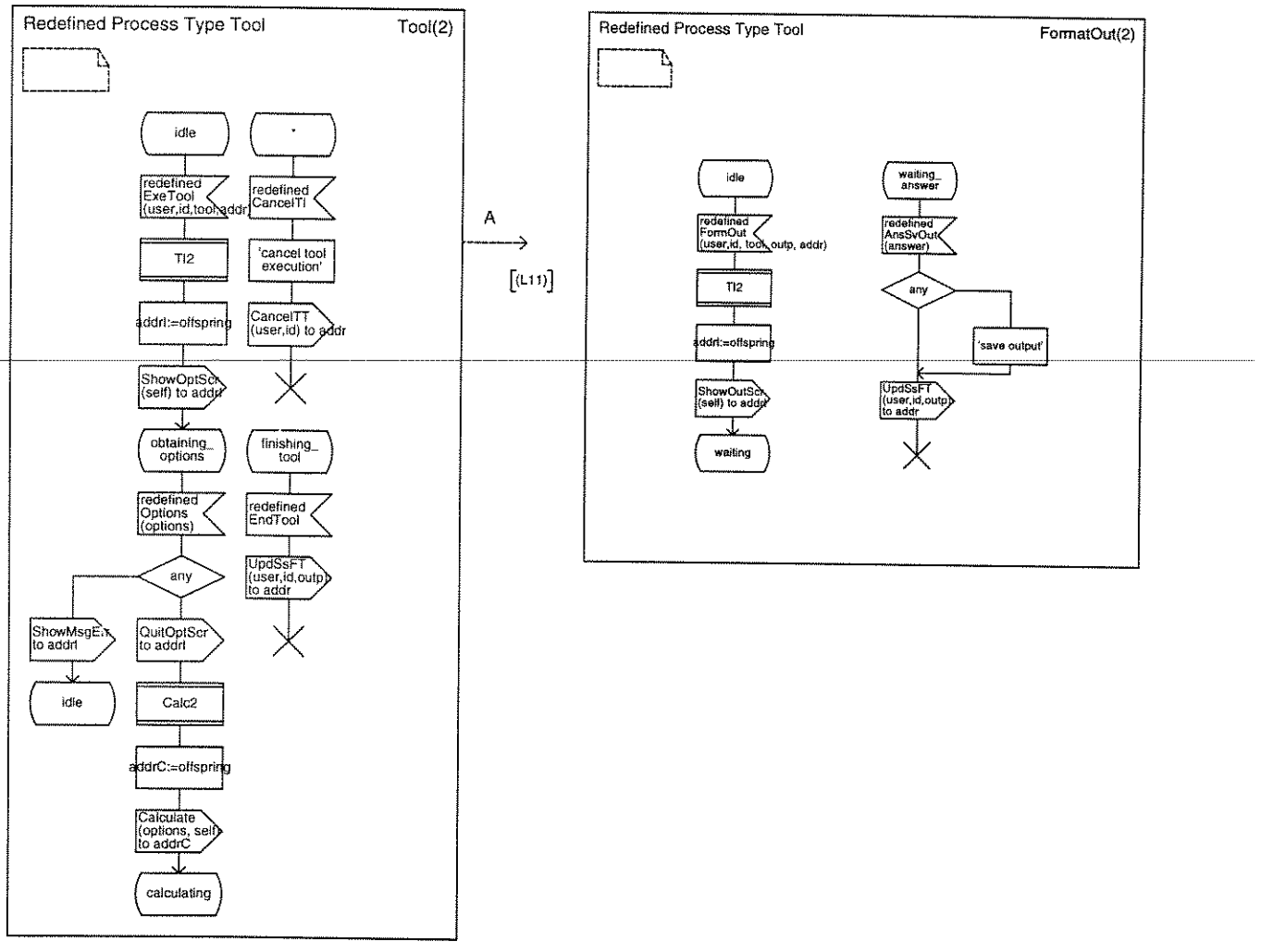


Figura 3.25- Redefined Process Type Tool

3.6. Validação do sistema AIDA

A validação do sistema AIDA foi feita através da ferramenta *Validator*, disponível no ambiente SDT [TELELOGIC AB, 1997]. A atividade de validação permitiu a identificação de falhas na especificação que provavelmente só se tornariam visíveis na fase de teste. Além disso, tornou a correção destas falhas uma atividade simples, já que o *Validator* [TELELOGIC AB, 1997] possibilita a identificação exata da situação em que elas ocorrem.

A ferramenta *Validator* do SDT é baseada na "state space exploration", uma técnica muito conhecida para a análise automática de sistemas distribuídos. Todas as ferramentas SDT baseadas nesta técnica (*Simulator* e *Validator*), consideram a geração automática de uma estrutura do tipo árvore que represente o sistema. Assim, durante o processo de validação, o sistema SDL é representado por uma árvore de comportamento, onde cada nó corresponde a um estado completo do sistema. Cada um destes estados é determinado, entre outras coisas, pelas instâncias ativas de processos que mudaram desde o estado anterior, pelo estado em que elas se encontram e pelas suas filas de sinais de entrada. Movendo-se pela árvore, o comportamento do sistema pode ser explorado, e os estados alcançados, examinados manual ou automaticamente. Para cada estado alcançado durante a exploração, um número de regras são testadas para detectar erros ou possíveis problemas no sistema. Se uma regra é violada, uma mensagem de erro é gerada para o usuário. Investigando-se esta mensagem e a árvore do sistema onde o erro é gerado, a sua causa pode ser identificada.

O *SDT Validator* apresenta 3 tipos de validação automática baseados na técnica "state space exploration":

- bit state exploration, que usa um algoritmo eficiente para testar sistemas de tamanho médio;
- random walk, que usa um algoritmo simples para testar sistema grandes, ou seja, sistemas que apresentam um grande número de estados gerados; e
- exhaustive exploration, que é usado para testar sistemas pequenos que devem apresentar um alto nível de corretitude.

Como o AIDA é um sistema onde um ou mais usuários podem executar um número indeterminado de ferramentas, diferentes instâncias de usuários e de ferramentas são criadas durante a validação, o que leva a um número grande de estados gerados. Assim, na sua validação é mais apropriado o uso do método *random walk*, que realiza uma exploração do tipo "depth-first" na árvore de comportamento, escolhendo repetidamente caminhos

aleatórios a serem percorridos na árvore para teste. Na execução deste teste devem ser informados a profundidade da busca, que determina o número máximo de transições a serem exploradas em cada um dos caminhos de teste escolhidos pela ferramenta, e o número de repetições, que indica quantas buscas aleatórias serão feitas. Quando a profundidade definida é alcançada, a pesquisa é reiniciada a partir do estado inicial da árvore e uma nova busca randômica é feita.

Como a escolha dos caminhos a serem percorridos na árvore é aleatória, nada garante que todos os símbolos do sistema (elementos utilizados na descrição dos processos) sejam testados. Assim, é importante a utilização de outra ferramenta disponível no SDT: o *Coverage Viewer*, que mostra o resultado da validação segundo a cobertura dos símbolos do sistema através de duas janelas, a principal e a detalhada (*main* e *detailed*), que devem ser analisadas em conjunto.

A janela principal do *Coverage Viewer* apresenta a cobertura dos símbolos de um sistema numa árvore que representa a sua estrutura. Todos os nós desta árvore correspondem a elementos dos sistema SDL (packages, sistemas, blocos e processos), sendo que os nós de nível mais baixo correspondem aos símbolos SDL usados na especificação dos processos. O nome de cada elemento aparece abaixo do seu desenho, e nos níveis mais altos, é informado o número total de vezes que os símbolos dos processos associados a este elemento foram executados, e também um intervalo que representa quantas vezes o menos e o mais executado destes símbolos foram testados. O preenchimento em cinza de cada elemento da árvore é proporcional à quantidade de símbolos associados a ele que foram cobertos.

Já a janela detalhada do *Coverage Viewer* mostra uma cobertura mais detalhada para um nó (elemento) especifico da árvore da janela principal, apresentando um gráfico de cobertura dos símbolos dos processos associados a este elemento. Acima deste gráfico aparecem o tipo e nome do nó selecionado na janela principal, o número total de vezes que os símbolos associados a este nó foram executados, o número total de símbolos cobertos e o número total de símbolos que não foram cobertos. O diagrama de cobertura mostra quantos símbolos do nó foram executados um determinado número de vezes. O eixo horizontal define o número de vezes que cada símbolo foi executado, e o intervalo deste eixo é o mesmo apresentado pelo nó selecionado na janela principal do *Coverage Viewer*. O eixo vertical corresponde ao número de símbolos que foram executados um determinado número de vezes. Para cada valor no eixo horizontal, uma barra vertical mostra quantos símbolos foram testados este número de vezes.

3.6.1. Resultado de validação do AIDA Centralizado

Os resultados da validação do sistema AIDA centralizado (Figura 3.3) são mostrados na Figura 3.26.

```
Random-walk

** Starting random walk **
Depth      : 100
Repetitions : 100

** Random walk statistics **
No of reports: 0
Gen states   : 14300
Max depth    : 100
Min depth    : 100
```

Figura 3.26 - Resultados da Validação do AIDA Centralizado

Na validação representada pela Figura 3.26, a árvore de comportamento criada possui uma profundidade máxima (*depth*) de 100, e foram realizadas 100 escolhas aleatórias (*repetitions*) de caminhos para teste. Segundo os resultados apresentados, não foi encontrada nenhuma situação de erro (*No. of reports*) na validação dos 14.300 estados gerados (*gen. states*).

Para verificar a cobertura dos símbolos usados na definição dos processos do sistema, isto é, verificar se todos eles foram executados, deve ser utilizada a ferramenta *Coverage Viewer*. A Figura 3.27 apresenta a árvore de cobertura de símbolos gerada pelo *Coverage Viewer* para o sistema *CentralizedAIDA*, onde estão representados os elementos mais executados do sistema (package, system type, block types e process types). Segundo esta árvore os símbolos utilizados na especificação dos processos do sistema foram executados 22.220 vezes. O menos executado deles foi executado 5 vezes e o mais executado foi executado 2.001 vezes. Alguns dos símbolos utilizados no processo *Initial* foram executados 2.001 vezes, tendo sido os mais executados do sistema. Os elementos dessa árvore tiveram todos os seus símbolos cobertos durante a validação do sistema, e portanto aparecem totalmente preenchidos na cor cinza. Isto também pode ser comprovado através da janela detalhada do *Coverage Viewer* apresentada na Figura 3.28, que dentre outras informações, indica que 280 dos 280 símbolos utilizados (100%) na especificação dos seus processos foram cobertos, ou seja, nenhum deles deixou de ser testado.

outras informações, indica que 280 dos 280 símbolos utilizados (100%) na especificação dos seus processos foram cobertos, ou seja, nenhum deles deixou de ser testado.

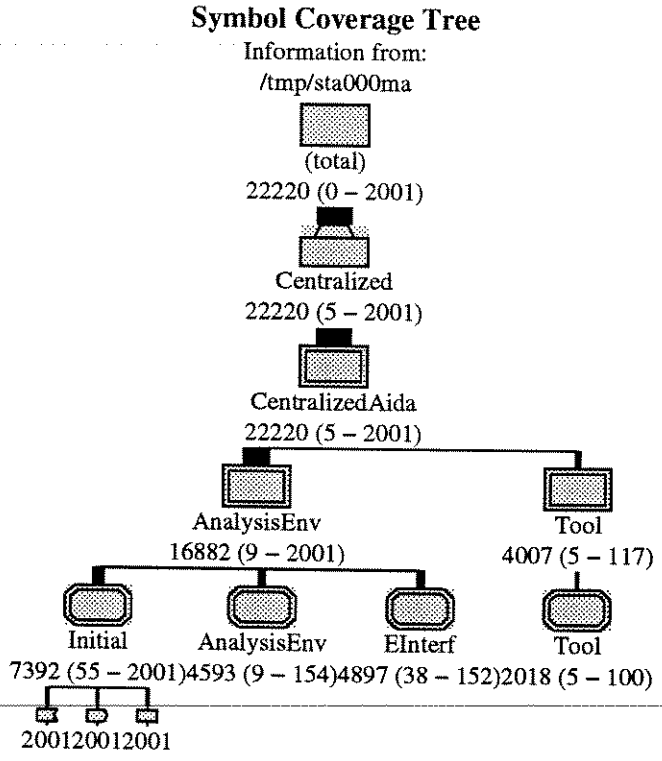


Figura 3.27 - Coverage Viewer - AIDA Centralizado - elementos mais executados

O gráfico de cobertura (Figura 3.28) apresenta a quantidade de símbolos pelo número de vezes em que foram executados. A maioria dos 280 símbolos foi executada algo em torno de 100 a 150 vezes. Apenas os símbolos do processo *Initial* (Figura 3.27) foram executados 2.001 vezes. Isto ocorre porque durante a validação foram feitas várias tentativas de acesso ao sistema por mais de um usuário ao mesmo tempo e como na versão centralizada apenas um usuário por vez executa o sistema, estas tentativas foram recusadas, não permitindo a execução dos demais símbolos utilizados na especificação do AIDA.

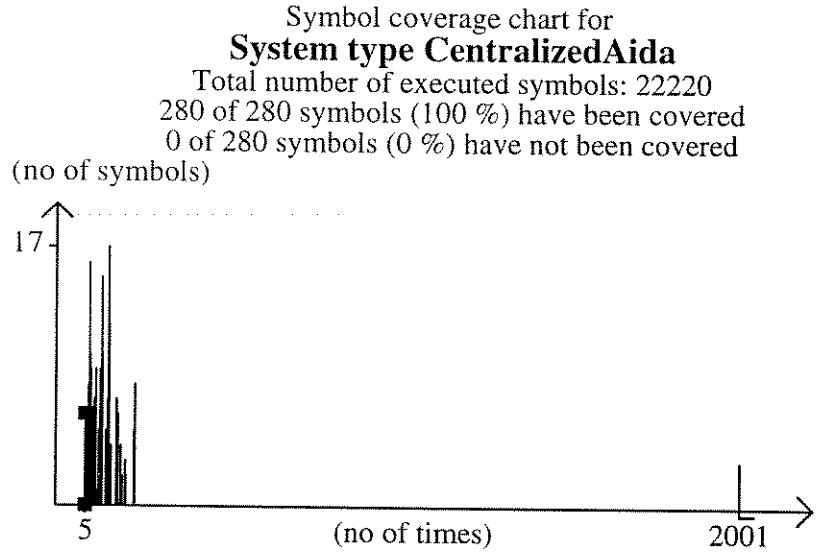


Figura 3.28 - Coverage Viewer Detalhado- AIDA Centralizado

3.6.2. Resultado de validação do AIDA Concorrente

Os resultados da validação do sistema AIDA Concorrente são mostrados na Figura 3.29.

```
Random-walk

** Starting random walk **
Depth      : 300
Repetitions : 100

** Random walk statistics **
No of reports: 0
Gen states   : 48683
Max depth    : 300
Min depth    : 300
```

Figura 3.29 - Resultados da Validação do AIDA Concorrente

Na validação representada pela Figura 3.29, a árvore de comportamento criada possui uma profundidade máxima (*depth*) de 300, e foram realizadas 100 escolhas (*repetitions*) de caminhos para teste. Segundo os resultados apresentados, não foi encontrada nenhuma situação de erro (*No. of reports*) na validação dos 48.683 estados

gerados (*gen. states*) para o sistema *ConcurrentAIDA*. A profundidade da árvore de comportamento (*depth*) foi aumentada para 300 para possibilitar a cobertura de todos os símbolos do sistema, o que não foi possível com a profundidade 100 ou 200. A Figura 3.30 e a Figura 3.31 apresentam o resultado da validação e o gráfico de cobertura detalhado para estes dois casos, mostrando que em ambos os casos, alguns dos 194 símbolos definidos na especificação do sistema *ConcurrentAIDA* não foram cobertos.

```
Random-walk

** Starting random walk **
Depth      : 100
Repetitions : 100

** Random walk statistics **
No of reports: 0
Gen states   : 16443
Max depth    : 100
Min depth    : 100
```

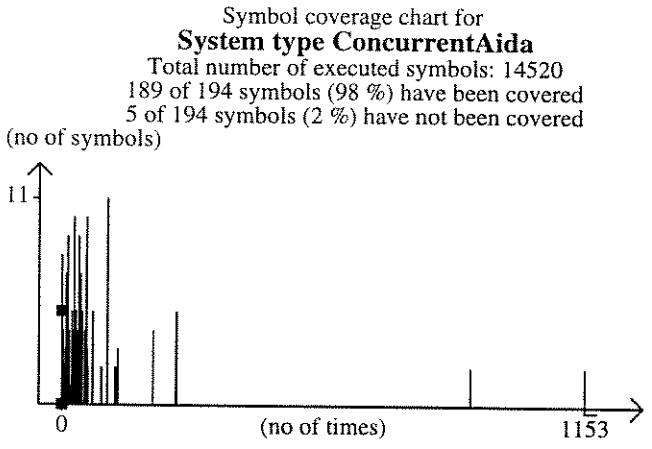


Figura 3.30 - Resultados da Validação do AIDA Concorrente com depth 100

```
Random-walk

** Starting random walk **
Depth      : 200
Repetitions : 100

** Random walk statistics **
No of reports: 0
Gen states   : 32477
Max depth    : 200
Min depth    : 200
```

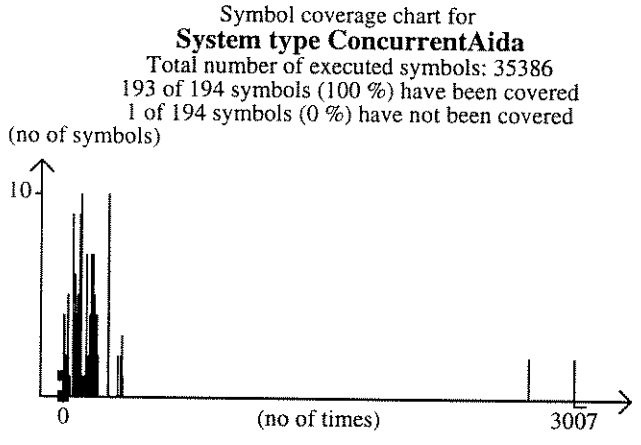


Figura 3.31 - Resultados da Validação do AIDA Concorrente com depth 200

A Figura 3.32 mostra a janela principal do *Coverage Viewer* para a validação final do *ConcurrentAIDA* onde são apresentados os nós mais executados do sistema. A especificação da versão concorrente foi elaborada através da herança e do reuso da versão centralizada, e por esta razão a árvore de comportamento criada na validação apresenta os dois sistemas (centralizado e concorrente), onde o elemento *total* desta árvore apresenta informações do sistema como um todo, considerando ambas especificações. Este elemento

não foi totalmente preenchido em cinza, indicando que parte de seus símbolos não foram cobertos. Ao obtermos informações detalhadas sobre ele (Figura 3.33), verifica-se que 74, dos 474 símbolos de processos associados a ele deixaram de ser cobertos. Isto se deve ao fato de que na herança do sistema centralizado pelo sistema concorrente parte dos processos do sistema centralizado tiveram algumas transições redefinidas, e assim, os símbolos que fazem parte das transições originais (definidas como virtual) deixam de ser executados e são substituídos por aqueles definidos nas suas redefinições (sistema concorrente). Ou seja, os 74 símbolos que deixaram de ser cobertos fazem parte do sistema centralizado, o que pode ser verificado na janela detalhada correspondente (Figura 3.34).

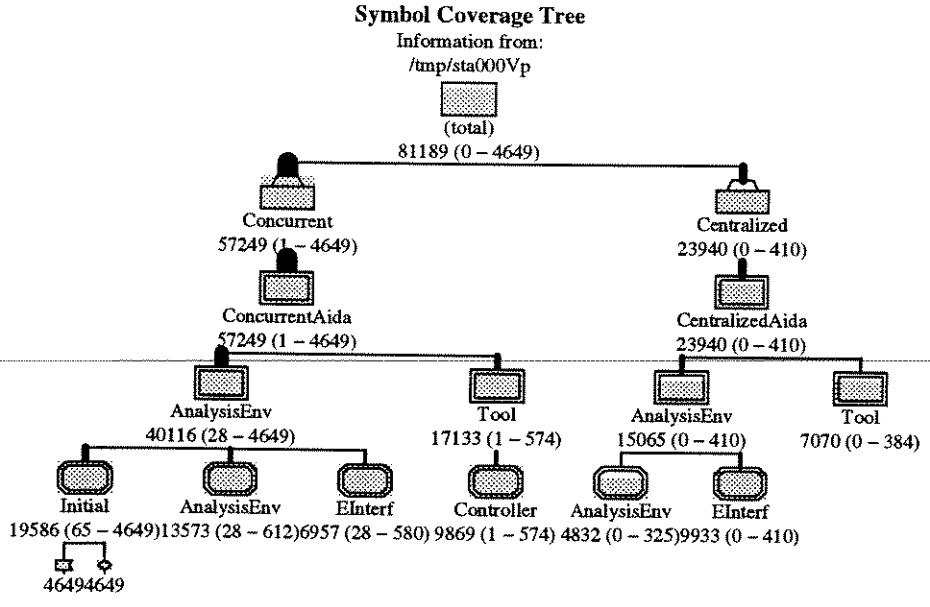


Figura 3.32 - Coverage Viewer - AIDA Concorrente

Por exemplo, pela janela detalhada do processo *AnalysisEnv* da versão centralizada (Figura 3.35), verificamos que 23 símbolos associados a este processo deixaram de ser executados, o que pode ser comprovado na sua especificação (Figura 3.7): 5 símbolos na transição definida pelo sinal *virtual InitEnv*, 7 na transição do sinal *virtual FormOutI* e 11 na transição do sinal *virtual Tool.*

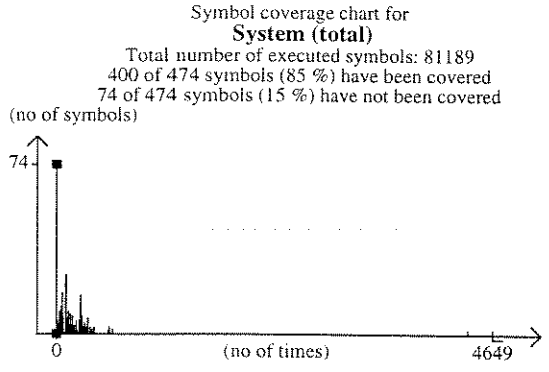


Figura 3.33 - Coverage Viewer Detalhado -Total

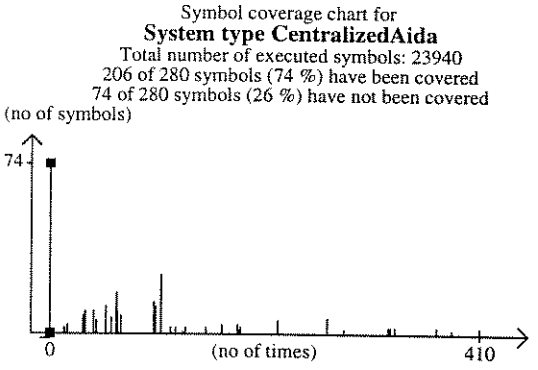


Figura 3.34 - Coverage Viewer Detalhado - System Type CentralizedAIDA

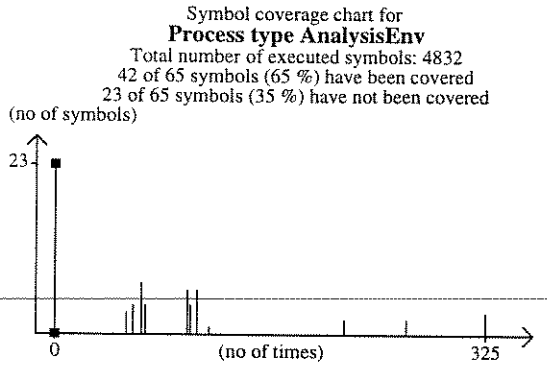


Figura 3.35 - Coverage Viewer Detalhado - Process Type AnalysisEnv/AIDA Centralizado

Segundo a Figura 3.32, os símbolos associados ao sistema *ConcurrentAIDA* foram executados 57.249 vezes, e o menos executado deles foi executado 1 vez e o mais executado foi executado 4.649 vezes. Da mesma forma que no *CentralizedAIDA*, os símbolos associados ao process type *Initial* foram os mais executados do sistema, e aqui foram executados 4.649 vezes.

A janela de cobertura detalhada da validação final do *ConcurrentAIDA* (com profundidade 300) é apresentada na Figura 3.36, indicando que os símbolos associados a este nó foram executados 57.249 vezes, e que, nenhum dos 194 símbolos definidos no sistema *ConcurrentAIDA* deixou de ser testado.

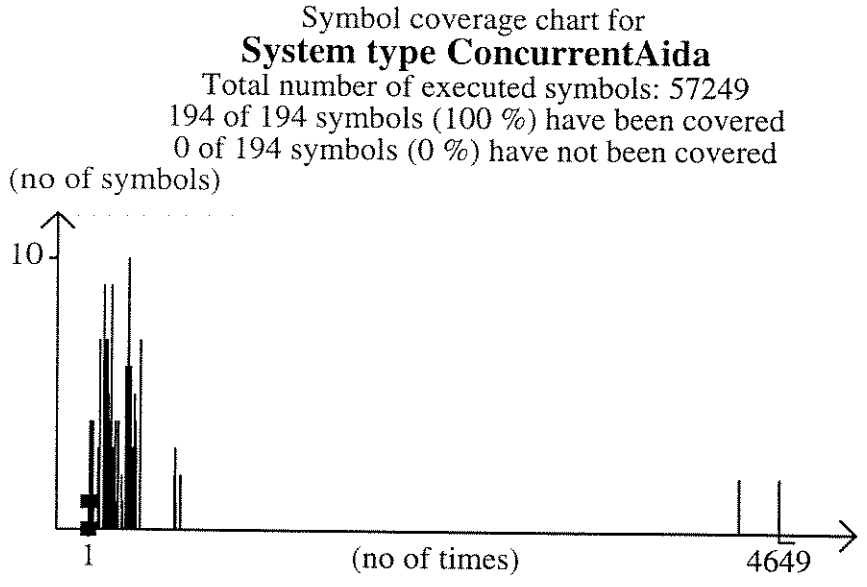


Figura 3.36 - Coverage Viewer Detalhado - AIDA Concorrente

A validação do sistema concorrente da Figura 3.29 é o resultado de outras tentativas de validação em que o número de situações de erros (*reports*) foi diferente de zero, como por exemplo quando da existência de sinais que chegavam para processos que não se encontravam em estados preparados para recebê-los.

Com a identificação destas falhas na especificação é possível retornar aos processos em SDL para sua correção. Em alguns casos, estas correções podem levar a alterações também no modelo de objetos, como aconteceu no AIDA. Através da validação da primeira versão do AIDA Concorrente [Macário, et al., 1997b], percebeu-se a necessidade da criação do processo *Initial* no modelo SDL (Figura 3.9 e Figura 3.22) para iniciar o sistema AIDA e controlar a criação de instâncias *AE* e *AE2* do processo *AnalysisEnv*, levando também à revisão dos modelos OMT (Figura 3.1 e Figura 3.15), onde foi incluída a classe *Initial*.

A próxima seção apresenta alguns exemplos de erros de especificação detectados durante tentativas de validação do sistema concorrente e como foram corrigidos.

3.6.3. Exemplos de erros encontrados e sua correção com o uso de diagramas MSC

A validação de um sistema geralmente leva à identificação de erros existentes na sua especificação, com relação a regras pré-definidas, como a existência de "deadlocks", consumação implícita de sinais, erros na criação de instâncias de processos e erros nos sinais de saída (output). Para facilitar a correção dos erros encontrados, o *SDT Validator* permite a geração de diagramas MSC (Message Sequence Chart) [ITU-T, 1993b], através do *MSC Editor*, que tornam possível a localização exata da situação em que cada erro ocorre. Nesta seção são apresentados alguns exemplos de erros detectados durante o processo de validação do sistema *ConcurrentAIDA*, explicando como localizá-los no sistema e corrigi-los.

O primeiro exemplo de erro trata da consumação implícita de sinais pelo sistema. Este erro ocorre quando um processo envia um sinal para outro que não se encontra num estado preparado para recebê-lo. Neste caso o sinal é implicitamente consumido. A mensagem informada pelo *Validator* durante a validação do AIDA concorrente [Macário, et al., 1997b] foi a seguinte:

Warning: Implicit signal
consumption of signal ConvFile
Sender: AE2:2
Receiver: FC:1

Este erro indica que a instância *AE2:2* do process type *AnalysisEnv* enviou o sinal *ConvFile* para a instância *FC:1* do processo *FileConv*, cuja versão antiga é mostrada na Figura 3.38, que não encontrava-se em um estado preparado para recebê-lo. A Figura 3.37 apresenta o diagrama MSC de parte do caminho de teste da árvore de comportamento onde este erro foi detectado:

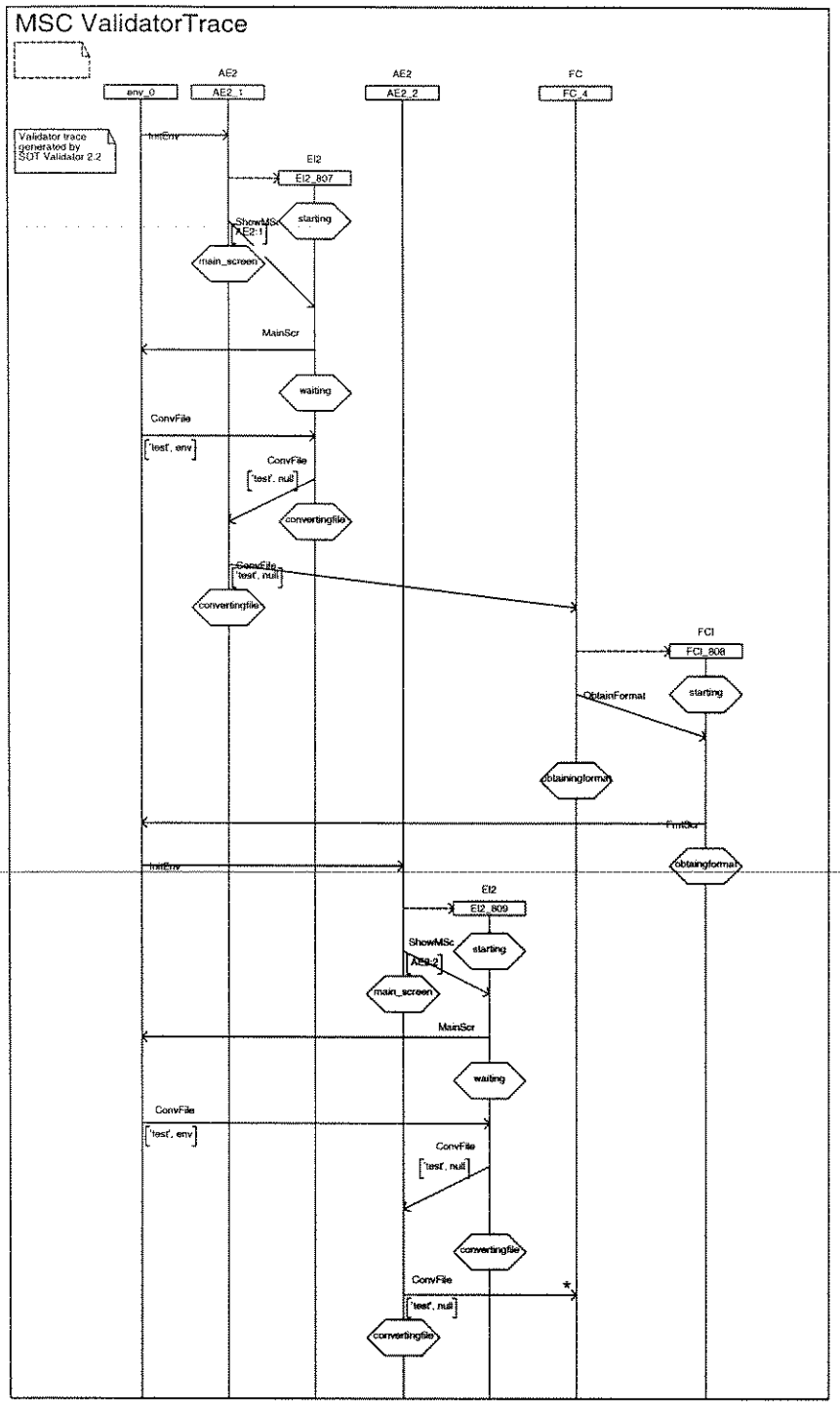


Figura 3.37 - Consumação implícita do sinal ConvFile

Um gráfico ou diagrama MSC representa as trocas de sinais entre as entidades que compõem o sistema, permitindo visualizar o seu comportamento. Neste tipo de diagrama as mensagens representam os sinais e as instâncias representam os processos. Cada instância criada aparece com o seu respectivo Pid, e o ambiente externo ao sistema é representado pela instância env_0.

O MSC da Figura 3.37 apresenta duas instâncias *AE2* do process type *AnalysisEnv* (Figura 3.20), com os seus Pid:*AE2_1* e *AE2_2*. Cada uma destas instâncias criou uma instância *EI2* do process type *EInterf* (*EI2_807* e *EI2_809* respectivamente), correspondendo às suas interfaces. A instância *FC* (Pid *FC_4*) do process type *FileConv,* responsável pela conversão de arquivos, cria a instância *FCI* do process type *FCInterf* (*FCI_808*), que representa sua interface. A situação de erro é representada pela mensagem com * na parte de baixo do diagrama, indicando que o sinal *ConvFile* enviado por *AE2_2* não pode ser consumido por *FC_4*. Verificando-se detalhadamente o diagrama MSC, percebe-se que o usuário do sistema representado pelas instâncias *AE2_1* e *EI2_807*, requisitou à *FC_4* a conversão de um arquivo externo (sinal *ConvFile*). Na recepção deste sinal, a instância *FC_4,* que se encontrava no estado *idle,* cria uma instância *FCI* da sua interface e vai para o estado *obtaining_format,* esperando pelo formato do arquivo externo para realizar a sua conversão. A instância *FCI_808* ao ser criada vai para o estado *starting,* recebe o sinal *ObtainFormat,* devendo requisitar ao usuário o formato do arquivo externo. Após emitir esta requisição (sinal *FmtScr*), também vai para o estado *obtaining_format.* Neste intervalo, um outro usuário (instâncias *AE2_2* e *EI2_809*) inicia o sistema, e também envia uma requisição de conversão de arquivo externo para a instância *FC_4,* que se encontra no estado *obtaining_format,* não está preparado para consumi-lo, levando a uma situação de erro.

Diante destas informações, passou-se, com a ajuda do próprio SDT, à especificação SDL antiga do processo *FileConv* (Figura 3.38). Percebe-se quando o processo se encontra no estado *obtaining_format,* ele está preparado para receber apenas o formato do arquivo a ser convertido (sinal *Format*). A especificação do processo *FileConv* foi corrigida e tomou a forma atual que foi apresentada na Figura 3.13.

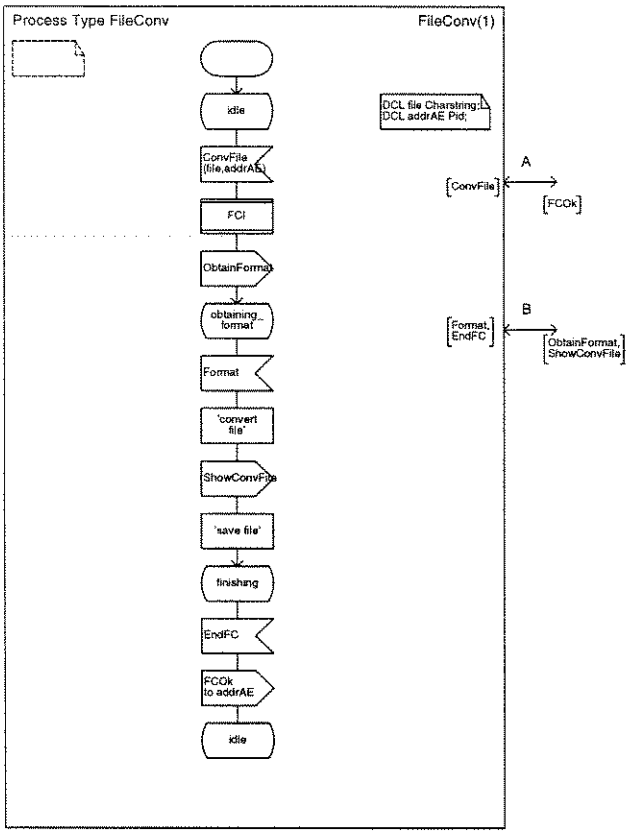


Figura 3.38 - Process Type FileConv (versão antiga)

Outro tipo de erro comum nas especificações SDL ocorre quando, por uma falha na especificação, não existe nenhum caminho para o processo receptor do sinal enviado, ou seja, na especificação do sistema ou do bloco, não foi definida nenhuma rota para um determinado sinal entre os dois processos envolvidos.

Um exemplo deste tipo de erro apresentou a seguinte mensagem:

```
Error in SDL Output of
signal ShowMScr
No path to receiver
Sender: AE:1
Receiver: EI2:2
```

A Figura 3.39 apresenta parte do MSC que detectou este erro. Neste diagrama a instância *AE:1* do process type *AnalysisEnv* enviou o sinal *ShowMScr* para a instância *EI2:2* do processo *EInterf*. Entretanto, na especificação do bloco *AnalisysEnv* não foi definida nenhuma rota de sinal entre estes processos.

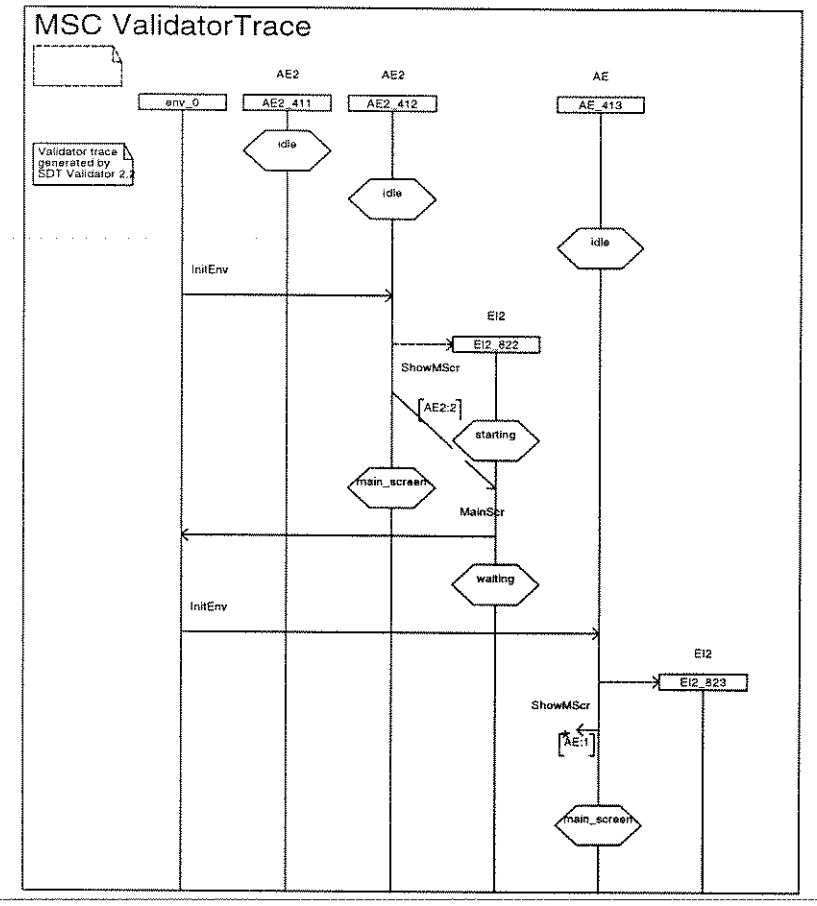


Figura 3.39 - Sinal ShowMScr:nenhum caminho para recepção

O diagrama MSC da Figura 3.39 apresenta as instâncias *AE2_412* e *AE_413* do process type *AnalisysEnv*, onde ambas criaram uma instância *EI2* do process type *EInterf*. A primeira delas criou a instância *EI2_822,* e enviou a ela o sinal *ShowMScr*, requisitando a exibição da tela principal do sistema para o usuário (sinal *MainScr*). Já a instância *AE2_413* criou a instância *EI2_823*, e também enviou para ela o *ShowMScr* para a exibição da tela principal. Neste caso o sinal não foi recebido, por não haver um caminho estabelecido entre estas duas instâncias.

Para identificar a causa deste erro, recorreu-se às especificações do block type *AnalysisEnv* na versão centralizada e concorrente, agora antigas (Figura 3.40 e Figura 3.41), pois são estes diagramas que especificam a comunicação entre as instâncias *AE* e *EI2* dos processos *AnalysisEnv* e *EInterf.* Após uma breve análise, percebe-se que a instância *AE* foi especificada na versão centralizada do sistema, no virtual block type *AnalysisEnv*, onde a instância de interface relacionada a ela é *EI.* Já na especificação da versão concorrente, no redefined block type *AnalysisEnv* foram definidas as novas instâncias *AE2* e *EI2*, que comunicam-se entre si através da rota *AEEI2.* O process type *AnalysisEnv* na sua redefinição, dentre outras coisas, passou a criar uma instância *EI2*

(Figura 3.43) do process type *EInterf*, e não mais *EI* como era na versão centralizada (Figura 3.42). Como o sistema *Concurrent* herda todas as definições do sistema *Centralized*, no início da sua execução existirão, além das novas instâncias, também aquelas definidas pelo sistema herdado. Assim, a instância *AE* também existe nesta versão do sistema, criando uma instância *EI2* da interface, sem ter sido definida nenhuma comunicação entre elas. Ou seja, não existe caminho definido para a sua troca de sinais.

Este erro levou à inclusão do process type *Initial* no bloco *AnalysisEnv* (Figura 3.4 e Figura 3.18), resultando nas especificações centralizada e concorrente atuais. Este processo é responsável por controlar o número de usuários que executam o sistema e por criar instâncias do process type *AnalysisEnv* para cada um destes usuários. Assim, na versão centralizada ele cria uma instância *AE*, que por sua vez cria uma interface *EI*. Na versão concorrente ele é redefinido para criar apenas instâncias *AE2*, as quais criam instâncias *EI2*, deixando de haver a possibilidade de existência de uma instância *AE*, criando uma instância *EI2*.

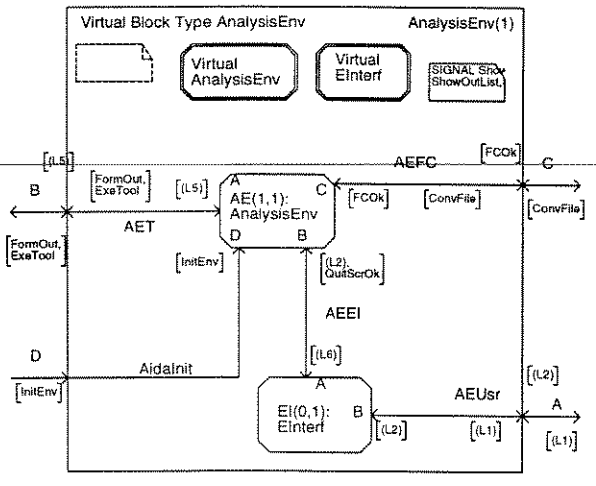


Figura 3.40 - Virtual Block Type AnalysisEnv

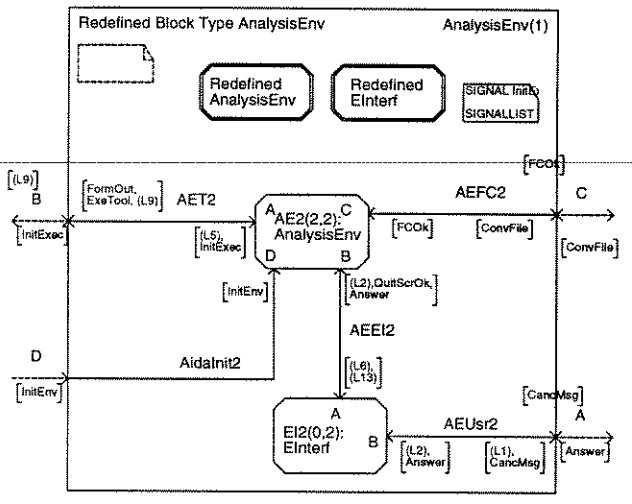


Figura 3.41 - Redefined Block Type AnalysisEnv

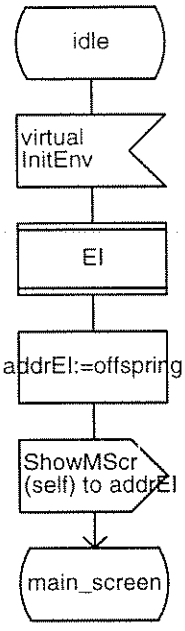


Figura 3.42 - Virtual Process Type AnalysisEnv - parcial

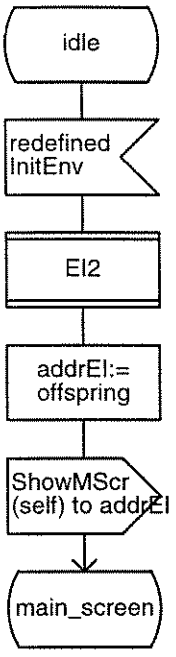


Figura 3.43 - Redefined Process Type AnalysisEnv - parcial

3.7. Exemplo de Simulação do AIDA Concorrente

A simulação de um sistema é feita principalmente para checar se o comportamento descrito na especificação de requisitos do software é coberto pela especificação SDL. Isto é obtido com a utilização de uma ferramenta de simulação que tem como entrada uma descrição de uma execução esperada para o sistema, para o teste de casos julgados críticos. Este processo pode revelar as principais falhas do sistema com relação ao seu comportamento (se existirem), bem como permitir a identificação de erros que violem as regras definidas para teste ("deadlocks", criação de instâncias, etc.).

A ferramenta *SDT Simulator* permite a execução de uma especificação SDL para a investigação interativa do comportamento do sistema especificado. Por exemplo, torna-se possível a execução passo a passo do sistema, permitindo a observação dos estados e valores em cada ponto do sistema. Com a facilidade de enviar sinais do ambiente para o sistema, torna-se possível simular execuções de um sistema, para o teste de suas respostas aos sinais recebidos, permitindo a verificação se os estados, os sinais e os valores de variáveis estão de acordo com os especificados e também acompanhar as variáveis de tempo. Outra possibilidade é estabelecer pontos de parada, como comandos ou funções, de forma a parar a execução em comandos que sejam interessantes para uma

determinada análise, como por exemplo para checar a criação de instâncias de processos. Também é possível a geração de um diagrama MSC como resultado da simulação do sistema. Esta forma gráfica torna mais fácil a visualização dos estados e instâncias.

Foram elaborados vários casos de execução do AIDA para as versões centralizada e concorrente. A simulação da versão centralizada foi feita basicamente para testar a estruturação do AIDA, ou seja, se os sinais estavam sendo passados para os processos corretos, e se o modelo proposto atendia às requisições do ambiente. Já no sistema concorrente foram elaborados casos de teste para verificar a utilização do sistema por mais de um usuário ao mesmo tempo, executando cada um deles várias ferramentas, testando, entre outras coisas o salvamento das requisições não atendidas na fila de sinais do processo *Controller*, ou a tentativa de execução do sistema por um número de usuários acima do permitido.

Um dos exemplos montados para a versão concorrente apresenta o seguinte cenário: O usuário 1 inicia a execução do sistema requisitando a execução de uma ferramenta de análise. Como o processo controlador pode atender até duas requisições simultaneamente (hipótese assumida), e não existe nenhuma ferramenta sendo executada ou relatório sendo formatado, a requisição do usuário é repassada para o servidor correspondente. Neste momento, o usuário 2 também inicia a execução do ambiente AIDA, requisitando a formatação de um relatório existente. O processo controlador repassa esta requisição ao servidor correspondente, e como o número limite de duas requisições em execução foi atingido, qualquer nova requisição que chegar será guardada na fila de requisições deste processo. O usuário 1 emite uma nova requisição, desta vez para formatar um relatório. Como não existem servidores disponíveis, a requisição é salva. O tempo de espera pela resposta a esta requisição é atingido e assim, é emitida uma mensagem para o usuário 1, solicitando manter ou excluir a sua requisição da fila de requisições pendentes. Neste meio tempo, o usuário 2 termina a formatação do seu relatório, disponibilizando um servidor para atender às requisições pendentes. Assim, a requisição de formatar relatório do usuário 1 é atendida.

Parte deste exemplo é apresentado na Figura 3.44, que mostra exatamente a simulação a partir do momento que a nova requisição do usuário 1 é colocada na fila de requisições pendentes.

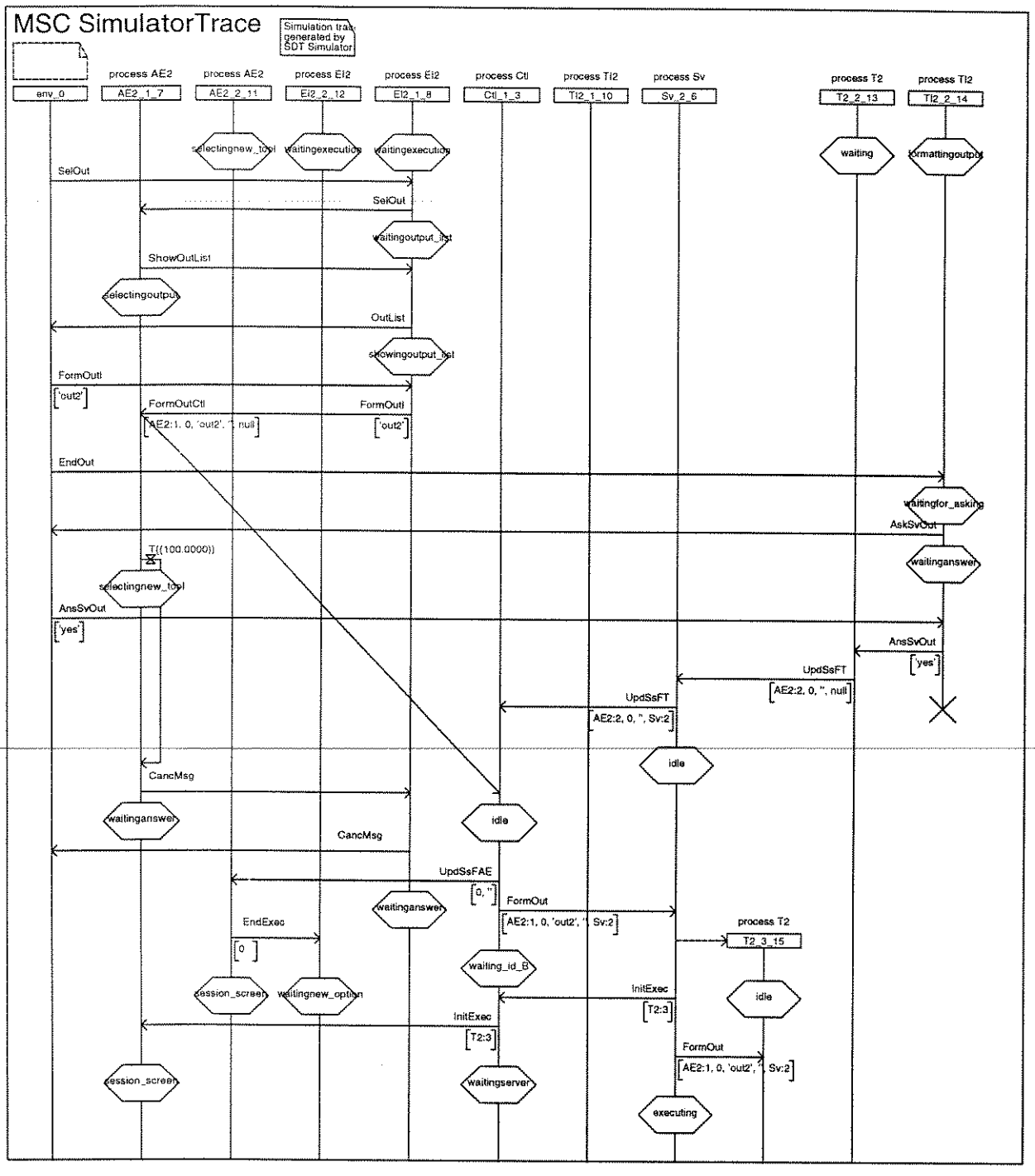


Figura 3.44 - Exemplo de simulação do AIDA Concorrente

O diagrama MSC gerado pelo simulador corresponde ao ponto em que o usuário 1 faz uma nova requisição de execução, desta vez para formatar um relatório (sinal *Outp*). Este sinal é recebido pela instância *EI2_1* que o repassa para a instância *AE2_1*, e que por sua vez envia o sinal *FormOut* para a instância *Ctl_1* do processo *Controller*, ativando o temporizador *T*, indicando um tempo máximo de espera por uma resposta. Como o

processo *Controller* está no estado *waiting_server*, o sinal recebido é salvo na fila de requisições (através da construção SDL SAVE). O tempo de espera se esgota e o usuário 1 é solicitado a cancelar ou a manter a sua requisição de execução. Neste ínterim, o usuário 2 termina a formatação do relatório, e o sinal *UpdSsFT* é enviado para o servidor, e deste para o controlador, indicando que o servidor está disponível para executar nova ferramenta. Na recepção deste sinal, o controlador vai para o estado *idle*, e assim, consome a requisição (*FormOut*) que se encontrava na fila de processos. A requisição de execução é então atendida e a instância *AE2_1* vai para o estado *waiting_new_tool*.

4. AIDA_CORBA Centralizado e Distribuído usando OMT e SDL

4.1. Introdução

As versões centralizada e concorrente do sistema AIDA propostas no capítulo 3 tratam da visão geral do sistema, apresentando uma especificação genérica do comportamento de cada classe em termos de seus modelos OMT e SDL. Este capítulo apresenta a versão AIDA_CORBA, cuja especificação, dentre outras coisas levou ao detalhamento das operações apresentadas por cada classe do sistema. Além disso, esta versão explora algumas funcionalidades do ambiente AIDA não presentes nas versões anteriores, como a utilização de arquivos de dados virtuais e os requisitos necessários para a sua execução em ambiente CORBA [Siegel, 1996], visando a aplicações distribuídas, construídas segundo uma arquitetura cliente-servidor.

Um dos requisitos antigos dos usuários do ambiente AIDA [Embrapa, 1996] era a utilização de arquivos virtuais de dados, ou seja, a possibilidade criar visões sobre os arquivos de dados, já que muitas vezes o arquivo existente não corresponde às suas necessidades. Assim, através da definição de arquivos virtuais, o usuário tem a possibilidade de criar um arquivo de acordo com as análises de dados a serem feitas. Um exemplo bem simples pode ser um arquivo com um informações diferenciadas para homens e mulheres, sendo que o usuário necessita de uma informação conjunta de ambos. Assim, ele define um arquivo com um campo que corresponde ao somatório das informações existentes para homens e mulheres no arquivo original.

Para permitir o tratamento de arquivos virtuais de dados, foram incluídas novas classes nos modelos OMT existentes, e utilizada a linguagem ASN.1 [ITU-T, 1995] para a especificação das estruturas de dados utilizadas na representação destes arquivos.

Neste capítulo são apresentados os modelos OMT e SDL dos sistemas AIDA_CORBA centralizado e distribuído, descrevendo as principais diferenças com relação às versões anteriores do sistema. Também serão apresentados o tratamento de arquivos de dados no modelo SDL e a definição das interfaces dos objetos em linguagem IDL (Interface Definition Language) [Siegel, 1996], que se constituem requisito fundamental da arquitetura CORBA para o suporte a uma aplicação distribuída. Ao final do capítulo são apresentados os resultados da validação do sistema AIDA_CORBA distribuído e um exemplo da sua simulação.

4.2. A Arquitetura CORBA

O Common Object Request Broker Architecture - CORBA [Siegel, 1996] é um padrão que define mecanismos de suporte a aplicações cliente-servidor baseado na chamada a métodos de objetos. Este padrão é definido pelo Object Management Group (OMG) que busca o desenvolvimento de uma arquitetura simples, usando a tecnologia de objetos, para a integração de aplicações distribuídas. Nesta definição deve-se levar em conta a reusabilidade de componentes, a interoperabilidade, a portabilidade, a disponibilidade e os produtos de software comerciais disponíveis.

Basicamente o CORBA permite a comunicação entre processos, independente da sua localização e de sua implementação, conectando apenas objetos, e incorporando a visão de componentes de software do tipo plug-and-play, ou seja, que podem ser facilmente trocados sem alterar os programas que os usam. Na arquitetura CORBA cabe ao ORB (Object Request Broker), estabelecer e endereçar as ligações entre os objetos, e gerenciar todas as requisições feitas. O ORB intercepta a chamada do cliente e é responsável por encontrar o objeto desejado na rede, passar os parâmetros da chamada para ele, invocar o método e retornar os resultados. Ao programa cliente cabe apenas identificar o objeto que possui o método desejado, sem precisar se preocupar onde ele se encontra ou em que linguagem ou sistema ele foi implementado. Desta forma, todo o problema de localização dos servidores passa para o ORB, que faz uso de serviços CORBA para encaminhar adequadamente as requisições do cliente, provendo a interoperabilidade entre aplicações de diferentes máquinas, em diferentes sistemas.

Esta interoperabilidade entre sistemas diferentes é possível devido à definição das interfaces dos objetos disponíveis através de uma linguagem comum denominada IDL (Interface Language Definition) também definida pelo consórcio internacional OMG, e que descreve as operações que o objeto está preparado para executar, seus parâmetros de entrada e saída, e qualquer exceção que pode ser gerada ao longo do tempo.

A linguagem de especificação IDL [Siegel, 1996] deve funcionar como um contrato entre o cliente e o servidor, onde o servidor informa a todos sobre os serviços que oferece, e o cliente fica sabendo quais os serviços que estão disponíveis. Assim, através do uso da especificação das interfaces dos objetos em IDL, torna-se possível aos clientes acessar remotamente, e de maneira transparente, os objetos existentes na rede, sem a preocupação com detalhes de sua implementação.

Para que seja possível a localização dos objetos na rede, cada objeto em CORBA apresenta sua própria referência de objeto, a qual é única naquele ORB e é atribuída na inserção do objeto ambiente, e que mantém-se válida até que o mesmo seja explicitamente excluído. Os clientes obtêm esta referência, através de serviços disponíveis, e a associam à invocação do objeto desejado, permitindo ao ORB direcionar a requisição para o objeto correto. Cada ORB armazena todas as interfaces dos seus objetos em um repositório de interface, e de outros ORBs também, permitindo a ele traduzir a ordem dos bytes e o formato dos dados de uma requisição sempre que necessário. Para que seja possível a interoperabilidade entre os sistemas, é necessário que todas as interfaces dos objetos sejam descritas utilizando a mesma linguagem OMG IDL.

Assim, a arquitetura CORBA oferece ao usuário o acesso transparente a informações, sem que seja necessário o conhecimento do software ou hardware que o suporta ou a sua localização.

O Apêndice B apresenta algumas informações adicionais sobre a arquitetura CORBA.

4.3. Mapeamento OMT - SDL92 para o AIDA_CORBA

Para que um sistema possa executar em um ambiente CORBA, é necessária a especificação dos serviços oferecidos por cada um de seus componentes que estarão distribuídos pela rede. Esta especificação é feita em linguagem IDL, e descreve as interfaces (operações) dos objetos que terão métodos transitando pela rede. As interfaces IDL podem ser obtidas a partir do modelo OMT, que neste caso deve conter a descrição detalhada das funcionalidades e dos atributos de cada classe.

Os modelos de classes das versões centralizada e concorrente (cap. 3) consistiam em descrições do sistema com um alto nível de abstração, não sendo detalhadas quais eram as operações específicas de cada classe. Por exemplo, a classe *AnalysisEnv* do AIDA centralizada (Figura 3.1) tinha como funcionalidades *manipular ferramentas* e *executar as opções do ambiente selecionadas pelo usuário*, descrevendo de maneira genérica as suas operações. Para permitir a especificação IDL das interfaces dos objetos do AIDA numa tradução quase direta, o modelo de classes desta versão, denominada AIDA_CORBA, passa a apresentar as operações detalhadas de cada uma de suas classes.

A evolução do modelo OMT levou também à revisão das especificações SDL anteriores (cap. 3), que passam a incorporar em seus processos a construção PROCEDURE

para a definição de procedimentos, os quais foram utilizados no mapeamento específico das operações de cada classe [TELELOGIC AB, 1996c][TELELOGIC AB, 1997].

O mapeamento de OMT para SDL-92 usado pelo AIDA_CORBA é apresentado na Tabela 4.1.

Modelo OMT	Representação em SDL
Sistema	System type
Classes dinâmicas (Initial, AnalysisEnv, etc.)	Process type
Operações das classes	Procedures
Classes passivas (Output, SessionFile, DataFile) e atributos de classes	SORTS
Módulos de classes	Block types
Objetos de classes dinâmicas	Instâncias dos process types
Objetos das classes passivas	instâncias de SORTS, definidas através de variáveis nos process types
Invocação de métodos	Signals

Tabela 4.1 - Mapeamento OMT para SDL-92 para o AIDA_CORBA

Com a adoção deste mapeamento os processos passam a especificar o comportamento geral de cada classe em termos de suas operações, tornando mais fácil o seu entendimento. Uma outra vantagem decorrente desta modificação é que as eventuais evoluções do sistema tornam-se atividades mais simples de serem realizadas, já que estas alterações geralmente são localizadas e agora ficam restritas aos procedimentos, não sendo necessária a revisão total de cada processo.

Além disso, o uso de procedimentos em SDL para mapear as operações das classes do modelo OMT, permite a identificação exata do que foi alterado, ou incluído, para a especialização de classes, já que na evolução de sistemas SDL apenas as diferenças entre o sistema especializado e o herdado devem ser especificadas. Na versões anteriores, que apresentam um visão geral de cada classe, boa parte dos processos eram rescritos na evolução do sistema, não deixando claro quais de suas funcionalidades propriamente ditas haviam sido alteradas.

Na definição das operações das classes no modelo OMT usou-se o conceito de polimorfismo, ou seja, a existência de operações com o mesmo nome em classes

diferentes. Na linguagem SDL esta funcionalidade algumas vezes não é possível de ser mapeada por duas razões:

- as operações são implementadas como PROCEDURES, e a linguagem não permite a definição de mais de uma com o mesmo nome;
- a invocação destas operações se dá através dos sinais, e caso os parâmetros das operações homônimas não sejam os mesmos, o que ocorre na maioria dos casos, não é possível o reuso dos sinais.

Assim, adotou-se como convenção incluir as iniciais das instâncias do processo nos nomes de suas procedures e de seus sinais de invocação, neste caso somente se os parâmetros forem diferentes. Por exemplo, as classes *AnalysisEnv* e *EInterf* possuem uma operação *InitSs*, com ações para iniciar uma sessão de trabalho no AIDA_CORBA. No modelo SDL estas operações são mapeadas pelas procedures *InitSsAE* e *InitSsEI*, incluindo nos seus nomes as iniciais de suas instâncias *AE* e *EI* respectivamente. Como neste caso os parâmetros de ambas são os mesmos, a sua invocação se dá através do sinal *InitSs*.

O sistema AIDA_CORBA distribuído poderia ser especificado diretamente sem a herança de classes. Como a intenção era manter o conceito de reuso, e verificar o comportamento do modelo SDL na redefinição de classes, da mesma forma que o AIDA concorrente (cap. 3), que é especificado a partir de uma versão centralizada, a versão distribuída para o ambiente CORBA é elaborada a partir de um modelo AIDA_CORBA centralizado. Nas seções seguintes serão apresentados os modelos OMT e SDL para os sistemas AIDA_CORBA centralizado e distribuído, denominados SingleUserAIDACORBA e DistributedAIDACORBA respectivamente, bem como a especificação de suas interfaces externas em linguagem IDL para o seu uso em ambiente CORBA e o tratamento de arquivos de dados virtuais utilizando a linguagem ASN.1.

4.4. Modelo OMT do AIDA_CORBA Centralizado

O modelo de classes apresentado na Figura 4.1 consiste numa evolução do modelo OMT do AIDA centralizado (seção 3.2.), descrevendo as funcionalidades de cada uma de suas classes detalhadamente.

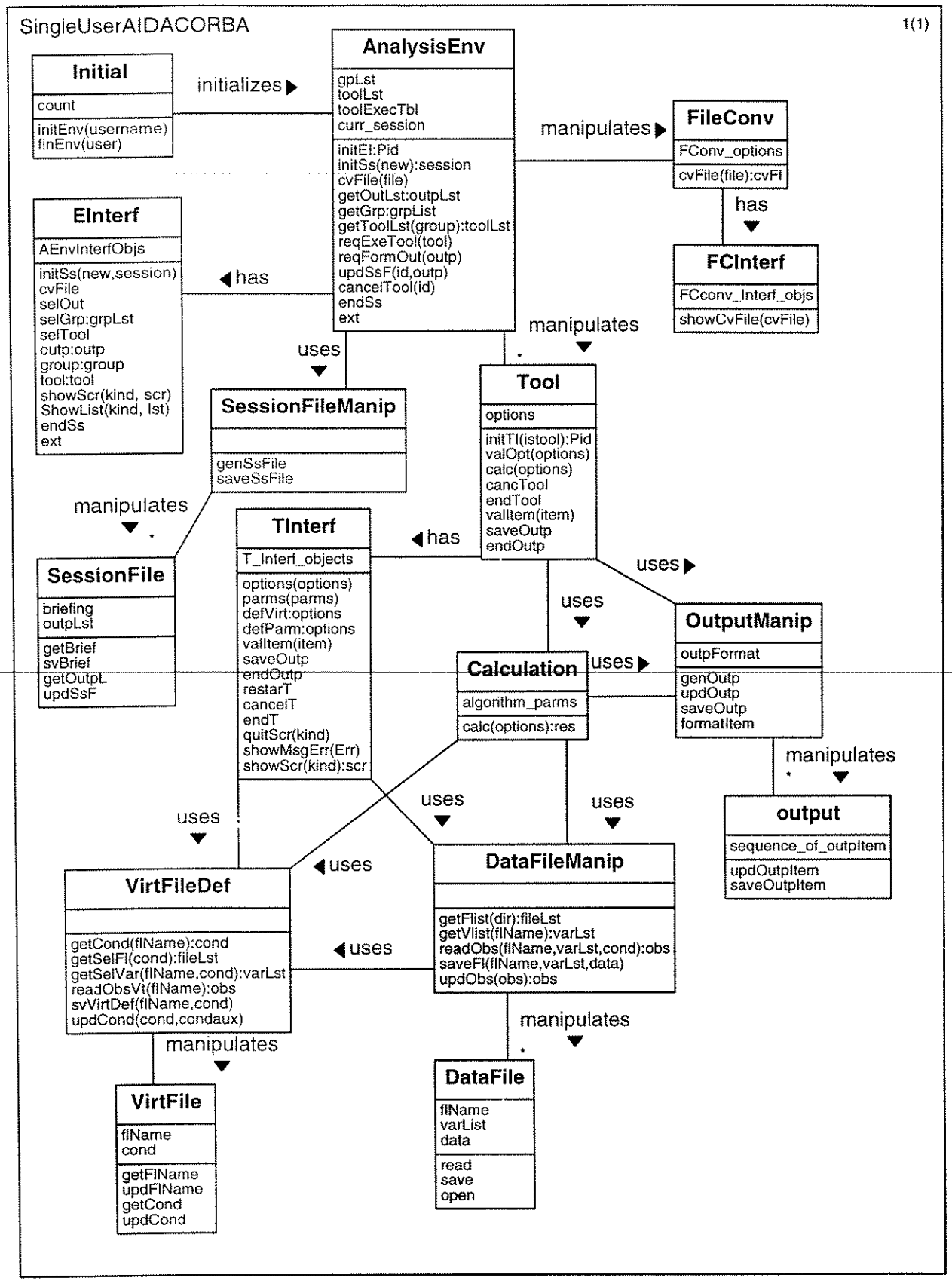


Figura 4.1 - Modelo OMT do SingleUserAIDACORBA

Na evolução para a nova versão centralizada do AIDA da Figura 4.1, foram identificadas as principais operações que cada classe deve apresentar para realizar as funcionalidades descritas no modelo OMT da versão centralizada anterior (Figura 3.1). Estas operações em cada classe são:

Classe Initial (Iniciador do sistema AIDA)

initEnv - inicia a execução do AIDA para um usuário;

finEnv - finaliza a execução do sistema por um usuário.

Classe AnalysisEnv (Gerenciador do ambiente de análise)

initEI - inicia a interface do gerenciador;

initSs - inicia uma sessão de trabalho já existente ou não;

cvFile - converter um arquivo de dados externo ao AIDA;

getOutpLst - obtém lista de relatório da sessão de trabalho corrente;

getGrpLst - obtém lista de grupos de ferramentas de análise;

getToolLst - obtém lista de ferramentas de um determinado grupo de análise;

reqExeTool - requisição para execução de ferramenta de análise;

reqFormOut - requisição para formatação de relatório;

updSsFl - atualiza o arquivo de sessão com o resultado da análise ou com a formatação de relatório;

cancelTool - execução de uma ferramenta foi cancelada; atualiza a tabela de ferramentas em execução;

endSs- finaliza uma sessão de trabalho;

ext - termina a execução do AIDA.

Classe EInterf (Interface gráfica do gerenciador do ambiente de análise)

initSs - requisição para iniciar uma sessão de trabalho já existente ou não;

cvFile - requisição para converter um arquivo de dados externo ao AIDA;

selOutp - requisição para exibir lista de relatório da sessão de trabalho corrente para a seleção de um deles para formatação;

selGrp - requisição para exibir lista de grupos de ferramentas de análise para a seleção de um deles;

selTool - requisição para exibir lista de ferramentas de um determinado grupo de análise para a seleção de uma delas;

group - obtém grupo de ferramentas análise selecionado para execução;

tool - obtém a ferramenta de análise selecionada para execução;

outp - obtém o relatório selecionado para ser formatado;

showList - exibir listas na tela;

showScr - exibir telas;

endSs- requisição para finalizar uma sessão de trabalho;

ext - requisição para terminar a execução do AIDA.

Classe Tool (Ferramenta de análise e formatador de relatório)

initTl - inicia a interface da ferramenta ou do formatador de relatórios;

valItem - valida cada item alterado do relatório sendo formatado;

calc - executa o cálculo do algoritmo matemático;

cancTool - cancelamento da execução da ferramenta;

endTool - término da execução da ferramenta;

valOpt - valida as opções de execução obtidas através da interface;

saveOutp - salva o relatório formatado;

endOutp - término de formatação de um relatório.

Classe TInterf (Interface gráfica da ferramenta de análise ou do formatador de relatório)

options - obtém as opções de execução do usuário;

parms - obtém os parâmetros de execução do usuário;

defVirt - define o arquivo virtual de dados;

defParms - define os parâmetros de execução da ferramenta;

formItem - requisição para validar o item alterado do relatório;

saveOutp - requisição para salvar o relatório;

endOutp - requisição de término de formatação de um relatório;

restart - requisição de reiniciação da execução da ferramenta;

cancelT - requisição de cancelamento da execução da ferramenta;

endT - requisição de término da execução da ferramenta;

quitScr - termina a interface da ferramenta ou do formatador;

showMsgErr - exibe as mensagens de erros da execução da ferramenta ou formatação de relatório;

showScr - exibe as telas da ferramenta ou do formatador.

Classe Calculation (Cálculo dos algoritmos específicos da ferramenta)

calc - executa o cálculo do algoritmo da ferramenta, fazendo a leitura de arquivos de dados.

Classe FileConv (Conversor de arquivos externos)

cvFile - converte um arquivo de dados externo para o formato do AIDA.

Classe FCInterf (Interface gráfica do conversor de arquivos externos)

showCvFl - exibe o arquivo de dados convertido.

Neste modelo também foram incluídas as classes *VirtFileDef*, *DataFileManip*, *VirtFile* e *DataFile*. A classe *VirtFileDef* representa as operações para a especificação de um arquivo virtual e a *DataFileManip*, as operações sobre um arquivo de dados comum. A classe *VirtFile* representa a definição de um arquivo virtual e a classe *DataFile* representa o arquivo de dados em si, podendo ser de entrada ou de saída. As operações apresentadas por cada uma delas são as seguintes:

Classe VirtFileDef (Operações para a definição de um arquivo de dados virtual)

getCond - obtém a condição de definição do arquivo virtual;

getSelFl - obtém, a partir de *cond*, os arquivos selecionados para a definição do arquivo virtual;

getSelVar - obtém, a partir de *cond*, as variáveis selecionadas para a definição do arquivo virtual;

readObsVt - lê uma observação do arquivo virtual; utiliza a procedure *readObs* da classe DataFileManip;

svVirtDef - salva a definição de um arquivo virtual; usa a procedure *saveFl* da classe DataFileManip ;

updCond - atualiza a condição de definição do arquivo virtual.

Class DataFileManip (Manipulador de um arquivo de dados)

getFlist - obtém lista de arquivos existente em um diretório;

getVlist - obtém lista de variáveis de um arquivo;

readObs - lê uma observação do arquivo de dados real;

saveFl - salva um arquivo de dados ;

updObs - atualiza uma observação lida.

Class VirtFIle (Definição do arquivo virtual)

getFlName - obtém o nome do arquivo virtual;

updFlName - salva nome do arquivo virtual;

getCond - obtém a condição de definição do arquivo virtual;

updCond - salva a condição de definição do arquivo virtual.

Class DataFileManip (Arquivo de dados)

read - lê uma variável do arquivo de dados;

open - abre o arquivo para leitura;

save - salva um arquivo de dados.

4.5. Especificação SDL do AIDA_CORBA Centralizado

Na versão anterior do sistema AIDA (Cap. 3) os processos eram utilizados para apresentar uma visão geral da funcionalidade de cada classe, sem o detalhamento de suas funções. Nesta versão do sistema cada classe é descrita em termos de suas operações, e para representar esta operações em SDL utiliza-se a estrutura PROCEDURE, o que permite um mapeamento direto do seu modelo de classes [TELELOGIC AB, 1996c][TELELOGIC AB, 1997].

Os procedimentos (procedures) em SDL funcionam como na maioria das linguagens de programação, sendo utilizados quando partes de um processo se repetem, ou para se ter uma melhor visão do sistema. Em SDL todos os procedimentos definidos em um processo são visíveis uns aos outros. Além disso, as variáveis definidas em um processo são visíveis aos seus procedimentos, o que possibilita a sua utilização para representar os atributos de uma classe que devem ser visíveis às suas operações. Um procedimento pode ter parâmetros de entrada e de entrada e saída, e valor de retorno. Os parâmetros de entrada são definidos como *in*, os de entrada e saída como *in/out*, e o valor de retorno por *returns*. No sistema *SingleUserAIDACORBA*, os blocos que representam agrupamentos lógicos de classes segundo suas funcionalidades, tiveram seus nomes revisados para não haver confusão com o nome dos processos homônimos. Assim, os nomes dos blocos sofreram as seguintes alterações:

Antigo Nome no CentralizedAIDA (Figura 3.3 - Cap. 3)	**Novo Nome no SingleUserAIDACORBA** (Figura 4.3)
Block Type AnalysisEnv	Block Type Manager
Block Type Tool	Block Type ToolExecution
Block Type FileConv	Block Type FileConvertion

Os nome das instâncias destes blocos também foram alterados de *AE*, *T* e *FC* para *Mng*, *TExec* e *FCvt*, respectivamente.

Com estas atualizações dos nomes dos blocos e de suas instâncias, e segundo a Tabela 4.1, o mapeamento do modelo OMT do AIDA_CORBA centralizado (Figura 4.1) para a linguagem SDL é apresentado na Tabela 4.2.

Classes OMT	Representação em SDL
AIDA	System Type SingleUserAIDACORBA, composto pelos block types Manager, FileConvertion e ToolExecution.
Initial	process type Initial composto pelas procedures initEnv e finEnv.
AnalysisEnv	process type AnalysisEnv composto pelas procedures initEI, initSs, cvFile, getOutp, getGrp, getTool, reqExeTool, reqFormOut, updSsFI, cancelTool, endSs e ext.
EInterf	process type EInterf composto pelas procedures initSs, showList, showSc, endSs e ext.
Tool	process type Tool composto pelas procedures initTI, valItem, calc, cancTool, endTool, valOpt, saveOutp e endOutp.
TInterf	process type TInterf composto pelas procedures defVirt, defParms, formItem, saveOutp, endOutp, quitScr, showMsgErr e showScr.
Calculation	process type Caculation composto pela procedure calc.
FileConv	process type FileConv composto pela procedure cvFile.
FCInterf	process type FCInterf composto pela procedure showCvFI.
VirtFileDef	process type VirtFileDef composto pelas procedures getCond, getSelFI, getSelVar, readObsVt, svVirtDef e updCond.
VirtFile	classe passiva a ser representada com linguagem ASN.1.
DataFileManip	process type DataFileManip composto pelas procedures getFlist, getVlist, readObs, saveFI e updObs.
DataFile	classe passiva a ser representada com linguagem ASN.1.
Output, SessionFile	SORTS
OutputManip, SessionFileManip	classes a serem representadas como operações sobre os SORTS.

Tabela 4.2 - Mapeamento de OMT para SDL do sistema AIDA_CORBA Centralizado

Assim como na versão apresentada no capítulo 3, aqui também foi criado um package (Figura 4.2) para permitir que versão distribuída pudesse reusar as definições do system type SingleUserAIDACORBA.

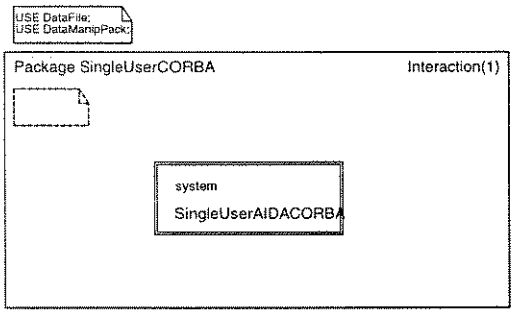


Figura 4.2 - Package SingleUserCORBA

O sistema SingleUserAIDACORBA, que é mostrado na Figura 4.3, da mesma forma que na versão centralizada anterior, também é composto por 3 block types: Manager, ToolExecution e FileConvertion, cujas instâncias são Mng, TExc e FCt respectivamente.

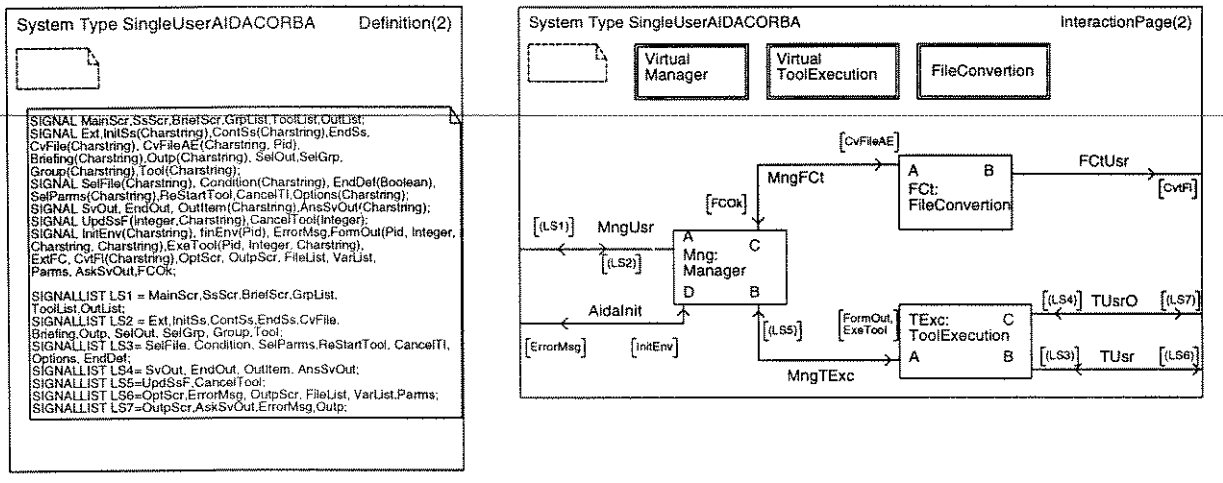


Figura 4.3 - System Type SingleUserAIDACORBA

A especificação SDL de cada um destes blocos é apresentada nas Figura 4.4, Figura 4.5 e Figura 4.6. As funcionalidades destes blocos, com exceção do block type *ToolExecution*, não foram alteradas com relação à versão anterior. A nova versão do bloco *ToolExecution* (Figura 4.5) teve a inclusão das instâncias do *process type VirtFileDef* e do *process type DataFileManip* para o tratamento de arquivos virtuais de dados. Estes processos foram definidos no *package DataManip*. O seu detalhamento é apresentado na seção 4.8 - Tratamento de Arquivos virtuais de Dados.

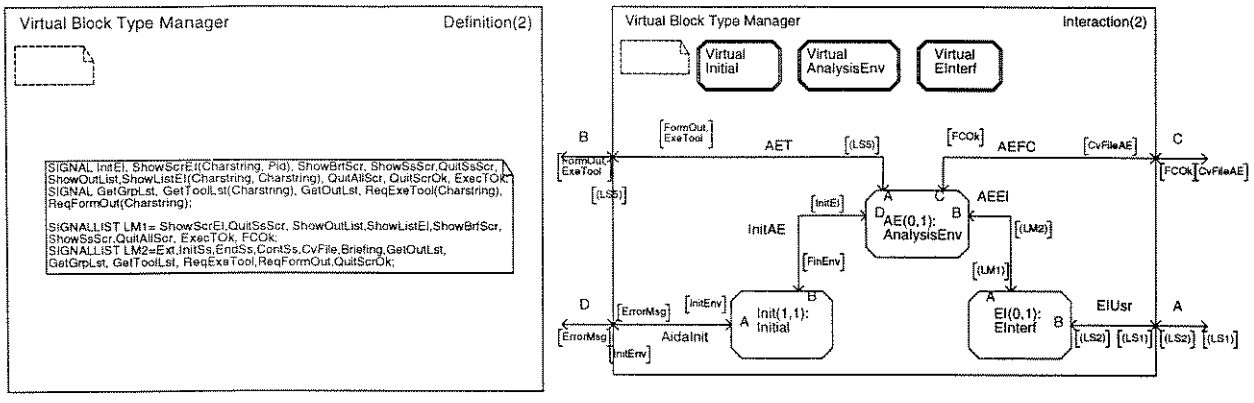


Figura 4.4 - Block Type Manager / SingleUserAIDACORBA

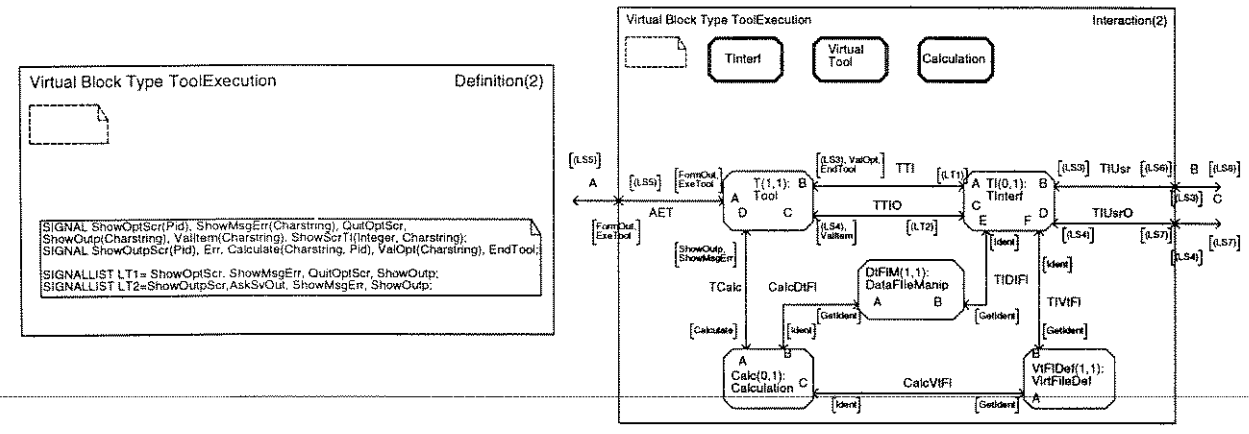


Figura 4.5 - Block Type ToolExecution / SingleUserAIDACORBA

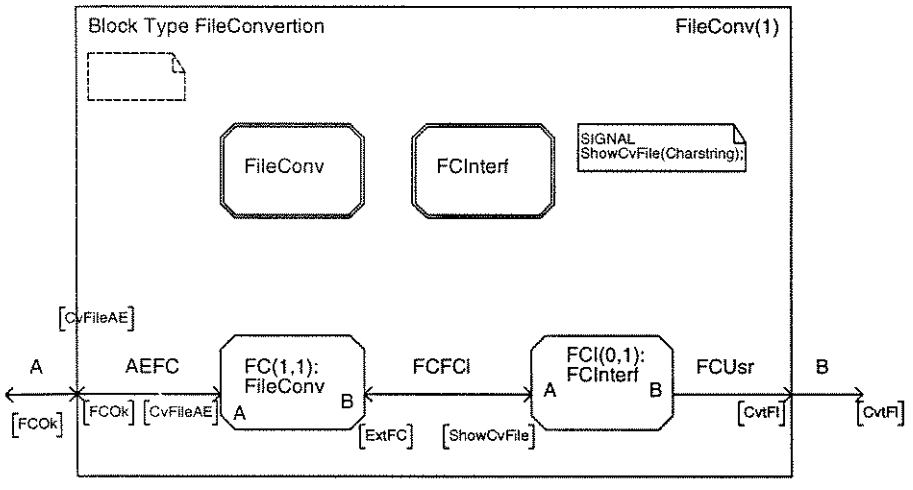


Figura 4.6 - Block Type FileConvertion / SingleUserAIDACORBA

Por outro lado, todas as especificações dos processos propriamente ditos foram revistas para a inclusão das PROCEDURES, de acordo com a seção 4.4.

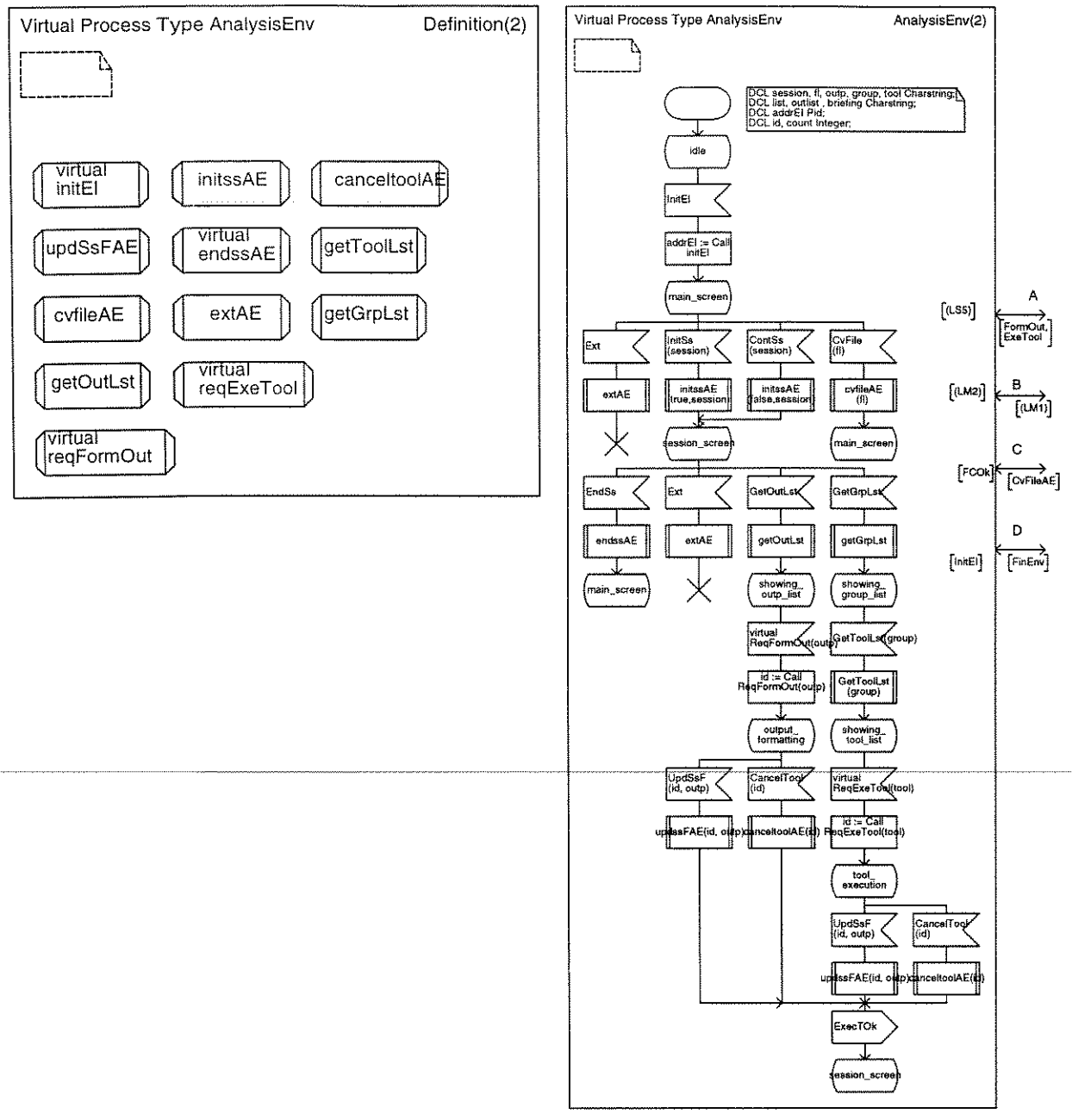


Figura 4.7 - Virtual Process Type AnalysisEnv / SingleUserAIDACORBA

O uso de PROCEDURES tornou mais fácil o entendimento dos processos, já que eles ficaram bem menores devido à sua modularização. Da mesma forma que um process type, uma procedure pode ser redefinida através do mecanismo VIRTUAL/REDEFINED. Assim, a maioria das transições declaradas como VIRTUAL no sistema centralizado anterior, foram mapeadas como VIRTUAL PROCEDURES. Para exemplificar a modularização de um processo em procedimentos, na Figura 4.7 é mostrado o processo *AnalysisEnv* na versão SingleUserAIDACORBA.

Na versão AIDA_CORBA os processos foram restruturados e suas especificações agora são feitas em termos de seus procedimentos. Quando um sinal é recebido, a procedure de mesmo nome é invocada. Por exemplo, quando o processo *AnalysisEnv* encontra-se no estado *idle* e recebe o sinal *InitEI*, a procedure *initEI*, que é apresentada na Figura 4.8, é iniciada.

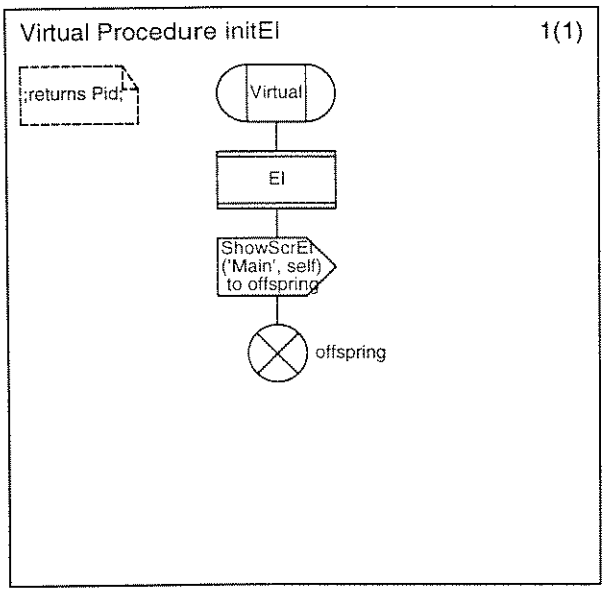


Figura 4.8 - Virtual Procedure initEI / Processo AnalysisEnv

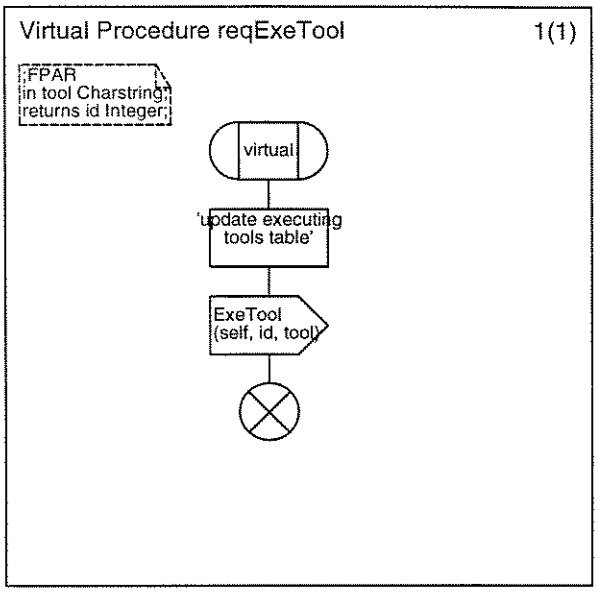


Figura 4.9 - Virtual Procedure reqExeTool / Processo AnalysisEnv

Analisando-se a Procedure *initEI*, percebe-se que corresponde exatamente à transição existente no processo *AnalysisEnv* da versão centralizada anterior (Figura 3.7) após a recepção do sinal *InitEI*. Basicamente esta procedure cria uma instância *EI* do processo de interface (*EInterf*), e envia para ela o sinal *ShowScrEI('Main',self)*, requisitando a exibição da tela principal do sistema. A execução desta procedure retorna o identificador (*self*) da interface criada.

Existem duas maneiras de se invocar uma procedure em SDL:

- através do símbolo `PROCEDURE CALL`, usado para invocar uma procedure que não retorna valor, e
- através da construção `TASK` e da construção `CALL`, para procedures que retornam valores.

Como a procedure *initEI* retorna um valor, ela é invocada no processo *AnalysisEnv* (Figura 4.7) pelo segundo mecanismo, tendo o seu valor de retorno atribuído à variável *addrEI*. As procedures *reqFormOut* e *reqExeTool* (Figura 4.9) são invocadas da mesma

forma. Já a emissão de requisição para atualizar o arquivo de sessão, que se dá nos estados *formatting_output* ou *executing_tool* com a recepção do sinal *UpdSsF(id, outp)*, onde a procedure *UpdSsF* é invocada através do símbolo PROCEDURE CALL, informando-se o nome índice da ferramenta na tabela de ferramentas em execução, e o nome do relatório a ser incluído na arquivo de sessão. As demais procedures deste processo são invocadas da mesma maneira. A Tabela 4.3 relaciona as procedures deste processo, os estados em que são invocadas e os seus respectivos sinais de invocação.

Procedure	Estado	Sinal
InitEI	idle	initEI
extAE	main_screen e session_screen	Ext
initSsAE	main_screen	initSs(kind, session)
initSsAE	main_screen	ContSs(kind, session)
endSsAE	session_screen	EndSs
getOut	session_screen	GetOut
getGrp	session_screen	GetGrp
reqFormOut	showing_outp_list	ReqFormOut(outp)
getTool	showing_group_list	GetTool((group)
reqExeTool	showing_tool_list	ReqExeTool(tool)
updSsFAE	output_ formatting e tool_execution	UpdSsF(id,outp)
cancelToolAE	output_ formatting e tool_execution	CancelTool(id)

Tabela 4.3 - Procedures e sinais de invocação do Processo AnalysisEnv / SingleUserAIDACORBA

4.6. Modelo OMT do AIDA_CORBA Distribuído

O modelo OMT do AIDA_CORBA distribuído é uma evolução do modelo concorrente (Figura 3.15), com a inclusão de funcionalidades para o tratamento de arquivos virtuais de dados e para a execução do sistema em ambiente CORBA, além da inclusão de operações correspondentes nas suas classes.

Na versão concorrente, a classe *AnalysisEnv* emitia uma requisição de execução da ferramenta (*Tool*) para a classe *Controller*, responsável por controlar a disponibilidade dos servidores de processamento existentes, que repassava a requisição diretamente à classe *Server*. Assim, naquela versão, a execução de uma ferramenta se dava da seguinte forma:

- *AnalysisEnv* solicitava a *Controller* a execução de uma ferramenta e ficava esperando pelo resultado da execução;

- *Controller* verificava a disponibilidade de algum *Server* de execução, e caso positivo, repassava a requisição de execução da ferramenta, avisando *AnalysisEnv* de que sua requisição foi atendida;

- *Server* instanciava *Tool* para execução, e ao seu término, retornava o resultado para o *Controller*;

- *Controller* retornava resultado a *AnalysisEnv*;

- *AnalysisEnv* atualizava a tabela de ferramentas em execução e o arquivo de sessão, se fosse o caso.

Nesta evolução (AIDA_CORBA), a classe *Controller* passa a ser responsável apenas por controlar os servidores de processamento (*Server*), cabendo agora à classe *AnalysisEnv* a comunicação direta com a classe *Server* para a execução de uma ferramenta de análise, como por exemplo a emissão da requisição de execução e obtenção do resultado. Adotando esta nova configuração, os passos para execução de uma ferramenta no AIDA_CORBA distribuído são os seguintes:

- *AnalysisEnv* verifica com *Controller* a disponibilidade de um *Server* de processamento para a execução de uma ferramenta;

- *Controller* retorna para *AnalysisEnv* o identificador do *Server* disponível;

- *AnalysisEnv* emite uma requisição de execução de ferramenta para o *Server* correspondente;

- *Server* instancia *Tool* para execução, e ao seu término, retorna o resultado para *AnalysisEnv*;

- *AnalysisEnv* atualiza tabela de ferramentas em execução e o arquivo de sessão, se for o caso, e avisa *Controller* qual *Server* foi liberado.

Assim, o modelo OMT para o sistema AIDA_CORBA distribuído é mostrado na Figura 4.10:

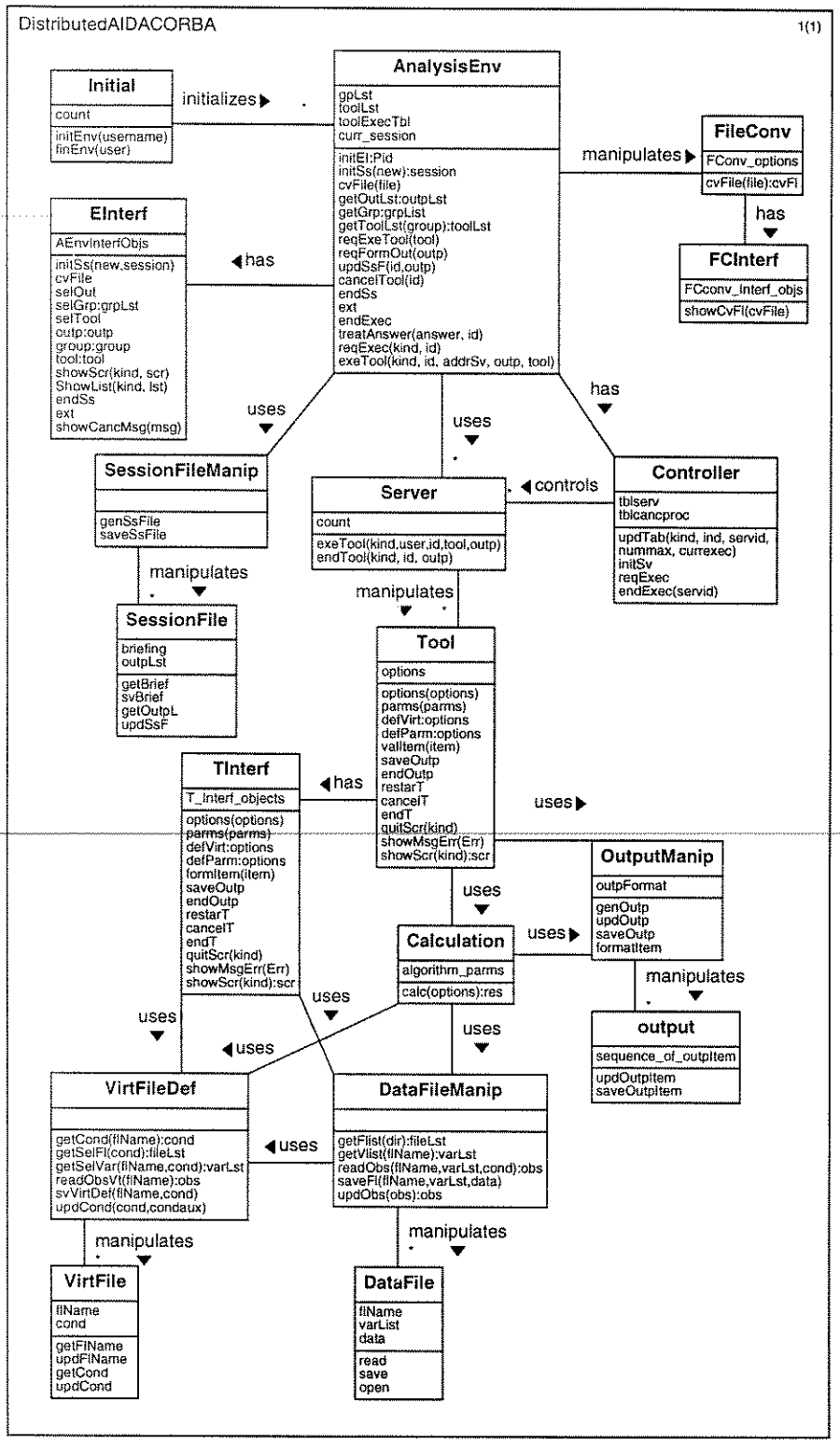


Figura 4.10 - Modelo OMT do DistributedAIDACORBA

O modelo OMT da Figura 4.10 é uma evolução daquele mostrado na Figura 4.1, para o sistema AIDA_CORBA centralizado. Nesta evolução foram incluídas as classes

Controller e *Server*, e operações adicionais em algumas das classes existentes (AnalysisEnv e EInterf). As novas características (operações e classes) deste modelo são:

Classe AnalysisEnv

endExec - chamada ao término de execução de uma ferramenta de análise ou da formatação de um relatório; avisa Controller sobre a liberação de um servidor de processamento;

treatAnsw - chamada quando o tempo de espera pela disponibilidade de servidor de processamento para a execução de ferramenta se esgota; solicita ao usuário manter ou excluir a requisição na fila de requisições;

reqExec - requisita à classe Controller um servidor de processamento para a execução de ferramenta ou formatação de relatório;

exeTool - envia uma requisição de execução ao servidor de processamento.

Classe EInterf

showCancMsg - exibe mensagem solicitando ao usuário manter ou excluir a requisição de execução na fila de requisições.

Classe Controller

updTab - atualiza a tabela de controle de servidores para o processamento de ferramentas;

initSv - inicia os servidores de processamento, atualizando a tabela de controle de servidores;

reqExec - verifica a disponibilidade de servidores;

endExec - disponibiliza um servidor de processamento, atualizando a tabela de controle de servidores.

Classe Server

exeTool - executa uma ferramenta de análise ou formata um relatório;

endTool - término de execução de uma ferramenta ou relatório.

4.7. Especificação SDL do AIDA_CORBA Distribuído

O uso de procedures na especificação em SDL dos processos no SingleUserAIDACORBA (seção 4.5) tornou mais fácil a sua evolução para a versão distribuída, já que as alterações geralmente são bem específicas, levando apenas à redefinição de alguns procedimentos, ou à inclusão de outros. Assim, os processos redefinidos tornaram-se bem mais concisos que os apresentados na versão do AIDA

concorrente (Figura 3.17), onde algum deles foram quase totalmente rescritos quando da sua evolução da versão centralizada correspondente (Figura 3.3).

Como o sistema AIDA_CORBA distribuído herda o sistema AIDA_CORBA centralizado, o modelo SDL da versão distribuída deve especificar apenas as diferenças existentes entre os sistemas. Estas diferenças podem ser a inclusão de novos processos e/ou procedures, representando as novas características do modelo OMT, ou a redefinição de elementos existentes no sistema herdado, indicando a sua evolução, como por exemplo o processo *Initial* foi redefinido para permitir a execução do sistema por mais de um usuário ao mesmo tempo. A Tabela 4.4 mostra o mapeamento OMT-SDL para a versão AIDA_CORBA distribuído, onde apenas estas diferenças são especificadas.

Classes OMT	Representação em SDL
AIDA	System Type DistributedAIDACORBA, composto pelos redefined block types Manager e ToolExecution
Initial	redefined process type Initial composto pela redefined procedure initEnv
AnalysisEnv	redefined process type AnalysisEnv composto pelas redefined procedures initEI, reqExeTool, reqFormOut, endSsAE e pelas novas procedures exeToolAE, treatAnsw, reqExecAE e endExecAE,.
EInterf	redefined process type EInterf composto pela nova procedure showCancMsg
Tool	redefined process type Tool composto pelas redefined procedures initTI, calc, cancTool, endTool e endOutp.
Controller	process type Controller composto pelas procedures updTab, initSv, reqExecCtl e endExecCtl.
Server	process type Server composto pelas procedures exeToolSv e endToolSv

Tabela 4.4 - Mapeamento de OMT para SDL do sistema AIDA_CORBA Distribuído

A especificação SDL dá uma idéia clara do que foi alterado na especialização da versão centralizada para a versão distribuída, diferente do modelo OMT, onde só é possível identificar a inclusão de novas classes ou de novas operações nas classes existentes.

Na evolução do ambiente AIDA_CORBA centralizado para a versão distribuída, foram redefinidas, através da construção `REDEFINED` algumas operações e outras foram criadas, além da inclusão de novas classes. Na especificação da classe *Initial*, a procedure *initEnv* existente foi redefinida. Já a classe *AnalysisEnv* teve três procedures redefinidas (*initEI*, *reqExeTool*, *reqFormOut*, e *endSsAE*) e a inclusão de outras três (*reqExecAE*, *endExecAE*, *exeToolAE* e *treatAnsw*). Na classe *EInterf* foi incluída apenas uma nova procedure, indicando que todos os procedimentos existentes nesta classe permanecem inalterados. A classe *Tool* teve apenas a redefinição de procedures (*initTI*, *calc*, *cancTool*, *endTool* e *endOutp*), sem a inclusão de nenhuma nova operação. Também foram criadas as novas classes *Controller* e *Server* com suas respectivas procedures.

A versão AIDA_CORBA distribuída utiliza a construção *USE* para incluir o PACKAGE DistributedCORBA (Figura 4.2) na sua especificação, tornando disponíveis todas as definições da versão centralizada. Da mesma forma que na versão concorrente (Cap. 3), a versão AIDA_CORBA distribuída também foi definida como um package (Figura 4.11) permitindo o seu reuso futuro por outros sistemas.

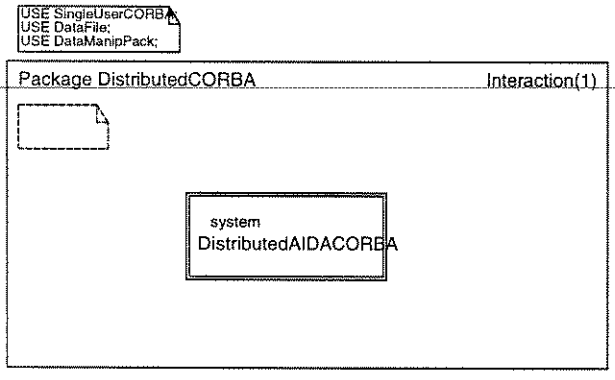


Figura 4.11 - Package DistributedCORBA

O system type *DistributedAIDACORBA* (Figura 4.12) consiste numa evolução da versão *SingleUserAIDACORBA* (Figura 4.3), com a inclusão de novos sinais para o controle dos diversos usuários que podem executar várias ferramentas ao mesmo tempo. Para que isto seja possível a versão distribuída herda a versão centralizada, através da construção *INHERITS*, redefinindo suas funcionalidades, indicado pela construção *REDEFINED*, devendo ser especificadas apenas as diferenças entre os dois sistemas. As instâncias *Mng* e *TExc* dos blocos *Manager* e *ToolExecution* respectivamente, definidas no sistema AIDA_CORBA centralizado foram reusadas, e os canais *MngUsr2* e *MngTExec2* foram

definidos para o transporte dos novos sinais. O bloco *FileConv* permanece como era na versão centralizada.

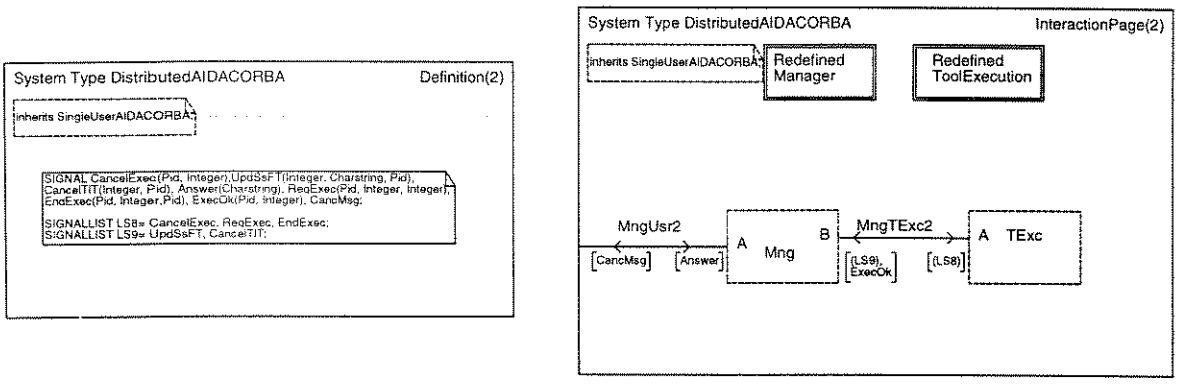


Figura 4.12 - System Type DistributedAIDACORBA

Com relação à versão concorrente (Figura 3.17), foram incluídos os novos sinais *ReqExec* e *ExecOk* no canal *MngTExc2*. Estes sinais foram definidos para permitir ao processo *AnalysisEnv* do bloco *Manager* a comunicação com o processo *Controller* do bloco *ToolExecution* para a requisição um servidor de processamento disponível para a execução de uma ferramenta de análise ou formatação de relatório (sinal *ReqExec*), ou para indicar a liberação de um servidor que se encontrava alocado (sinal *ExecOk*).

Na redefinição do block type *Manager* (Figura 4.13) foram definidas as novas instâncias *AE2* e *EI2* para os processos *AnalysisEnv* e *EInterf*, com a especificação de todos os sinais que são recebidos ou enviados por elas.

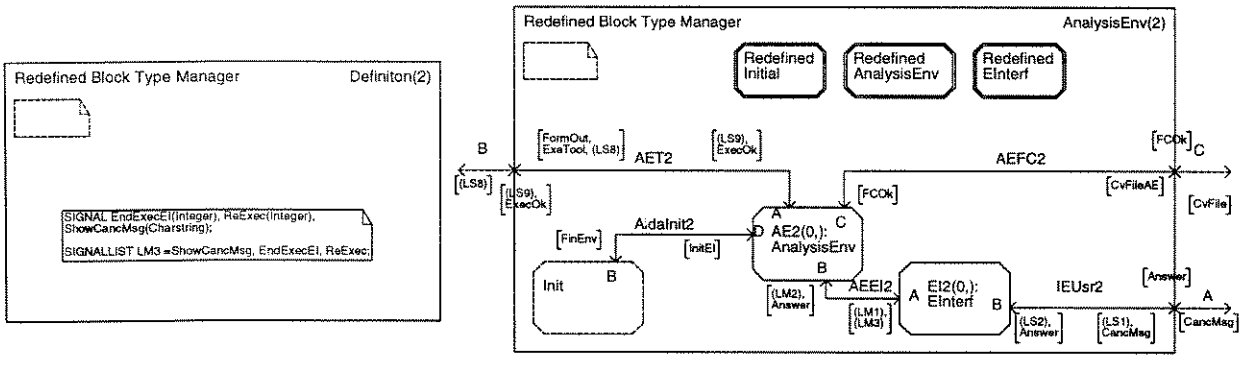


Figura 4.13 - Redefined Block Type Manager / DistributedAIDACORBA

Estes processos foram redefinidos para permitir o controle de várias requisições de execução de ferramentas de análise ou de formatação de relatórios feitas por um usuário. Já o processo *Initial* foi redefinido para permitir a execução do sistema por mais de um usuário simultaneamente, e a sua instância *Init* foi reusada.

O block type *ToolExecution* (Figura 4.14) foi redefinido, com base na sua versão centralizada (Figura 4.5), para incluir os novos processos *Controller* e *Server* (instâncias *Ctl* e *Sv* respectivamente). O processo *Controller* gerencia os servidores de processamento existentes, informando ao processo *AnalysisEnv* o identificador do servidor disponível. Já o processo *Server* é responsável por enviar as requisições de execução recebidas do gerenciador para o processo *Tool*, que executa a ferramenta ou formata o relatório. Diferente da versão concorrente anterior (Figura 3.19), onde apenas o processo *Controller* se comunicava com o bloco *AnalysisEnv* para a execução de uma ferramenta, o processo *AnalysisEnv* agora se comunica também com o processo *Server*, requisitando que ambos (*Controller* e *Server*) tenham ligação com o bloco *Manager*. Assim, cada um dos dois processos passa a apresentar uma rota de sinais para o *gate A*. Também foram definidas novas instâncias *T2*, *TI2* e *Calc2* do processos *Tool*, *TInterf* e *Calculation* para definir novos números inicial e máximo de instâncias criadas durante uma execução, permitindo a execução de mais de uma ferramenta ao mesmo tempo. O processo *Tool* teve alguns de seus procedimentos redefinidos para a criação das novas instâncias definidas.

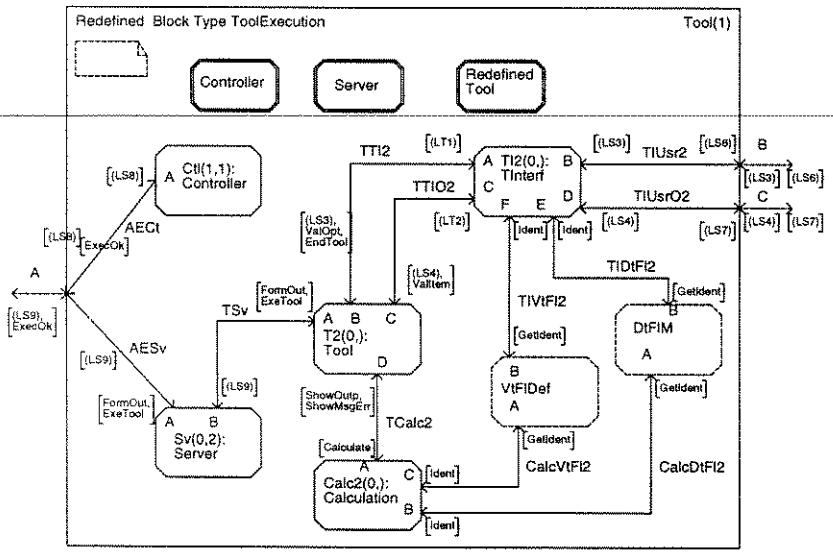


Figura 4.14 - Redefined Block Type ToolExecution / DistributedAIDACORBA

Na evolução do sistema SingleUserAIDACORBA para o DistributedAIDACORBA, a especificação dos processos redefinidos tornou-se mais simples, sendo necessário na maioria dos casos apenas a redefinição de alguns procedimentos ou a inclusão de outros.

- **Redefined Process Type AnalysisEnv (Figura 4.15) / Bloco Manager (Figura 4.13)**

O processo *AnalysisEnv* no sistema AIDA_CORBA distribuído, que foi redefinido baseado na versão centralizada da Figura 4.7, controla as diversas requisições de execução das ferramentas de análise e de formatação de relatórios emitidas por um usuário. Para emitir uma requisição de disponibilidade de servidor de processamento, foi definida a procedure *reqExecAE*, que envia o sinal *ReqExec* para o processo *Controller*, solicitando o identificador de um servidor e ativando o temporizador para um determinado tempo de espera da resposta.

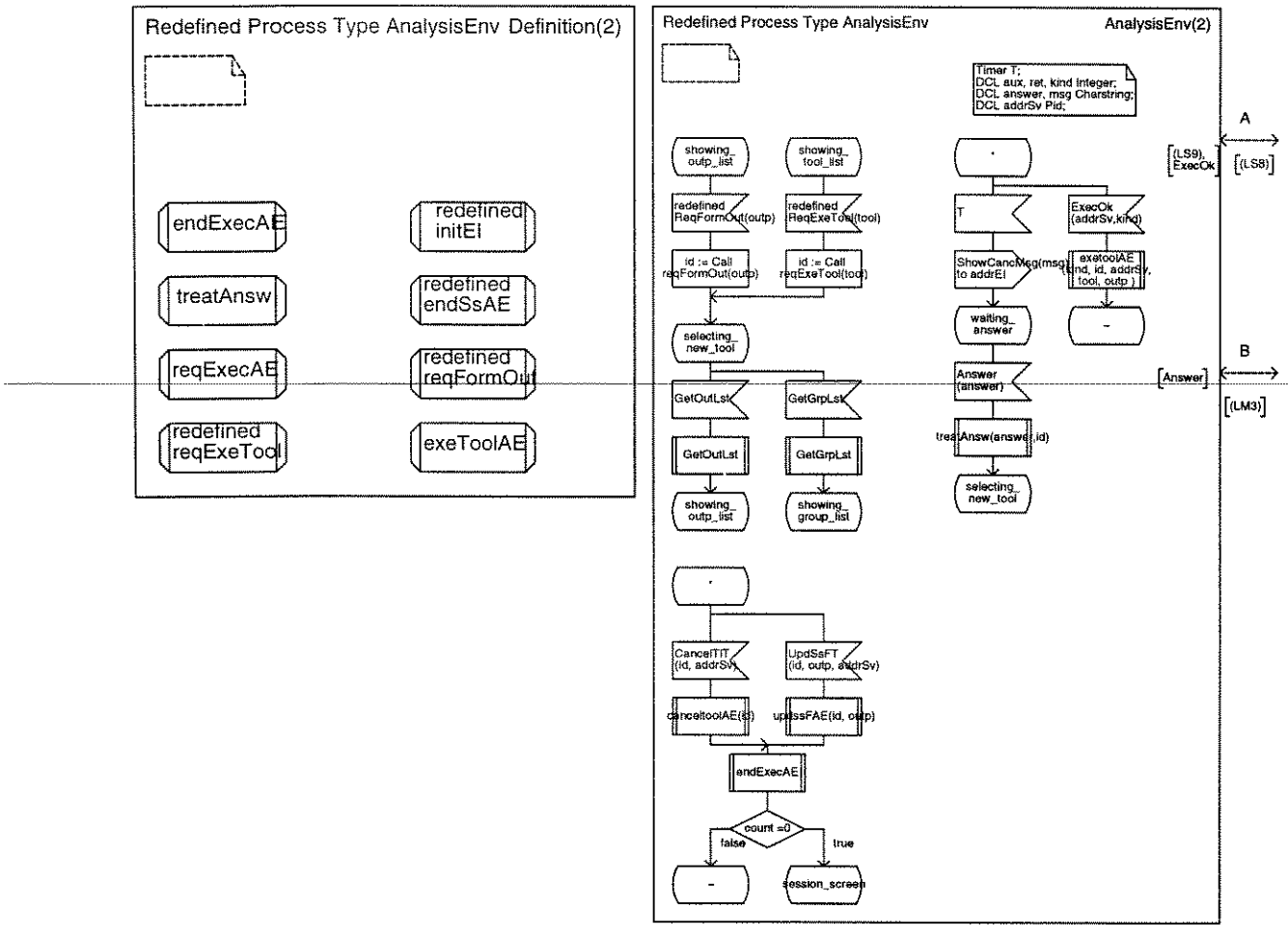


Figura 4.15 - Redefined Process Type AnalysisEnv / Bloco Manager

Toda vez que o usuário fizer uma requisição de execução (sinal *ReqExeTool ou ReqFormOut*), a procedure *reqExecAE* (Figura 4.16) é acionada, enviando o sinal *ReqExec(addr, id)* para o processo *Controller*, que indica a solicitação de um servidor para atender à requisição do usuário identificado por *addr*, cuja requisição está armazenada no

elemento de índice *id* da tabela de requisições do processo *AnalysisEnv*. É, então ativado um temporizador *T*, que define um tempo máximo de espera por uma resposta.

Na recepção da mensagem informando o servidor disponível, o processo aciona a redefined procedure *exeToolAE* (Figura 4.17), que envia a requisição (sinais *FormOut* e *ExeTool*) para o processo *Server* que dará início à execução da ferramenta de análise (*Tool*).

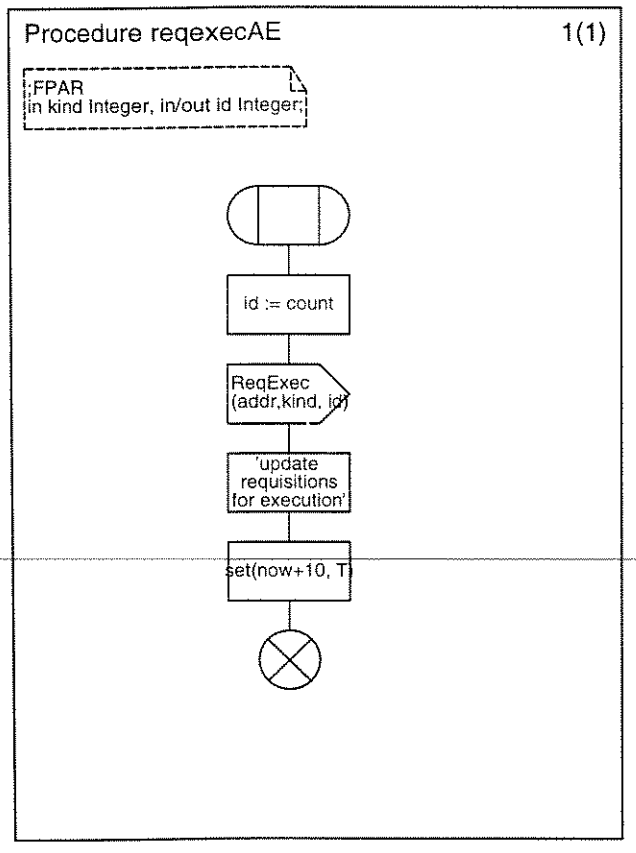


Figura 4.16 - Procedure reqExecAE

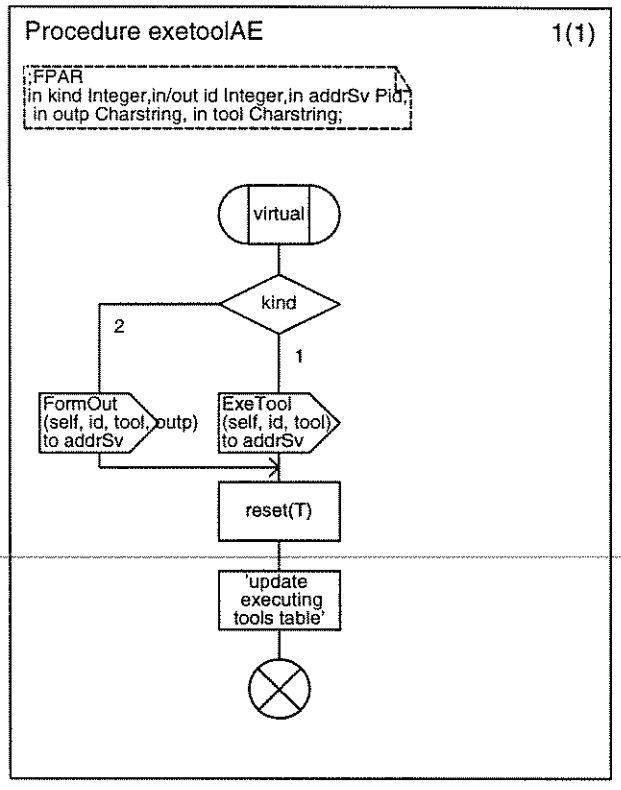


Figura 4.17 - Procedure exeToolAE

Na sua redefinição, o processo *AnalysisEnv* teve a procedure *initEI* (Figura 4.19) redefinida para possibilitar a criação da instância *EI2* do processo *EInterf*. Além disso, quando a execução da ferramenta termina (indicada no processo pela recepção dos sinais *CancelTIT* e *UpdSsFT*), a procedure *endExec* (Figura 4.18) é acionada, onde o sinal *EndExec* é enviado para o processo *Controller*, identificando o servidor liberado para novas execuções. A procedure *initEI* (Figura 4.19) foi redefinida para permitir a criação da instância EI2 do processo *EInterf* do bloco *Manager* (Figura 4.13).

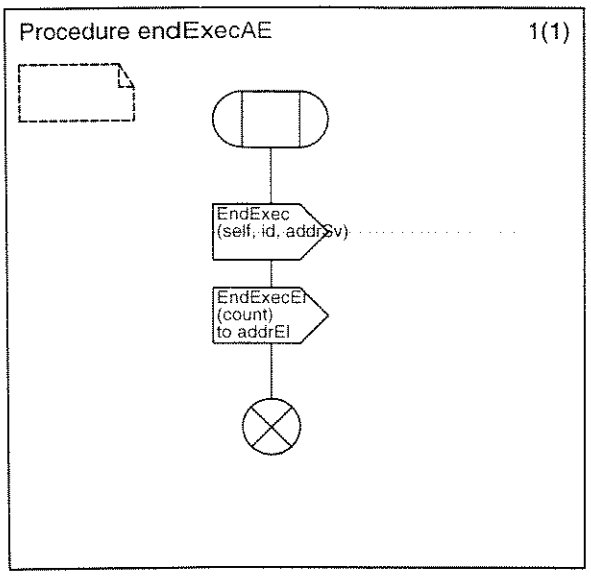


Figura 4.18 - Procedure endExecAE

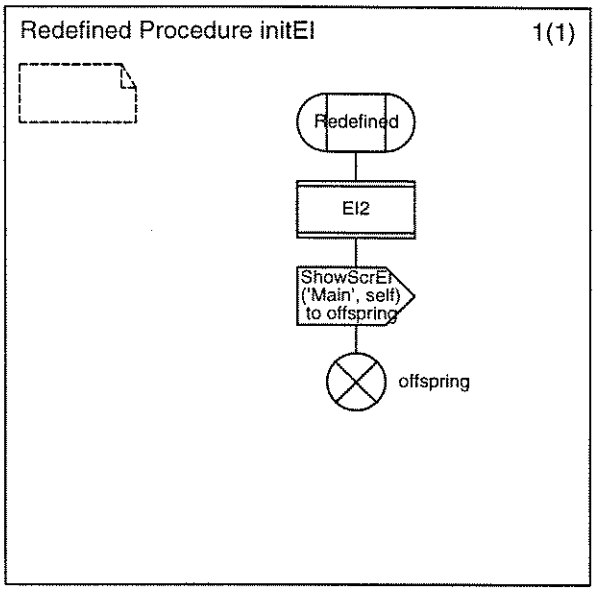


Figura 4.19 - Redefined Procedure initEl

- **Process Type Controller (Figura 4.20) / Bloco ToolExecution (Figura 4.14)**

O processo *Controller* além da inclusão de procedimentos, passou por uma revisão que o tornou genérico de maneira a permitir o controle de um número ilimitado de servidores de processamento. Na versão concorrente (Figura 3.23) este processo era específico para o controle de dois servidores apenas.

Para que fosse possível o controle dos servidores, e das ferramentas de análise que ele executa, foram definidas duas estruturas de dados neste processo:

- *Cell* (Tabela 4.5), uma estrutura que contém informações sobre um servidor, e que é representada em SDL através da construção `STRUCT`;

Nome	Tipo	Descrição
ident	Pid	Identifica o servidor na rede
nummax	Integer	Número máximo de ferramentas que podem ser executadas simultaneamente pelo servidor
currexec	Integer	Número de ferramentas sendo executadas simultaneamente pelo servidor

Tabela 4.5 - Estrutura Cell

- *Table*, um vetor onde cada elemento é um *Cell*, ou seja, cada elemento desta estrutura representa um dos servidores sob controle deste processo; a representação de vetores em SDL é através da construção `ARRAY`.

A declaração destas estruturas é feita na página de definições do processo *Controller* (Figura 4.20). A definição de um novo tipo de dados SDL se dá através da construção `NEWTYPE / ENDNEWTYPE`, usada neste processo para definir os novos tipos *cell* e *table*.

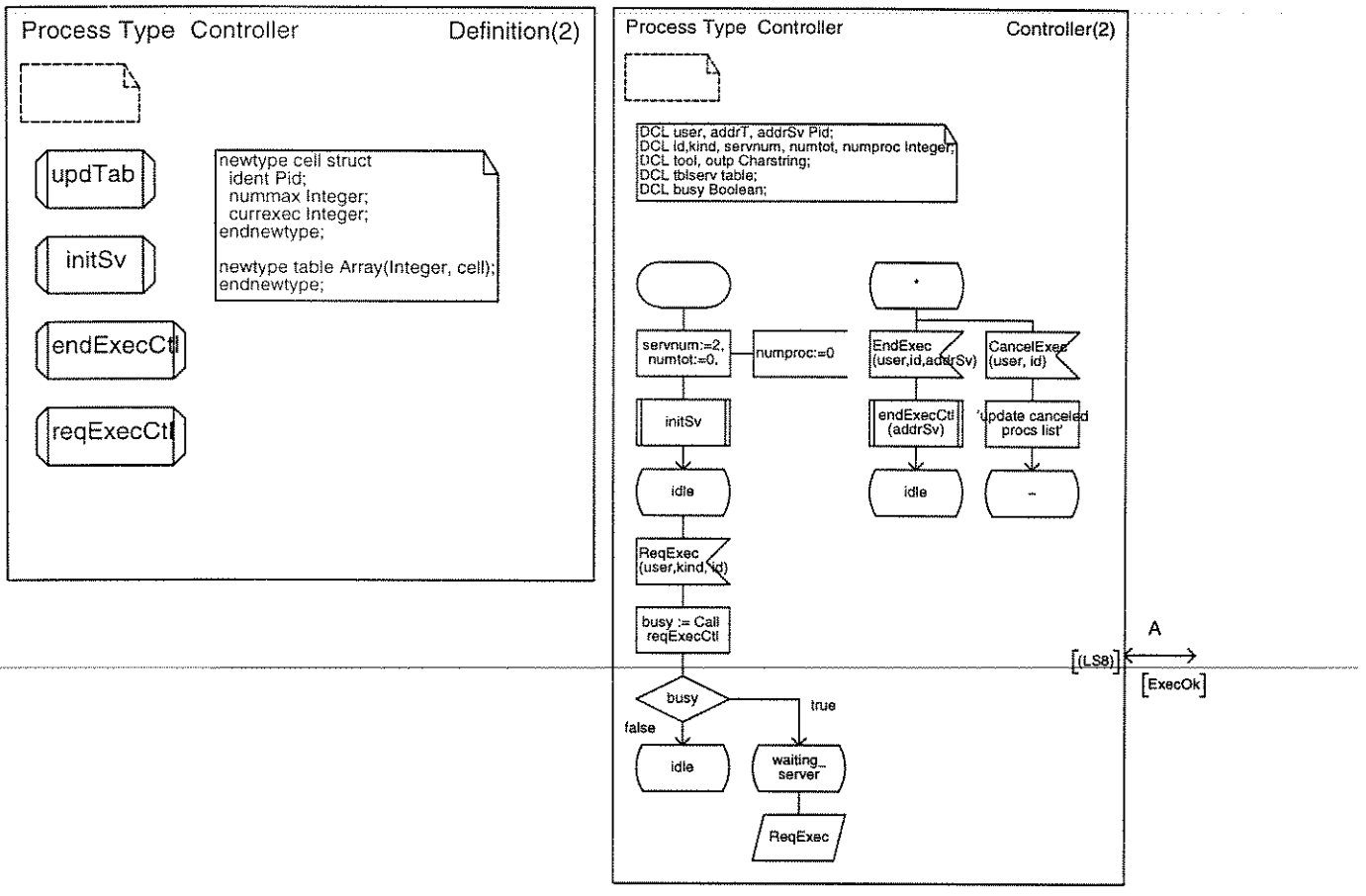


Figura 4.20 - Process Type Controller / Bloco ToolExecution

O tipo *cell* é definido como uma *struct* formada pelos três elementos definidos na Tabela 4.5: *ident* , *nummax* e *currexec*. Este tipo será usado na definição do tipo *table*, a estrutura utilizada para armazenar os diversos servidores sob controle do processo *Controller*. A definição de *table* é feita através da construção *array*, que apresenta a seguinte sintaxe:

xxx ARRAY(yyy,zzz)

onde:

xxx é o nome do tipo sendo definido;

yyy é o tipo do índice do array; e

zzz é o tipo para os elementos do array.

Assim, a definição da tabela de servidores do processo *Controller* se dá da seguinte forma: *table array(Integer,cell)*, definindo um vetor de nome *table*, cujos elementos são do tipo *cell* e os índices de acesso do tipo *Integer*. Como todo tipo de dados SDL, para que o novo tipo *table* seja utilizado pelo processo, deve ser definida uma variável deste tipo. Isto é feito na página de definição do processo, onde é definida através da construção *DCL* (declare) a variável *tblserv* que será usada para armazenar as informações de cada servidor sob controle do processo, durante a execução do sistema.

O processo *Controller* é composto por quatro procedures utilizadas para o controle dos servidores de processamento: *updTab*, *initSv*, *endExec* e *reqExec*.

A procedure *updTab* (Figura 4.21) é responsável pela atualização da tabela de servidores *tblserv*. Tem como parâmetros de entrada o tipo de atualização a ser feita (*kind*), o índice do elemento a ser atualizado (*ind*), o identificador do servidor (*servid*), o número máximo de ferramentas que podem ser executadas simultaneamente pelo servidor (nummax) e o número de ferramentas sendo executadas por ele (*currexec*). Este procedimento é utilizado para a inclusão de um novo servidor em *tblserv* (*kind=1*), onde são atualizados todos os elementos do servidor (Tabela 4.5), ou para atualizar o número de ferramentas sendo executadas pelo servidor (*kind=2*), onde é atualizado apenas o elemento *currexec* do servidor.

O acesso em SDL a um elemento de um *struct* se dá através da construção nome_struct!elemento_struct, e o acesso a um elemento de um *array* por nome_array(índice). Como a tabela de servidores *tblserv* é definida como um *array* de *struct*, o acesso aos seus elementos é feito da seguinte forma: nome_array(índice)!elemento_struct. Assim, *tblserv(ind)!ident:=servid* indica que o valor de *servid* será atribuído ao elemento *ident* da estrutura localizada no índice *ind* do vetor *tblserv*.

A procedure *initSv* (Figura 4.22) é responsável por iniciar cada servidor sob controle do processo *Controller*, utilizando o símbolo `CREATE REQUEST` para criar uma instância *Sv* do processo *Server*, e atualizando a tabela *servTb* com as suas informações do servidor iniciado, através da procedure *updTab*.

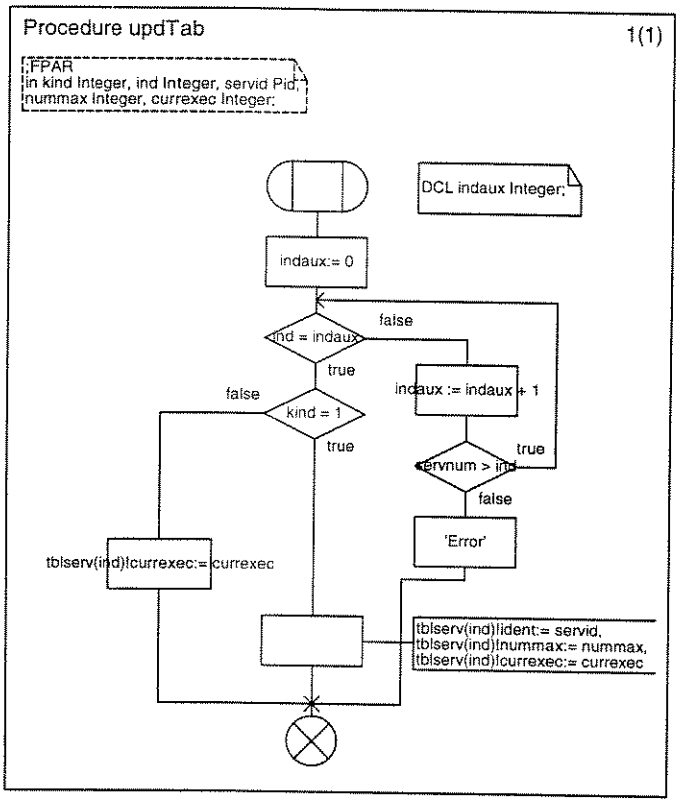


Figura 4.21 - Procedure UpdTb

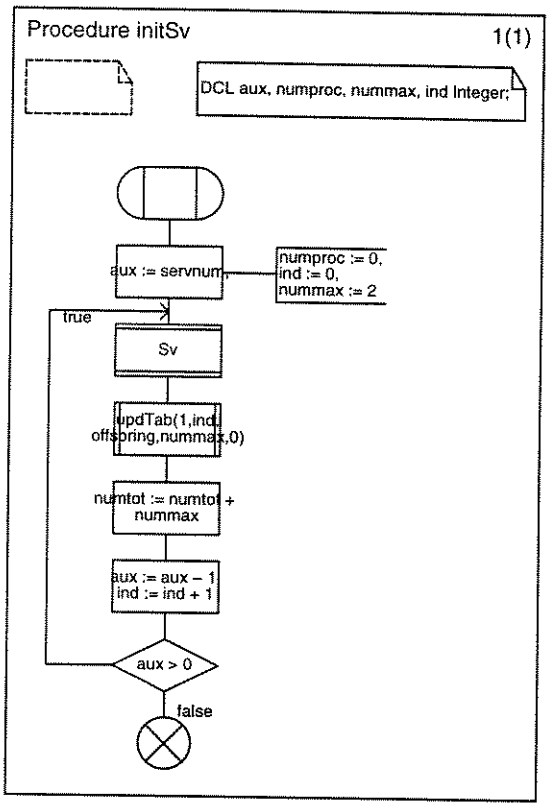


Figura 4.22 - Procedure initSv

A procedure *endExec* (Figura 4.23) é utilizada para tornar um servidor disponível para uso, após a recepção do sinal *EndExec* pelo processo *Controller* (Figura 4.20). Este procedimento procura na tabela *servTbl* o servidor identificado por *servid* para obter seu número atual de ferramentas em execução (*currexec).* Encontrando-o, decrementa *currexec* de um, atualizando este valor na tabela através da procedure *updTab.*

A procedure *reqExec* (Figura 4.24) é responsável por controlar a disponibilidade de servidores de processamento na consumação do sinal *ReqExec* pelo processo *Controller.* Inicialmente o procedimento verifica se a requisição recebida está na lista de requisições canceladas. Não tendo sido cancelada, o procedimento procura na tabela *tblserv* o primeiro servidor que tenha disponibilidade de execução. Quando este servidor é encontrado, o seu número de processos em execução (*currexec*) é incrementado de um, e este valor é atualizado na tabela, pelo acionamento da procedure *updTab.* Em seguida, este procedimento envia o sinal *ExecOk* para o processo *AnalysisEnv,* com o identificador do servidor que irá atender à sua requisição de execução. O valor *busy* a ser retornado, indica se ainda existem servidores disponíveis (*busy=false*) ou não (*busy=true*), que neste caso implica ao processo *Controller* (Figura 4.20) salvar na fila de sinais todas as requisições de execução que chegarem até que um servidor esteja disponível novamente (construção SAVE).

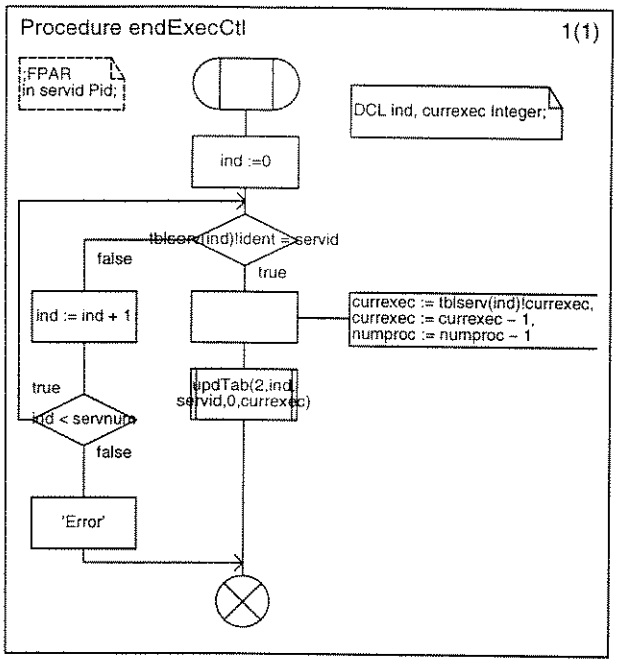


Figura 4.23 - Procedure endExecCtl

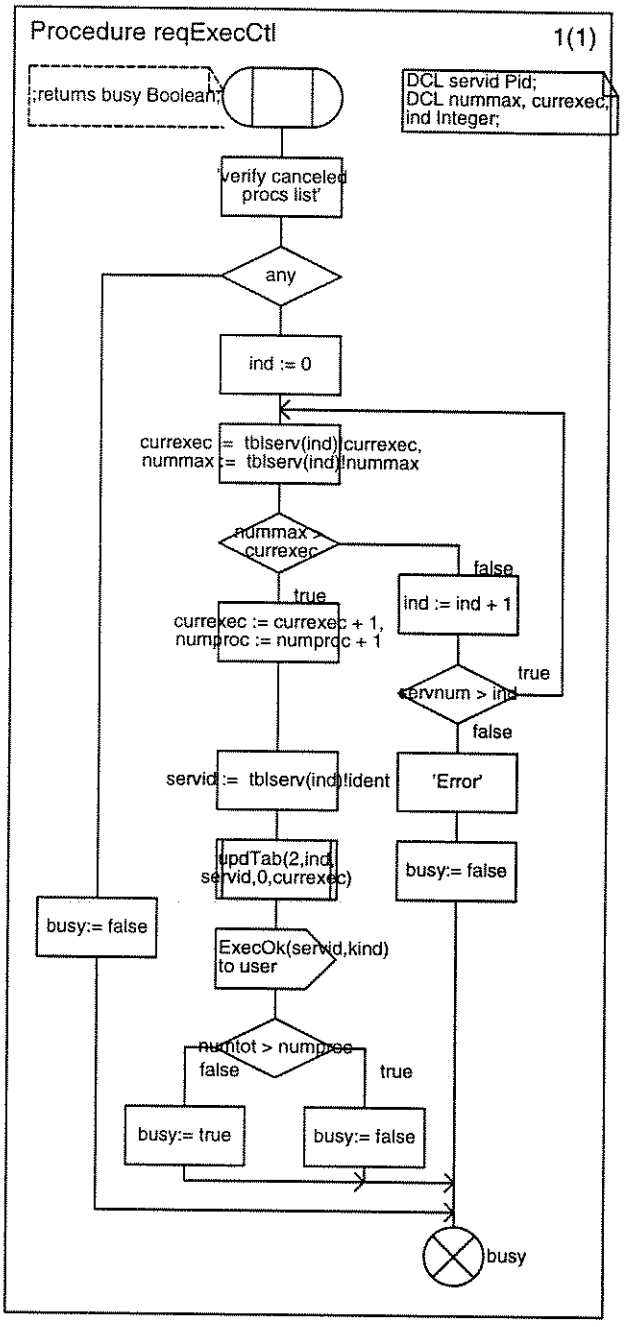


Figura 4.24 - Procedure reqExecCtl

- **Process Type Server (Figura 4.25) / Bloco ToolExecution (Figura 4.14)**

O processo *Server* possui duas procedures *exeToolSv* e *endToolSv* para a execução de uma ferramenta. A procedure *exeToolSv* é acionada quando o servidor recebe uma requisição de execução (sinais *ExeTool* e *FormOut*), criando uma instância *T* do processo *Tool*, que efetivamente executa a requisição. Já a procedure *endToolSv* é ativada na

recepção dos sinais que indicam o término da execução de uma ferramenta (sinais *UpdSsFT* e *CancelTIT*), repassando o sinal recebido para o processo *AnalysisEnv*.

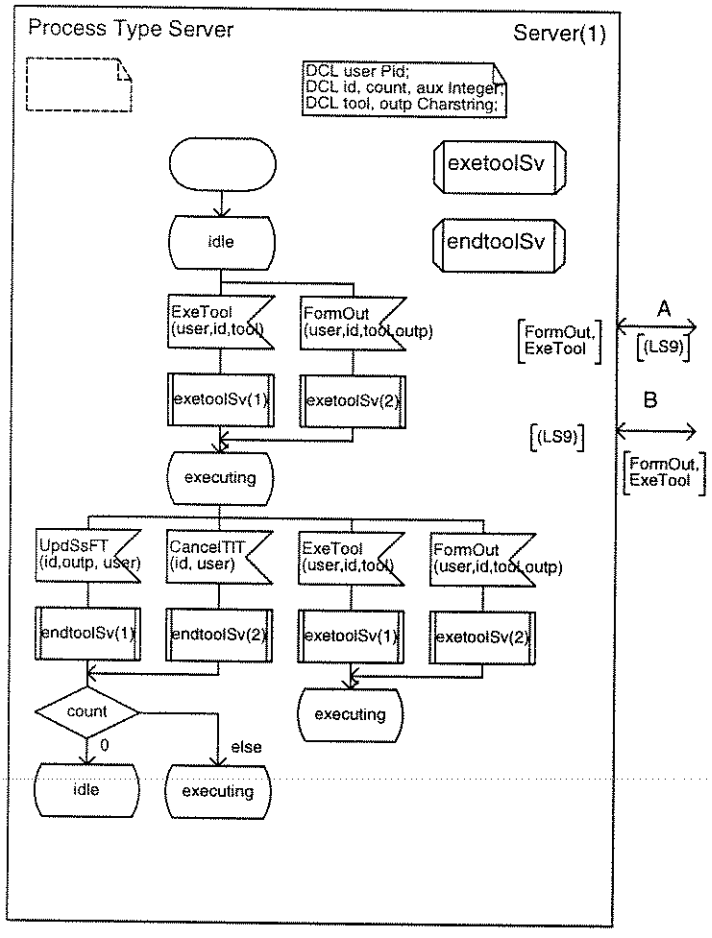


Figura 4.25 - Process Type Server

4.8. Tratamento de Arquivos virtuais de Dados

Um dos requisitos antigos dos usuários do ambiente AIDA [Embrapa, 1996] era a utilização de arquivos virtuais de dados, ou seja, a possibilidade de visualizar um arquivo de dados de uma maneira diferente, adequada às suas necessidades. Por exemplo, criar uma variável a mais num arquivo que corresponda à soma dos valores apresentados pelas suas demais variáveis.

As versões AIDA centralizada e concorrente (Cap.3) apresentavam visões gerais do sistema, e por esta razão, os modelos elaborados apenas citavam o tratamento de dados, sem fazer algum tipo de detalhamento das operações envolvidas. Nos modelos AIDA_CORBA (Figura 4.1 e Figura 4.10) foram incluídas as classes VirtFileDef,

DataFileManip, VirtFile e DataFile para especificar as classes que realizam o tratamento de dados.

A linguagem ASN.1 - Abstract Syntax Notation One, é uma linguagem definida pela ITU-T [iTU-T, 1994] para a descrição de tipos de dados e de valores, muito utilizada na especificação de dados em protocolos e serviços de telecomunicações, especialmente os que envolvem os níveis de aplicação. Muitos dos padrões da área de telecomunicações são baseados nesta linguagem. O seu uso é recomendado na linguagem SDL principalmente para a especificação de parâmetros de sinais de e para o ambiente, por possuir regras de codificação embutidas, as quais serão aplicadas sobre estes sinais. A definição de como usar a linguagem ASN.1 na especificação de um sistema SDL é apresentada em [ITU-T, 1995].

Os arquivos de dados não fazem parte do sistema AIDA, estando localizados em algum(ns) servidor(es) da rede com grande capacidade de armazenamento. Assim, como constituem entidades externas ao sistema, a linguagem ASN.1 foi utilizada para a especificação das estruturas de dados que representam estes arquivos. Como esta linguagem não permite a especificação de operações sobre os dados, foram definidos os processos *DataFileManip* e *VirtFileDef* que contém procedimentos em SDL que atuam sobre estas estruturas para a geração e acessos aos arquivos virtuais de dados.

O uso de arquivos virtuais no AIDA se dá em duas fases: a definição do arquivo em si, através da procedure *defVirt* (Figura 4.26) definida no processo *TInterf* do bloco *ToolExecution* (Figura 4.14) para a definição do arquivo virtual, e o acesso aos dados deste arquivo, através da procedure *calcC* (Figura 4.27), definida no processo *Calculation* do mesmo bloco *ToolExecution* para o cálculo dos algoritmos utilizando os dados lidos dos arquivos de dados selecionados.

A procedure *defVirt* (Figura 4.26) inicialmente obtém os arquivos existentes em um determinado diretório *dir* através da procedure *getFlist*, definida no processo *DataFileManip* do package *DataManipPack* (Figura 4.28).

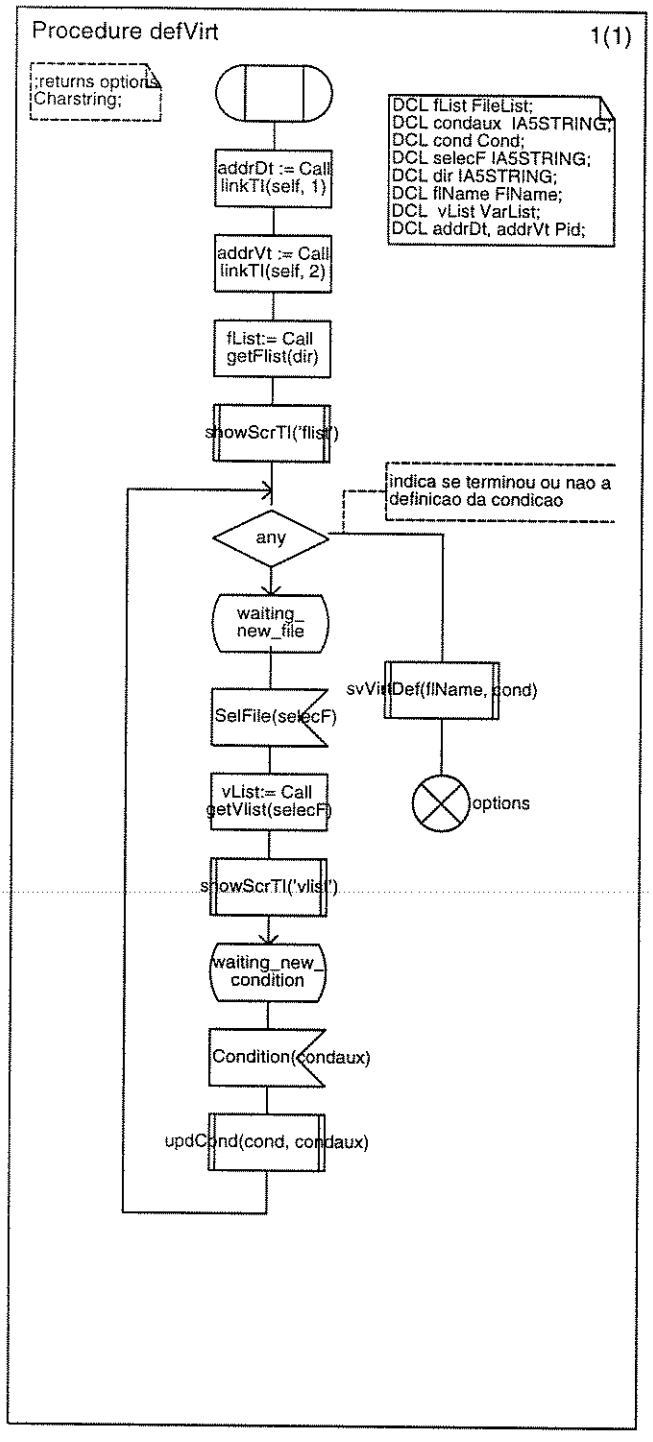


Figura 4.26 - Procedure defVirt / Processo TInterf / Bloco ToolExecution

A lista de arquivos é exibida na tela através da procedure *ShowScrTI*, para que o usuário possa selecionar os arquivos que lhe interessam. A medida que um arquivo é selecionado, indicado pelo sinal *SelFile(selecF)*, a procedure *getVlist* (processo *DataFileManip*) é acionada para informar as variáveis associadas a este arquivo, as quais também são exibidas para que o usuário possa especificar a condição *cond* de definição do arquivo virtual. A cada arquivo selecionado, esta condição é atualizada através da

procedure *updCond* (processo *VirtFileDef*). Ao final, a definição do arquivo virtual é armazenada em um arquivo de nome *flName*, através da procedure *svVirtDef* (processo *VirtFileDef*, também definido no package *DataManipPack*).

A procedure *calcC* (Figura 4.27), após obter o identificador do processo *VirtFileDef*, através da procedure *linkC*, inicia a leitura do arquivo de dados definido pelo usuário através da procedure *defVirt*.

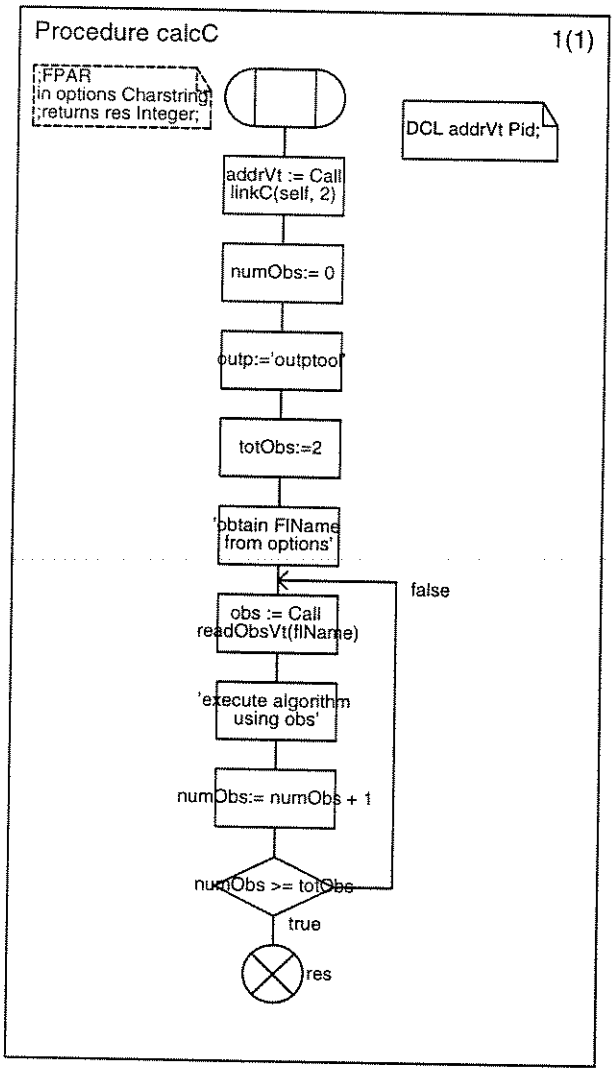


Figura 4.27 - Procedure calcC / Processo Calculation / Bloco ToolExecution

Isto é feito através da procedure *readObsVt*, (processo *VirtFileDef*), que lê uma observação do arquivo virtual por vez. A cada observação lida, é executado o cálculo sobre ela, com o procedimento continuando neste processo até que tenham sido lidas todas as observações do arquivo de dados definido.

Como os arquivos de dados e suas operações não fazem parte do ambiente AIDA, a especificação dos processos SDL para o gerenciamento destes arquivos foi feita num

package. O package *DataManipPack* (Figura 4.28) apresenta a definição de dois process types, onde cada um representa as operações definidas nas classes *VirtFileDef* e *DataFileManip* dos modelos OMT das versões AIDA_CORBA (Figura 4.1 e Figura 4.10). As definições deste package tornam-se disponíveis para os modelos SDL das versões centralizada e distribuída do AIDA_CORBA através da construção *USE* (Figura 4.2 e Figura 4.11 respectivamente).

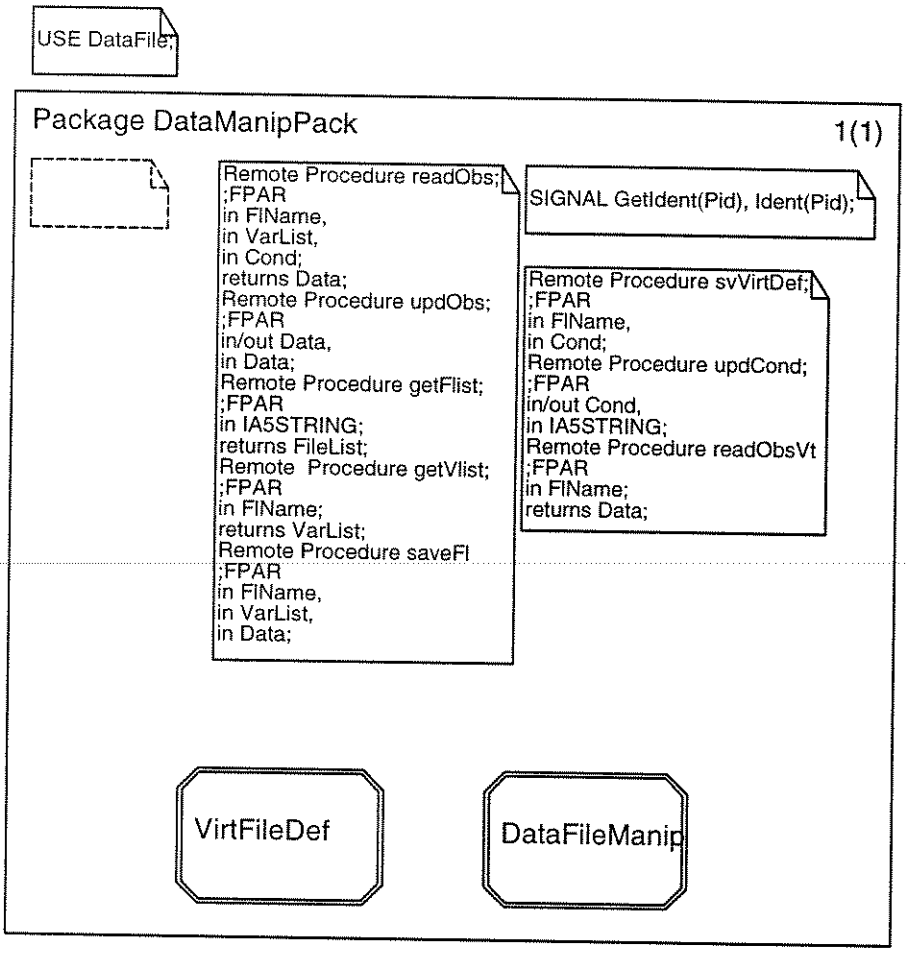


Figura 4.28 - Package DataManipPack

O package da Figura 4.28 contém também a definição dos sinais utilizados pelos processos e dos procedimentos que serão usados pelo AIDA_CORBA, através de RPC, um outro conceito disponível na linguagem SDL-92 e que é utilizado nesta versão. A RPC (Chamada Remota a Procedimento) permite a um processo utilizar um procedimento que está fora de seu escopo, fazendo uso de sinais implícitos, sem a necessidade de definição de canais ou sinais entre eles. Para que isto seja possível, o procedimento a ser invocado remotamente deve ser definido na sua especificação como uma *exported procedure*, como *imported* pelo procedimento que o usa e também como *remote* num nível visível aos dois

processos envolvidos, definindo-se os tipos de seus parâmetros. Os sinais implícitos utilizados nas RPC são definidos por *pCALL* e *pREPLY* para a invocação e resposta à invocação de cada umas das procedures. O processo requisitante envia o sinal *pCALL* à procedure remota e entra num estado de espera implícito, denominado *pWAIT_rpc.* Na recepção do sinal enviado, a procedure remota é executada e retorna ao processo requisitante o sinal *pREPLY*, com os valores de retorno, caso eles existam.

Os processos *VirtFileDef* e *DataFileManip* definidos no package *DataManipPack* atuam sobre estruturas de dados que correspondem aos atributos das classes *VirtFile* e *DataFile* nos modelos OMT da versão AIDA_CORBA (Figura 4.1 e Figura 4.10). Estas estruturas são usadas para representar a definição de um arquivo virtual de dados e um arquivo de dados existente, respectivamente. A definição destas estruturas no modelo SDL é feita em linguagem ASN.1, e é apresentada na Figura 4.29.

```
DataFile DEFINITIONS ::=
BEGIN
FlName ::= IA5STRING
FileList ::= SEQUENCE OF IA5STRING
VarList ::= SEQUENCE OF IA5STRING
Data ::= SEQUENCE OF IA5STRING
Cond ::= SEQUENCE OF IA5STRING
VirtFile ::= SEQUENCE {
        flName FlName,
        cond Cond }
DataFile ::= SEQUENCE {
        flName FlName,
        varList VarList,
        data Data }
END
```

Figura 4.29 - Definição dos atributos em ASN.1

São definidos os tipos de dados:

FlName, do tipo IA5STRING, usado para definir o nome do arquivo;

Cond, uma seqüência de elementos IA5STRING, usada para representar a condição para a geração do arquivo virtual, sendo composta por nomes de arquivos selecionados, suas variáveis e condições relacionadas;

VarList, uma seqüência de elementos IA5STRING, que define a lista de variáveis de um arquivo;

Data, uma seqüência de elementos IA5STRING, correspondendo aos dados do arquivo;

VirtFile, uma seqüência composta pelos tipos *FlName* e *Cond*, que contém o nome de um arquivo virtual de dados e a sua condição de definição;

DataFile, uma seqüência composta pelos tipos *FlName* , *VarList* e *Data*, representado um arquivo de dados que é definido por seu nome, suas variáveis e os seus dados em si.

Estes tipos de dados definidos em linguagem ASN.1 são utilizados na especificação dos processos *VirtFileDef* e *DataFileManip* através de operadores disponíveis em sorts (tipos abstratos de dados) da linguagem SDL. A recomendação Z.105 [ITU-T, 1995] apresenta a correspondência de alguns tipos de dados ASN.1 para sorts SDL. A Tabela 4.6 apresenta os tipos de dados utilizados no AIDA_CORBA.

ASN.1	Correspondente em SDL	Operadores disponíveis
IA5STRING	Charstring	todos existentes para Charstring
SEQUENCE OF	String	aqueles normalmente disponíveis para String
SEQUENCE	Struct	operadores para acesso, modificação e verificação do elemento da estrutura

Tabela 4.6 - Tradução ASN.1 - SDL

- **Process Type *DataFileManip*(Figura 4.30) / Package DataManipPack (Figura 4.28)**

O process type *DataFileManip* é formado pelos procedimentos para a definição e o acesso ao arquivo de dados virtual, que atuam sobre a estrutura *DataFile*, representada no processo pela variável *dtfl*. Os procedimentos são os seguintes:

getFlist - obtém lista de arquivos existente em um diretório;
getVlist - obtém lista de variáveis de um arquivo;
readObs - lê uma observação do arquivo de dados;
saveFl - salva um arquivo de dados;
updObs - atualiza uma observação lida.

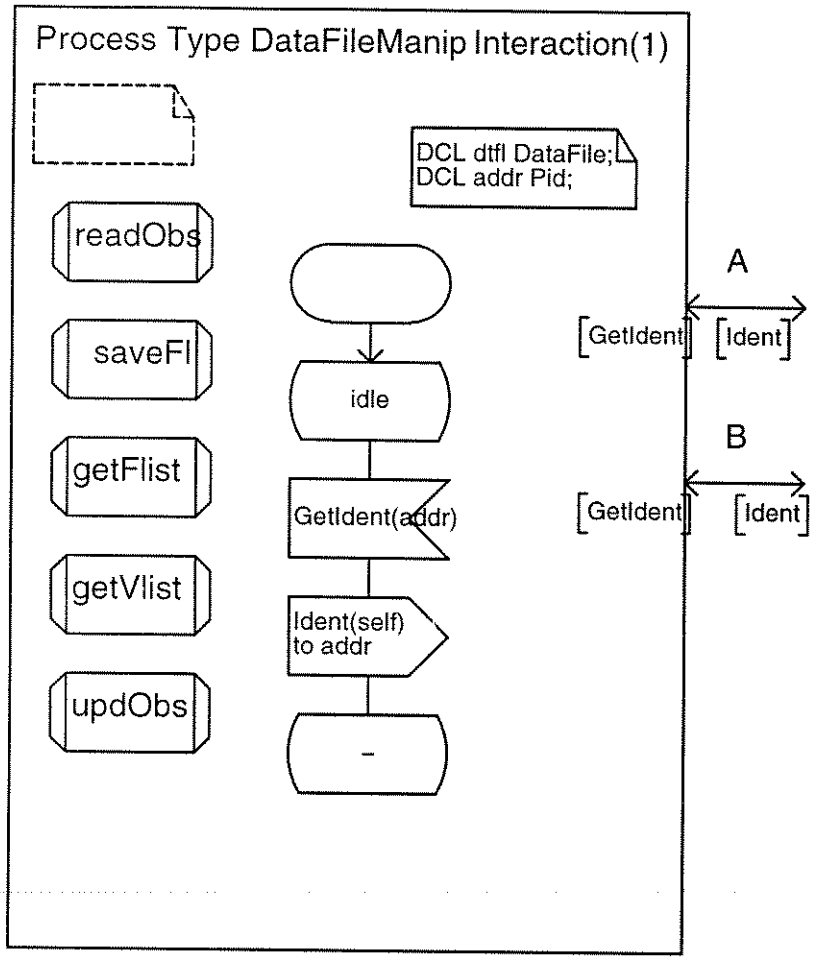


Figura 4.30 - Process Type DataFileManip

Este processo recebe apenas o sinal *GetIndent*, que representa uma requisição de um dos processos do ambiente AIDA_CORBA para obter o seu identificador (*addr*) e poder acessar remotamente os seus procedimentos, já que na invocação remota a procedures deve ser identificado o processo que contém a procedure invocada. Este identificador é retornado pelo sinal *Ident*.

A procedure *readObs* (Figura 4.31) lê uma observação de um arquivo de dados, tendo como parâmetros de entrada o nome do arquivo (*flname*), a lista de suas variáveis selecionadas (*selVarLst*) e a condição estabelecida para estas variáveis (*cond*). O procedimento obtém o número de variáveis selecionadas, usando para isso o operador *length*, e lê o valor de cada variável selecionada, atualizando a observação (*obs*) lida através da procedure *updObs* (Figura 4.34). Quando uma observação for totalmente lida, o procedimento verifica se ela atende à *cond*. A primeira observação que satisfizer a condição estabelecida é retornada.

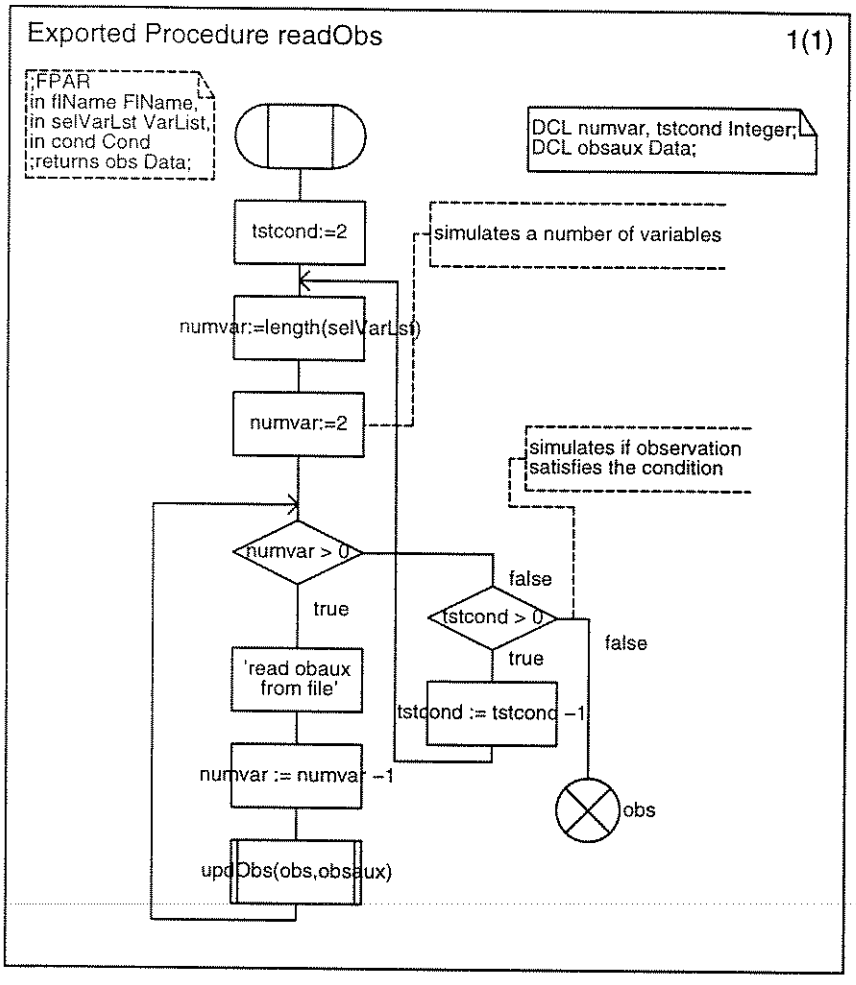


Figura 4.31 - procedure readObs / Processo DataFileManip

- **Process Type VirtFileDef (Figura 4.32) / Package DataManipPack (Figura 4.28)**

O process type *VirtFileDef* é formado pelos procedimentos para a definição e acesso ao arquivo de dados virtual, atuando sobre a estrutura *VirtFile,* representada no processo pela variável *virtfl* . São eles:

getCond - obtém a condição de definição do arquivo virtual;

getSelFl - obtém, a partir de cond, os arquivos selecionados para a definição do arquivo virtual;

getSelVar - obtém, a partir de cond, as variáveis selecionadas para a definição do arquivo virtual;

readObsVt - lê uma observação do arquivo virtual; utiliza as procedures *readObs* e *updObs* do processo DataFileManip;

svVirtDef - salva a definição de um arquivo virtual; usa a procedure *saveFl* do processo DataFileManip;

updCond - atualiza a condição de definição do arquivo virtual.

Este processo, da mesma forma que o processo *DataFileManip* (Figura 4.30), também recebe apenas o sinal *GetIndent* e retorna o sinal *Ident.* Além disso, como utiliza procedimentos remotos definidos no processo *DataFileManip*, deve definir estes procedimentos como *imported*, incluindo os tipos de dados dos seus parâmetros.

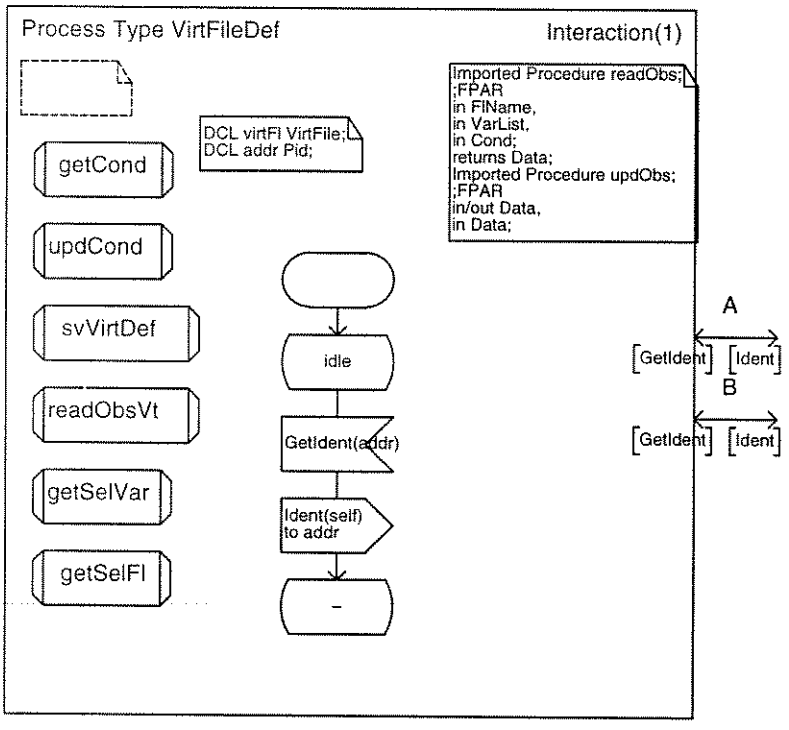


Figura 4.32 - Process Type VirtFileDef

A procedure *readObsVt* (Figura 4.33) especifica a leitura de uma observação do arquivo virtual de dados, tendo como entrada o nome do arquivo (*flName*) que contém a definição do arquivo virtual. O procedimento obtém a condição de definição deste arquivo (*cond*), através da procedure *getCond* e a partir de *cond* , obtém os arquivos de dados selecionados para definir o arquivo virtual (*selFl*). Passa então à leitura de cada arquivo selecionado, obtendo para isso sua lista de variáveis (procedure *getSelVar*). É então invocada a procedure *readObs* (processo *DataFileManip*), informando-se o nome do arquivo, suas variáveis e a condição estabelecida para estas variáveis, que tem como retorno uma observação deste arquivo que satisfaz a condição para a definição do arquivo virtual. A cada arquivo lido, a observação do arquivo virtual é atualizada com a observação retornada, através da procedure *updObs* (Figura 4.34). Quando todos os arquivos selecionados tiverem sido lidos, a observação (*obs*) resultante é retornada.

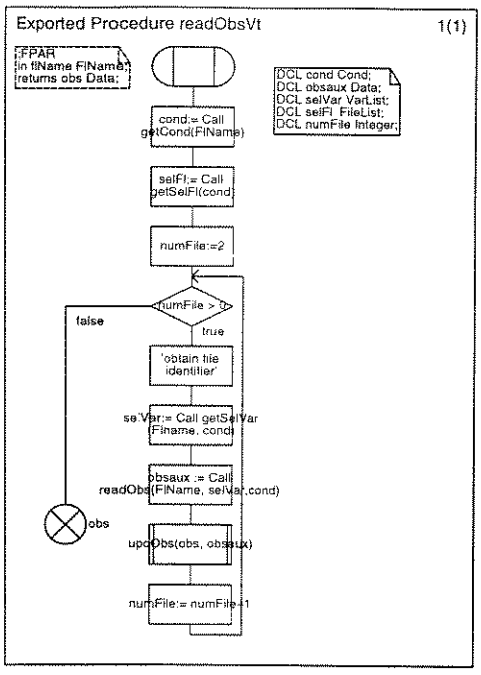


Figura 4.33 - procedure readObsVt

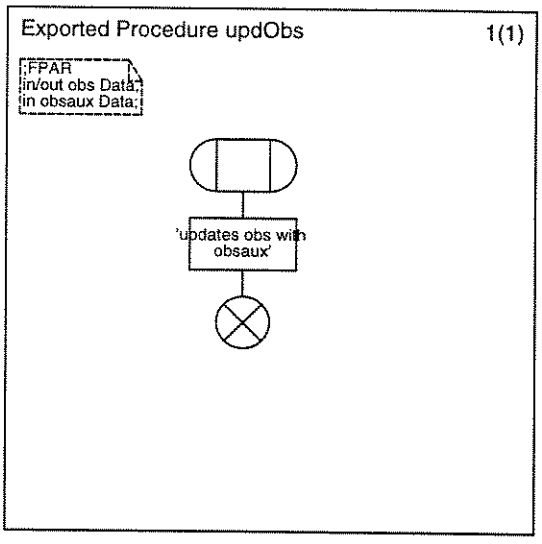


Figura 4.34 - procedure updObs

A procedure *svVirtDef* (Figura 4.35) é responsável por salvar a definição de um arquivo virtual de dados, tendo como entrada o nome do arquivo virtual (*flName*) e a sua condição de definição (*cond*). O procedimento atribui estes valores para os elementos correspondentes definidos na estrutura *VirtFile*, representada neste processo pela variável *virtFl*. Em seguida aciona a procedure *saveFl* do processo *DataFileManip*, que efetivamente salva a definição em um arquivo.

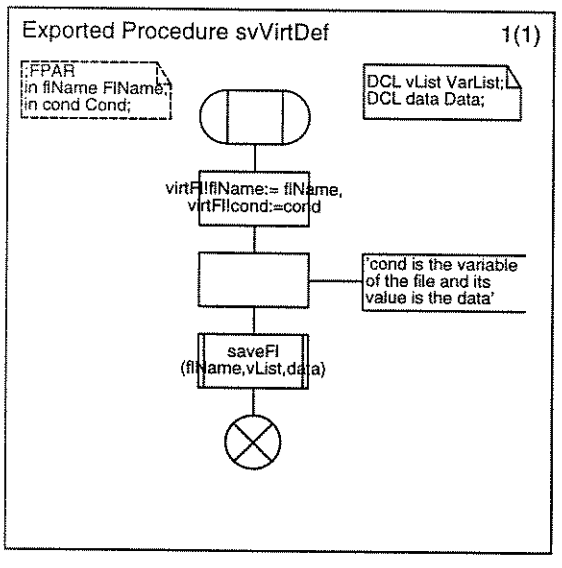


Figura 4.35 - procedure svVirtDef

4.9. *Especificação das interfaces do sistema AIDA_CORBA Distribuído em linguagem IDL*

Um sistema distribuído é formado por um conjunto de componentes que interagem entre si para satisfazer os requisitos do um sistema. Na arquitetura CORBA (Common Object Resquest Broker Architecture) esta interação deve ser especificada através de contratos entre os objetos do sistema que oferecem (servidor) e os que recebem (cliente) serviços. A linguagem IDL (Interface Definition Language) é usada para escrever estes contratos. Uma definição IDL inclui a definição de operações oferecidas pelos objetos (interfaces), bem como dos atributos e dos tipos usados por elas [Siegel, 1996].

Uma vez definida a especificação IDL, é necessária a geração de um código para que o Object Request Broker (ORB) gerencie a integração entre os clientes e a aplicação, e entre a aplicação e os servidores que ela tem de acessar, permitindo ao ORB traduzir a ordem dos bytes e o formato dos dados de uma requisição, ou sua resposta, sempre que necessário, e possibilitando a interoperabilidade entre sistemas Este código é gerado numa forma automática por um compilador ORB IDL específico para o ORB sendo utilizado, normalmente provido pela implementação da plataforma CORBA.

A operação oferecida pelo objeto pode ser pode ser *síncrona*, i.e. o objeto cliente fica bloqueado até que a execução do serviço requisitada é finalizada, ou *assíncrona*, onde o cliente envia a solicitação de serviço e continua a sua execução, e neste caso, o servidor não retorna uma informação ao cliente.

Na especificação da interface dos objetos, alguns itens devem ser levados em conta [TELELOGIC AB, 1996c][TELELOGIC AB, 1997].

1. especificação do nome e do tipo dos parâmetros de cada operação;
2. direção dos parâmetros, que podem ser:
    - in : somente de entrada;
    - out: somente de retorno;
    - inout: de entrada e de retorno simultaneamente;
3. atributos readonly, os atributos definidos em uma classe que não podem ser alterados por outros objetos;

4. especificação apenas das operações externas de cada objeto, ou seja, as operações locais aos objetos que não estarão transitando pela rede, não devem ser definidas.

Geralmente a especificação das interfaces IDL é feita a partir do modelo OMT existente. Na metodologia combinada OMT-SDL esta especificação pode ser elaborada tendo como base os dois modelos. Esta é uma operação quase automática, onde são utilizadas algumas regras de mapeamento entre as linguagens. As regras estabelecidas para o mapeamento OMT-SDL para IDL são as seguintes [TELELOGIC AB, 1996c][TELELOGIC AB, 1997][Loftus et. al., 1997][Carracedo et. al., 1997]:

block type: é mapeado como um módulo IDL;

process type: representa uma interface IDL; se o processo faz parte de um bloco SDL, a interface deve ser mapeada no módulo correspondente;

procedure: corresponde a uma operação síncrona (RPC ou não);

sinais: utilizados para a invocação e o retorno das operações síncronas, e para representar operações assíncronas, do tipo *oneway*;

parâmetros de procedures e sinais: são mapeados como parâmetros das operações IDL. Os parâmetros IDL podem ser do tipo: *in*, *inout* e *out*. Como o SDL só tem os dois primeiros, o do tipo *out* não é utilizado.

atributos de classe, declaradas como variáveis nos processos: são definidos como atributos da interface IDL.

newtype: representado por *typedef* em IDL;

tipos básicos (sorts): os tipos SDL são mapeados da seguinte forma:

SDL	**IDL**
integer	long, short, unsigned long ou unsigned short
real	double, float
boolean	boolean

tipos construídos: os tipos construídos SDL são mapeados da seguinte forma:

SDL	**IDL**
struct	struct
Charstring	string
array	sequence

A especificação de um sistema cliente-servidor em SDL apresenta:

- um package contendo as definições de tipos básicos, newtypes, sinais, procedimentos remotos e parâmetros associados definidos na interface IDL;
- um sistema SDL que contém a definição de blocos, processos, procedimentos e sinais de acordo com a especificação IDL;
- caso o sistema SDL também seja uma aplicação cliente/servidor é necessário gerar um package contendo as especificações IDL dos componentes do sistema que requerem os serviços. Esta interface contém as mesmas definições descritas para o package do servidor e ambos devem ser incluídos no sistema SDL através da cláusula USE.

A especificação de cada processo SDL cliente deve conter :

- uma requisição (output) às procedures que representam as operações especificadas na interface IDL, apresentando os mesmos argumentos que o procedimento ou o sinal de input do processo servidor;
- declarações dos sinais e gates para permitir a simulação em SDL das operações IDL.

e o processo servidor:

- uma variável para cada atributo da interface IDL;
- um procedimento para cada operação da interface IDL.

Para a especificação das interfaces IDL para o sistema AIDA_CORBA são feitas algumas considerações sobre o sistema:

- o iniciador do ambiente AIDA (classe Initial) se encontra em uma determinada máquina da rede; quando um usuário inicia a execução do sistema, são instanciados para ele o gerenciador do ambiente (classe AnalysisEnv) e a sua interface (classe EInterf), que é exibida utilizando algum mecanismo disponível na máquina deste usuário; o conversor de formatos (classes FileConv e FCInterf) se encontra nesta máquina também;
- a execução das ferramentas, ou a formatação de relatórios, (classe Tool) se dá em um dos servidores de processamento disponíveis (classe Server), que não necessariamente corresponde à máquina que abriga o iniciador e o

gerenciador do AIDA, nem à máquina do usuário; o controlador (classe Controller) pode, ou não, se encontrar em um dos servidores;

- existe uma, ou mais, máquina(s) com grande capacidade de armazenamento, para armazenar e gerenciar os arquivos de dados de cada usuário.

Com estas considerações e o mapeamento de OMT-SDL para IDL, tem-se a definição de três módulos IDL para o AIDA (Manager, ToolExecution e FileConvertion) e um módulo para o tratamento de arquivos de dados (DataPackManip).

O módulo *Manager* é composto por três interfaces: *Initial*, *AnalysisEnv* e *EInterf*, para o gerenciamento do sistema AIDA e das execuções de um usuário. Na definição das interfaces são especificadas apenas as operações que serão utilizadas por outras classes, e portanto, nem todas as operações do modelo OMT e procedures do modelo SDL são descritas aqui.

```
Module Manager {
        typedef sequence <string> lst;
        typedef long pid;

                interface Initial {
                oneway void initEnv(in string username);
                oneway void finEnv(in pid user);
        };
        interface AnalysisEnv {
                readonly attribute grpLst lst;
                readonly attribute toolLst lst;

                pid initEI();
                string initSs(new);
                oneway void cvFile(in string file);
                lst getOutLst();
                short reqExeTool(in string tool);
                short reqFormOut(in string outp);
                oneway void endSs();
                oneway void ext();
                oneway void treatAnswer(in string answer, in short id);
```

```
        oneway void reqExec(in short kind, in short id);
        oneway void endExec(in pid servid);
        };

    interface EInterf {
        typedef string scr;

        string initSs(in short new);
        lst selOut();
        lst selTool(in string group);
        lst selGrp();
        string void Group();
        string Tool();
        string Outp();
        oneway void endSs();
        oneway void ext();
        scr showScr(in string kind, in string information);
        scr showList(in string kind, in lst);
        string showCancMsg(in string msg);
        };
};
```

No módulo *Manager* foram definidos dois tipos de dados (*lst* e *pid*) a serem usados pelas suas interfaces. A classe *Initial* é quem faz a interação inicial e final do sistema AIDA com o usuário, e por esta razão, todas as suas operações são definidas em linguagem IDL. Na especificação da classe *AnalysisEnv*, devem ser descritas apenas as operações e atributos que são disponibilizados para a classe *EInterf* (*initSs*, *cvFile*, *getOutLst*, *reqExeTool*, *reqFormOut endSs*, ext e *treatAnswer* e as operações relacionadas aos seus atributos *grpLst* e *toolLst*, que a linguagem IDL automaticamente define), para a classe *Controller* (*reqExec*) e para a classe *Server* (*endExec*). As operações internas ao processo como por exemplo *initEI* e *updSsF* não são especificadas. A classe *EInterf* apresenta as operações invocadas pelo usuário (*initSs*, *selOut*, *selGrp*, *selTool*, *group*, *tool*, *outp*, *endSs* e ext) e pelo processo *AnalysisEnv*(*showScr*, *showList* e *showCancMsg*).

O módulo *ToolExecution* é composto por cinco interfaces: *Controller*, *Server*, *Tool*, *TInterf* e *Calculation* para a execução de ferramentas e formatação de relatórios.

```
Module ToolExecution {
        typedef string opts;
        typedef string params;
        typedef long pid;

        interface Controller {
                pid reqExec(in pid user, in short id, in short kind);
                oneway void endExec(in pid servid);
        };
        interface Server {
                oneway void exeTool(in pid user, in short id, in short kind, in tool, in
outp);
                oneway void endTool(in pid user, in short id, in short kind, in outp);
        };
        interface TInterf {
                typedef string scr;

                oneway void parms(in params);
                oneway void options(in opts);
                opts defVirt();
                opts defParm(in params);
                oneway void formItem(in string item);
                oneway void saveOutp();
                oneway void endOutp();
                oneway void restarTool();
                oneway void cancelTool();
                oneway void endTool();
                oneway void quitScr(in string kind);
                scr showScr(in string kind, in string information);
                string showMsgErr(in string err);
        };
        interface Tool{
                attribute params;

                Boolean valOpt(in opts);
```

```
            oneway void cancTool();
            oneway void endTool();
            boolean valItem(in string item);
            oneway void saveOutp();
            oneway void endOutp();
        };
        interface Calculation {
            long calc(in opts);
        };
};
```

Neste módulo foram definidos os tipos de dados *parms*, *options* e *Pid* globais às suas interfaces. A classe *Controller* define em IDL apenas as operações por controlar as requisições emitidas pela classe *AnalysisEnv* (*reqExec* e *endExec*). A classe *Server* apresenta a operação *exeTool* também acessada pela classe *AnalysisEnv*, e *endTool*, que é acessada pela ferramenta de análise ou pelo formatador de relatórios (classe *Tool*). A classe *TInterf* apresenta as operações invocadas pelo usuário (*options*, *parms*, *formItem*, *saveOutp*, *endOutp* *restartTool*, *canceTool* e *endTool*) e as operações invocadas pela classe *Tool* (*defVirt*, *defParm*, *showMsgErr*, *showScr*, e *quitScr*). Na especificação da classe *Tool*, são descritas as operações invocadas pelas classes *TInterf* (*valOpt*, *cancTool*, *endTool*, *valItem*, *saveOutp* e *endOutp*). A sua operação *calc* é interna e é invocada pela própria ferramenta. A classe *Calculation* apresenta apenas a operação *calc* que é invocada pela ferramenta de análise (classe *Tool*).

O módulo *FileConvertion* apresenta duas interfaces: *FileConv* e *FCInterf*.

```
Module FileConvertion {
    typedef string cvFl;
    typedef string file;

    interface FileConv {
        cvFl cvFile(in file);
    };
    interface FCInterf {
        cvFl showCvFl(in cvFl);
    };
};
```

Neste módulo foi definido o tipo de dado *cvFile*, que representa o arquivo externo convertido. A classe *FileConv* define em IDL a operação *cvFile* para converter arquivos externos, requisitada pela classe *AnalysisEnv*. A classe *FCInterf* apresenta a operação *showCvFl* que é acionada pela classe *FileConv*.

O módulo *DataManipPack* é composto por duas interfaces: *DataFileManip* e *VirtFileDef* para o tratamento de arquivos de dados, virtuais ou não.

```
Module DataManipPack {
        typedef string lst;
        typedef string obs;
        typedef string flName;
        typedef string cond;
        typedef sequence <string> data;

        interface DataFileManip {
                lst getFlist(in string dir);
                lst getVlist(in flName);
                obs readObs(in flName, in lst varLst, in cond);
                oneway void saveFl(in flName, in lst varLst, in data);
                oneway void updObs(inout obs, in obsaux obs );
        };

        interface VirtFileManip{
                cond getCond(in flName);
                fileLst getSelFl(in cond);
                varLst getSelVar(in flName, in cond);
                obs readObsVt(in flName);
                oneway void svVirtDef(in flName, in cond);
                oneway void updCond(inout cond, in condaux cond);
        };
};
```

No módulo *DataManipPack* foram definidos os tipos de dados *Lst*, *obs*, *flName*, *cond* e *Data* globais às suas interfaces. A classe *DataFileManip* apresenta operações que serão acessadas pelas classes *TInterf* (*getFlist*, *getVlist*) e *Calculation* (*saveFl*) do AIDA e pela classe *VirtFileDef* (*readObs*, *updObs* e *saveFl*). A classe *VirtFileDef* apresenta operações

que serão acessadas pelas classes *TInterf* (*updCond* e *svVirtDef*) e *Calculation* (*getCond*, *getSelFl* e *readObsVt*).

Com a definição das interfaces do AIDA_CORBA distribuído em linguagem IDL e a sua especificação completa, o sistema poderia ser validado com a sua execução em ambiente CORBA [TELELOGIC AB, 1996c]. Entretanto, é necessária a existência de uma ferramenta apropriada que faça a conversão desta especificação SDL, utilizando a sua interface IDL, para uma aplicação executável em ambiente CORBA. Tal ferramenta não existe na versão do SDT disponível no DT. Por esta razão, a seção seguinte apresenta a validação tradicional do sistema resultante.

4.10. Validação do sistema AIDA_CORBA Distribuído

Os resultados da validação pelo SDT Validator [TELELOGIC AB, 1996c] do sistema AIDA_CORBA distribuído são apresentados na Figura 4.36.

```
Random-walk

** Starting random walk **
Depth      : 300
Repetitions : 150

** Random walk statistics **
No of reports: 0
Gen states   : 84355
Max depth    : 300
Min depth    : 300
```

Figura 4.36 - Resultados da Validação do AIDA_CORBA Distribuído

Na validação deste sistema, a árvore de comportamento criada possui uma profundidade máxima (*depth*) de 300, e foram realizadas 150 escolhas (*repetitions*) de caminhos para teste. Segundo os resultados apresentados, não foi encontrada nenhuma situação de erro (*No. of reports*) na validação dos 84.355 estados gerados (*gen. states).*

A Figura 4.37 mostra a janela principal do *Coverage Viewer* para a validação final do AIDA_CORBA distribuído, apresentado a cobertura dos símbolos do sistema, e onde são apresentados os seus nós mais executados. Os símbolos associados ao sistema *DistributedAIDACORBA* foram executados 110.109 vezes, e o menos executado deles não foi executado nenhuma vez e os mais executados, 2.098 vezes, sendo estes símbolos

foi executado nenhuma vez e os mais executados, 2.098 vezes, sendo estes símbolos definidos no processo Initial. A raiz da árvore, denominada *total*, apresenta a cobertura do sistema como um todo, onde são considerados, além do sistema *DistributedAIDACORBA*, os packages *DataManipPack* e *SingleUserAIDACORBA*, sendo estes últimos usados na definição do sistema através da cláusula USE.

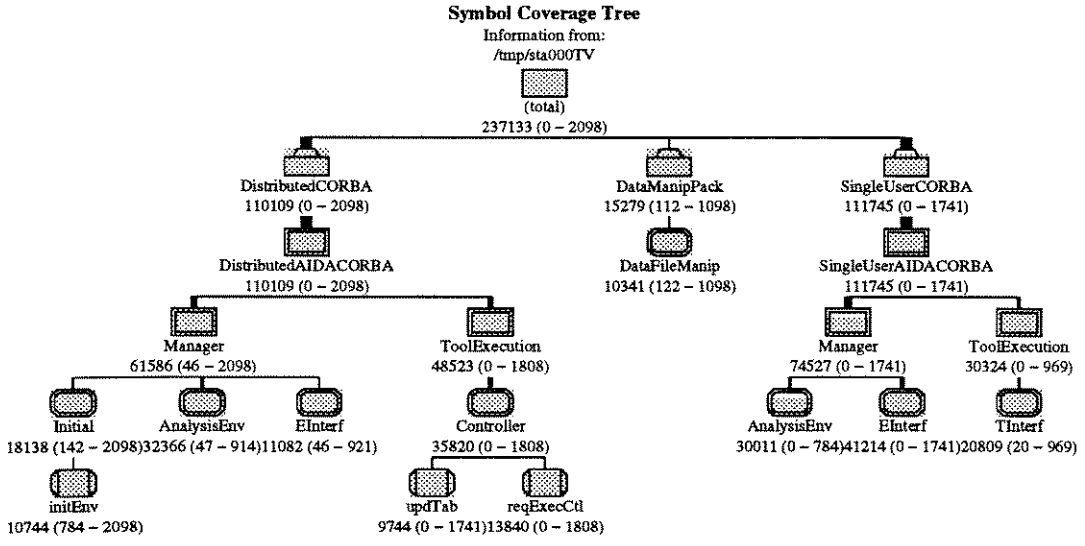


Figura 4.37 - Coverage Viewer - AIDA_CORBA Distribuído

A janela de cobertura detalhada do *DistributedAIDACORBA* é apresentada na Figura 4.38, indicando que os símbolos associados a este nó foram executados 110.109 vezes, e que, 230 dos 234 símbolos definidos no sistema *DistributedAIDACORBA* foram testados, ou seja, 4 deles deixaram de ser testados. Através da janela detalhada foi possível verificar que estes símbolos estão relacionados a situações de erros no processo *Controller*, quando um servidor não é encontrado na tabela de servidores. A princípio estes casos de erros nunca vão acontecer realmente, pois o processo só busca na tabela de servidores por servidores que ele controla. Assim, durante a validação do sistema este erro não aconteceu e por esta razão os símbolos não foram testados.

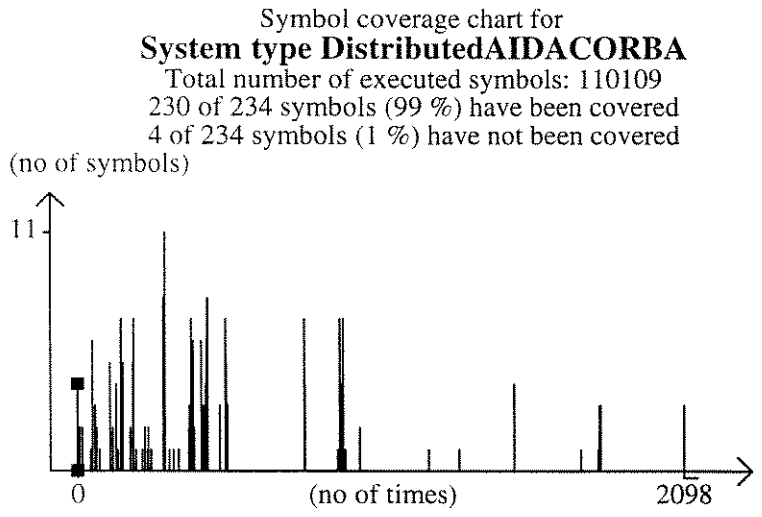


Figura 4.38 - Coverage Viewer Detalhado - AIDA_CORBA Distribuído

4.11. Exemplo de Simulação do AIDA_CORBA Distribuído

A execução do sistema AIDA_CORBA distribuído foi feita para verificar o comportamento do sistema especificado usando-se procedimentos, bem como o uso de chamadas remotas a procedimentos.

O diagrama da Figura 4.39 apresenta parte do diagrama MSC gerado para o seguinte cenário: O usuário 1 faz uma requisição de execução de ferramenta de análise, a qual é enviada ao controlador para verificar a disponibilidade de servidores. Como não existe nenhuma ferramenta sendo executada, o controlador retorna ao gerenciador do ambiente de análise instanciado para este usuário, o identificador do servidor que irá atender à sua requisição. Esta requisição é repassada ao servidor correspondente que inicia a execução da ferramenta em si. O usuário deve fornecer a definição do arquivo virtual sobre o qual será feita a análise.

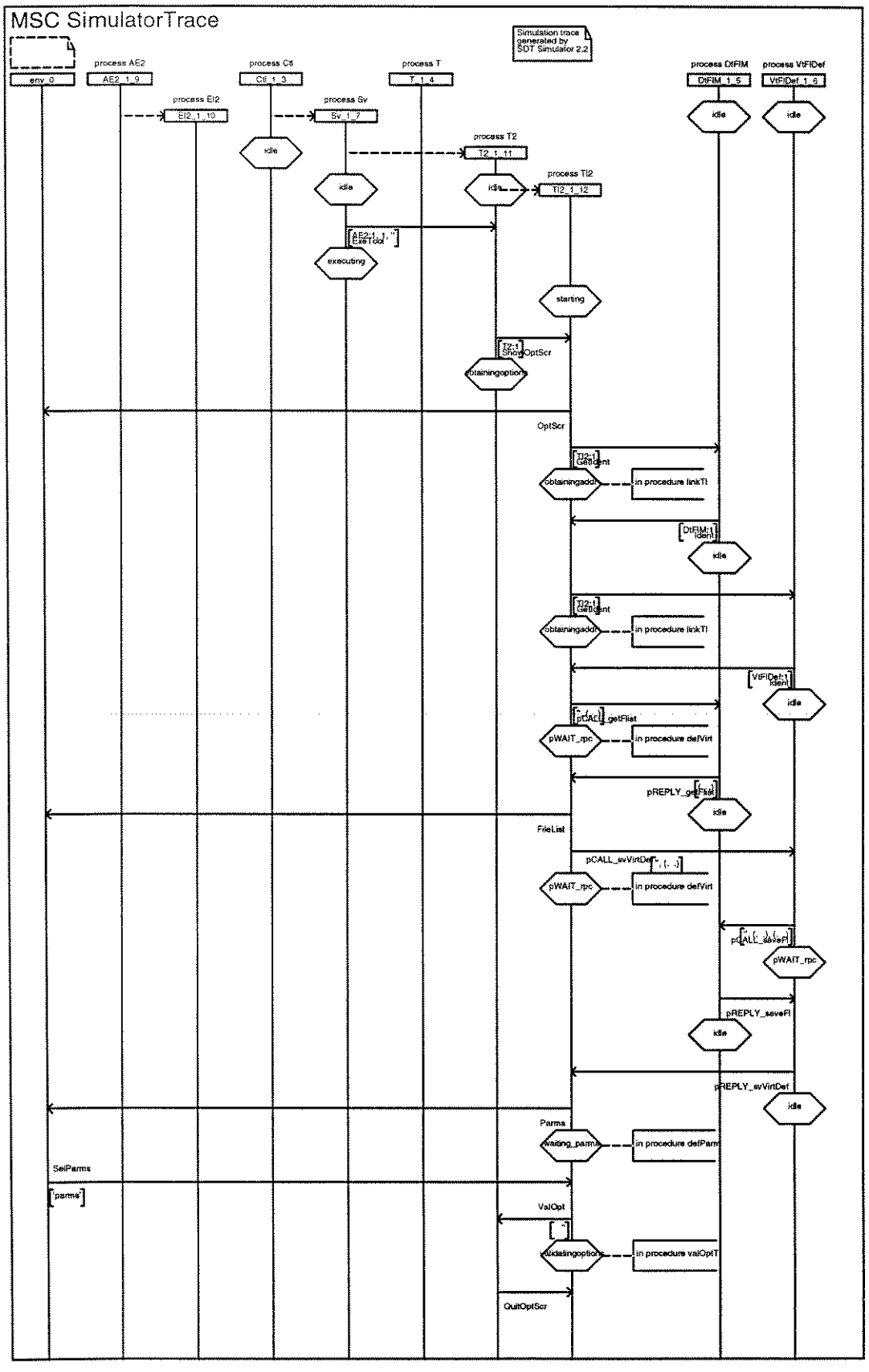


Figura 4.39 - Exemplo de simulação do sistema AIDA_CORBA Distribuído

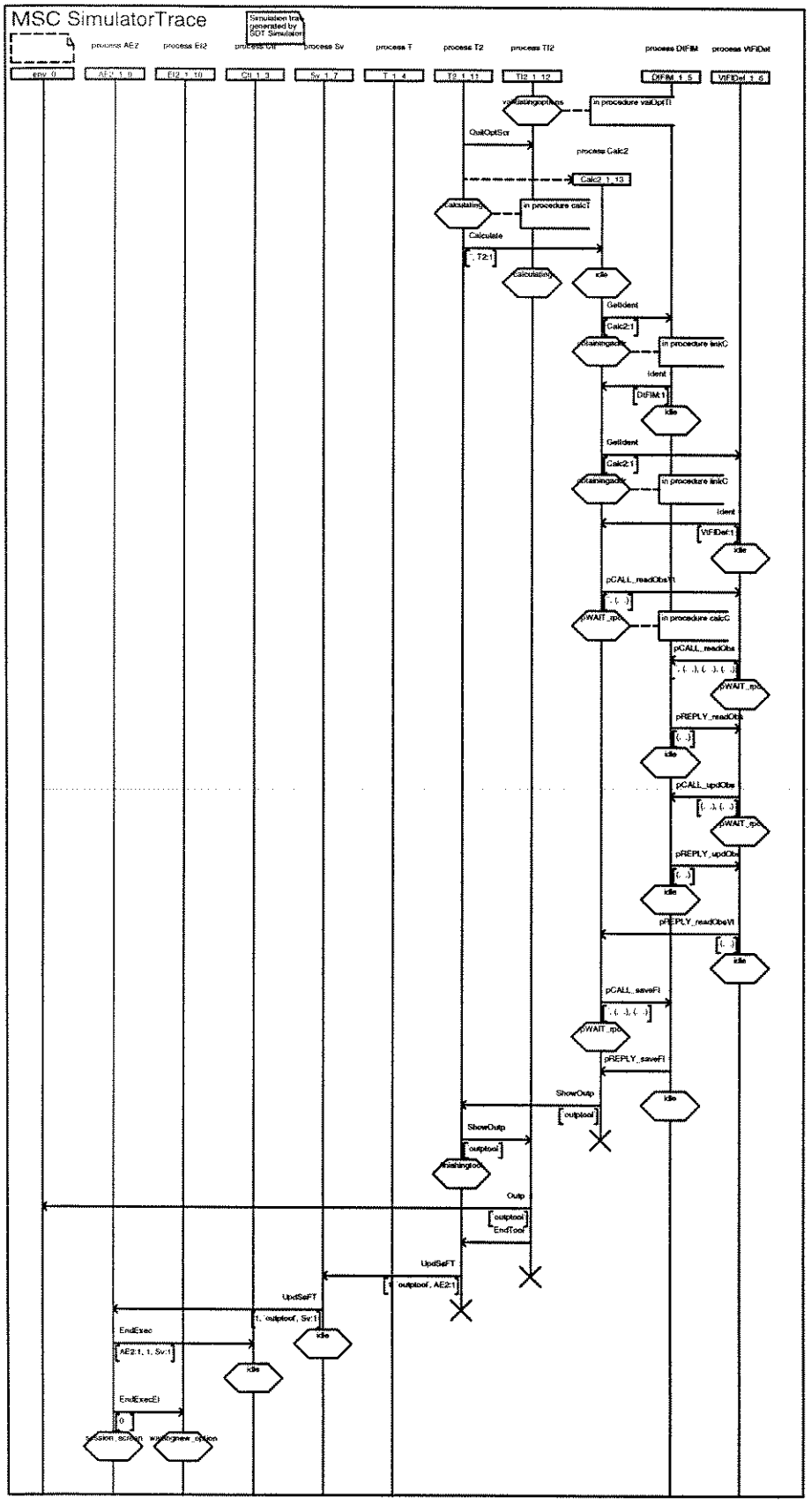


Figura 4.39 - Exemplo de simulação do sistema AIDA_CORBA Distribuído (continuação)

Uma vez informados o arquivo a ser utilizado, bem como os parâmetros de execução da ferramenta, tem início o cálculo do algoritmo de análise, que realiza a leitura aos arquivos definidos. Ao final do cálculo, é gerado um arquivo de dados de saída, e é exibido o relatório com os resultados da execução. Ao final da execução da ferramenta, o controlador recebe a informação de que o respectivo servidor se encontra disponível para uma nova execução.

O diagrama MSC apresentado corresponde ao ponto em que tem início a execução da ferramenta. Neste sentido, a instância *T2_1* da ferramenta cria a instância *TI2_1* da sua interface. *TI2_1* estabelece uma ligação com os processos *DataFileManip* e *VirtDataFile* (instância *DF* e *VF* respectivamente). *TI2_1* faz requisições remota a procedimentos do processo *DataFileManip e VirtDataFile*. A chamada remota a procedimentos é feita pelo sinal implícito *pCALL_<nome_do_procedimento>*, e o seu retorno através do sinal *pREPLY_<nome_do_procedimento>*. A instância requisitante entra no estado de espera implícito *pWAIT_rpc*. *TI2_1* obtém a lista de arquivos disponíveis no diretório *dir*, através do procedimento *getFList,* e exibe a lista de arquivos. O usuário seleciona um arquivo e *TI* obtém a lista de variáveis deste arquivo, através da procedure *getVList*, que serão exibidas para o usuário. Ele indica a condição para a definição do arquivo virtual. Esta definição é salva em um arquivo através da procedure *saveFI*. O usuário é solicitado para definir os parâmetros de execução da ferramenta, e após isso, as opções de execução são repassadas para *T2_1* para serem validadas. Não havendo erros, o processo *Calc* é instanciado, que inicia a leitura dos arquivos de dados, e executa, para isso, a procedure *readObsVt* que lê um arquivo virtual, usando a procedure *readObs* e *updObs* do processo *DataFileManip*. A cada observação virtual lida é executado o cálculo do algoritmo, e ao final o processo gera um arquivo de dados com os resultados obtidos, usando a procedimento *saveFI*. O relatório gerado é retornado para o processo *T2_1* para que ele seja exibido na tela por *TI2_1*. A execução da ferramenta é finalizada, e o server envia o sinal *UpdSsFT* para *AE2_1*, que envia o sinal *EndExec* para *Controller*, indicando o servidor que está liberado para atender a uma nova requisição.

5. Conclusão

5.1. Introdução

Este capítulo apresenta as considerações gerais desta tese, comentando as suas contribuições e os resultados obtidos. Ao seu final são feitas sugestões para eventuais especializações decorrentes deste trabalho.

5.2. Considerações Gerais

O uso da Object Modeling Technique - OMT tem se mostrado como uma excelente opção na engenharia de software para as etapas iniciais do processo de desenvolvimento de software, através da utilização de conceitos de objetos para a representação da arquitetura do sistema. Dentre outras coisas, leva ao reuso de software, permite a abstração de detalhes e provê modularidade aos sistemas, tornando as atividades de manutenção e evolução menos complexas. Entretanto, esta técnica não consiste numa notação tão rigorosa, permitindo interpretações ambíguas dos modelos elaborados.

Já a SDL (Specification and Description Language) é uma linguagem de especificação formal bastante utilizada no desenvolvimento de produtos da área de telecomunicações, permitindo a definição de características estruturais e dinâmicas dos sistemas, ou seja, representa a comunicação dos elementos do sistema e o seu comportamento, podendo utilizar diferentes níveis de abstração. Além disso, por ser formal, possibilita, através de uma ferramenta apropriada, a validação, a simulação e a prototipação dos sistemas ainda nas fases iniciais de desenvolvimento.

O uso combinado de OMT com SDL proposto nesta tese em uma área fora de telecomunicações, visa impor um certo formalismo ao processo de desenvolvimento e de evolução do ambiente de software AIDA, aparecendo como uma metodologia bastante interessante para a especificação e o projeto de sistemas baseados na orientação a objetos, unindo as vantagens da orientação a objetos com as existentes no uso de uma linguagem de especificação formal. Uma das características da nova metodologia é a facilidade que oferece para a evolução de sistemas através da utilização de conceitos como herança e reuso, presentes na OMT, em conjunto com construções de SDL como *package*, *virtual* e *redefined block* e *process types* [Macário, et al., 1997a][Macário, et al., 1997b].

Este trabalho apresenta o uso desta metodologia combinada como alternativa para o desenvolvimento e a evolução do AIDA, um ambiente de software para o gerenciamento e análise de dados experimentais, em desenvolvimento na Embrapa.

A aplicação da metodologia teve início com a especificação de uma versão centralizada [Macário, et al., 1997a], baseada na descrição informal do sistema [Embrapa, 1996]. Foi elaborado o modelo de classes desta versão, e a partir dele a especificação SDL com a aplicação de conceitos básicos da linguagem. Esta versão, para ser evoluída para uma versão concorrente [Macário, et al., 1997b], incorporou novas construções presentes na versão SDL-92, como o mecanismo package e a construção virtual. A versão concorrente foi elaborada herdando a nova versão centralizada, através da construção inherits, com a redefinição de algumas das transições existentes na versão herdada, através da construção redefined. A validação desta versão levou à evolução dos sistemas existentes, com a incorporação da nova classe Initial ao modelo OMT, e correspondente processo Initial em SDL. As primeiras versões do AIDA [Macário et al., 1997a][Macário et al., 1997b], e suas evoluções (capítulo 3), apresentavam uma visão global do sistema.

Numa etapa seguinte, foi proposta a definição do sistema AIDA_CORBA distribuído para ser executado numa plataforma CORBA. A definição desta nova versão levou a uma revisão total dos modelos existentes, com o detalhamento das operações de cada classe no modelo OMT. A construção procedure foi utilizada para mapear as operações das classes na especificação SDL. A versão CORBA (capítulo 4) também manteve os conceitos de reuso e herança, e por esta razão foi especificada uma versão AIDA_CORBA centralizada que foi evoluída para a versão distribuída. Além disso, também foram abordados detalhes como o tratamento de arquivos virtuais, utilizando para isso a linguagem ASN.1. Na especificação do tratamento de arquivos de dados foi usada uma construção disponível na linguagem SDL-92, as chamadas remotas a procedimentos - RPC. Para que o sistema pudesse ser executado em ambiente CORBA, foram definidas as interfaces externas dos elementos do sistema em linguagem IDL.

O uso desta nova metodologia, em conjunto com a ferramenta CASE SDT[5], permitiu a simulação e a validação dos sistemas ainda nas fases de análise e projeto (design), identificando erros e falhas que provavelmente só se tornariam visíveis na fase de teste, o que reduz o tempo de desenvolvimento do software e leva ao desenvolvimento de produtos de qualidade.

A metodologia combinada OMT + SDL mostrou-se como uma ótima alternativa para o desenvolvimento do AIDA, aparecendo como uma excelente opção no desenvolvimento de outros ambientes de software dentro e fora da Embrapa.

5.3. Sugestões para o Futuro

Aqui são apresentadas algumas da eventuais especializações decorrentes deste trabalho. São elas:

- Simulação do sistema AIDA distribuído em ambiente CORBA, com uma ferramenta apropriada;
- Extensão do conceito RPC a todos os procedimentos do sistema AIDA_CORBA;
- Migração da metodologia para a nova versão da linguagem SDL (SDL-96, SDL-2000, etc);
- Utilização de outras técnicas de orientação a objetos (Fusion, UML, etc), como alternativa à técnica OMT.

[5] SDT Pacote SDT, versão 3.02 (SDT Base, MSC Editor, Simulator e Validator) e versão 3.2 (SDT Base, MSC Editor, Simulator, Validator e OMT Editor): adquirido pelo DT/FEEC/UNICAMP através de Projeto Temático - FAPESP (Proc. 91/3660-0).

Bibliografia

[Belina et al., 1991] Belina, F.; Hogrefe, D.; Sarma, A. *SDL with applications from protocol specification.* Prentice Hall International, 1991. 275p

[Belloquim, 1997] Belloquim, A. "Qualidade de software: um compromisso de toda a empresa" *Developers' Magazine*, número 10, 1997. p12-15 .

[Booch, 1991] Booch, G. *Object oriented design with applications.* The Benjamin/Cummings Publishing Conpany, 1991, 578p.

[Carracedo et. al., 1997] Carracedo J., Ramos C., Diego R., González C., Gil J.J., Rodriguez E., Bjorkander M. *Introducing SDL in the Development of CORBA-compliant applications* In:SDL'97 TIME FOR TESTING SDL, MSC and Trends, Eighth SDL Forum: Proceedings. Every-France Elsevier Science Publishers, 1997. p.351-365.

[Chaim et al., 1994] Chaim, M.L.; Ternes, S.; Delfino A.; Aoki, R.; Alvim, L.; Medeiros, S.; Macário, C.G.N.; Moura, M.F.; Higa, R.H.; Arantes, M.P.C.; Porto, J.R.; Bacarin, E.; Festa, M. *Ambiente Integrado para Desenvolvimento e Análise - AIDA.* Campinas: Embrapa-CNPTIA, 1995 não paginado (EMBRAPA-CNPTIA. Projeto 12.0.96.121.00).

[Chaim et al., 1996] Chaim, M.L.; Ternes, S.; Delfino A.; Alvim, L.; Medeiros, S.; Macário, C.G.N.; Moura, M.F.; Higa, R.H.; Porto, J.R.; Bacarin, E.; Festa, M. *Ambiente Integrado para Desenvolvimento e Análise - AIDA.* Campinas: Embrapa-CNPTIA, 1996 não paginado (EMBRAPA-CNPTIA. Relatório de andamento de projeto 12.0.96.121.00).

[Embrapa, 1996] Embrapa/ Centro Nacional de Pesquisa Tecnológica em Informática para a Agricultura(Campinas, SP). *Software AIDA - especificação de requisitos.* Campinas, 1996.

[Færgemand et. al. 1997] Færgemand O.; Olsen A. *New features in SDL-92.* Disponível: site SDL[literature]. URL: http://www.dr.dk/public/SDL/litt.html# papers Consultado em 20 fev. 1997.

[Gehani, 1982] Gehani, N. "Specifications: formal and informal - a case study" *Software - Practice and Experience*, vol. 12, 1982. p433-444.

[ITU-T, 1993a] ITU-T(Geneve, Switzerland). *Recommendation Z.100 CCITT specification and description language (SDL): programming languages.* Geneve, 1993

[ITU-T, 1993b] ITU-T(Geneve, Switzerland). *Recommendation Z.120. message sequence chart* (MSC). Geneve, 1993.

[ITU-T, 1994] ITU-T(Geneve, Switzerland). *Recommendation X.680-683 abstract syntax notation one (ASN1).* Geneve, 1994.

[ITU-T, 1995] ITU-T(Geneve, Switzerland). *Recommendation Z.105 SDL Combined with ASN.1 (SDL/ASN.1)* Geneve, 1995.

[Loftus et. al., 1997] Loftus C., Sherratt E., Inocêncio E., Viana, P. *The Unifications of OMT, SDL and IDL for Service Creation* In:SDL'97 TIME FOR TESTING SDL, MSC and Trends, Eighth SDL Forum: Proceedings. Every-France Elsevier Science Publishers, 1997. p.443-457.

[Macário et al., 1991] Macário, C.G.N.; Bonfim, W.S.; Chaim, M.L.; Antunes, J.F.G.; Ternes, S.; Aoki, R.; Alvim, L.; Pacheco, O.I.P.; Palmieri, S.; Festa, M.N.; Gaspar, D.M.; Serra, R.; Higa, R.H.; Arantes, M.P.C. *Evolução do Ambiente de Software NTIA.* Campinas: EMBRAPA-CNPTIA, 1994. não paginado. (EMBRAPA-CNPTIA. Projeto 12.0.94.071.00).

[Macário, et al., 1997a] Macário, C.G. N.; Pedroso Júnior, M.; Borelli, W. C. *OMT+SDL: An alternative methodology for the AIDA development.* In: III CONGRESO INTERNACIONAL DE INGENIERIA INFORMATICA, Buenos Aires. ICIE 96-97: Proceedings. Buenos Aires:Universidad de Buenos Aires - Faculdad de Ingenieria - Departamento de Computacion, 1997. p.351-365.

[Macário, et al., 1997b] Macário, C.G. N.; Pedroso M. J; Borelli, W. C. *Designing a multi-user software environment for development and analysis using a combination of OMT and SDL92.* In:SDL'97 TIME FOR TESTING SDL, MSC and Trends, Eighth SDL Forum: Proceedings. Evry-France Elsevier Science Publishers, 1997. p.03-17.

[Meyer, 1988] Meyer, B *Object-oriented software construction*, Prentice Hall International, 1988, 534p.

[Pacheco et al., 1997] Pacheco, H.A.; Santos, A.D. dos; Figueiredo, K.N.; Chaim, M.L.; Pedroso Júnior, M.; Fileto, R. *Cartilha Azul: guia do processo de desenvolvimento de software do CNPTIA.* Campinas: Embrapa-CNPTIA, 1997. 53p.

[Reed, 1997] Reed R. *SDL and MSC in International Organizations: ITU-T* In:SDL'97 TIME FOR TESTING SDL, MSC and Trends, Eighth SDL Forum: Proceedings. Every-France Elsevier Science Publishers, 1997. p.231-241.

[Rumbaugh et al., 1991] Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy,F.; Lorensen, W. *Object oriented modeling and design.* New Jersey: Prentice Hall International, 1991. 500p.

[SAS, 1985] SAS INSTITUTE (Cary, NC, USA). *SAS user's guide:basics.* Cary, 1985. 1290p.

[Siegel, 1996] Siegel, J. *CORBA fundamentals and programming* John Wiley & Sons Inc, USA, 1996.

[Taurion, 1997] Taurion, C. "A busca da qualidade para alavancar os negócios" *Developers' Magazine*, número 10, 1997. p10-11.

[TELELOGIC AB, 1995a] TELELOGIC AB (Malmö, Sweden). *Getting started with SDT3.02.* Malmö, 1995.

[TELELOGIC AB, 1995b] TELELOGIC AB (Malmö, Sweden). *SDT3.02 methodology.* Malmö, 1995.

[TELELOGIC AB, 1996a] TELELOGIC AB (Malmö, Sweden). *SDT3.1 methodology guidelines - part1: the SOMT method.* Malmö, 1996.

[TELELOGIC AB, 1996b] TELELOGIC AB (Malmö, Sweden). *SDT3.1 methodology guidelines - part2: pratical SDL guidelines.* Malmö, 1996.

[TELELOGIC AB, 1996c] TELELOGIC AB (Malmö, Sweden). *SDT3.1 getting started - part2: SOMT and CORBA tutorials.* Malmö, 1996.

[TELELOGIC AB, 1997] TELELOGIC AB (Malmö, Sweden). *SDT3.2 getting started - part2: SOMT and CORBA tutorials.* Malmö, 1997.

APÊNDICE A - A linguagem de especificação SDL

A linguagem SDL - Specification and Description Language [ITU-T, 1993a] começou a ser desenvolvida em 1972 pela CCITT (hoje pela ITU-T), visando padronizar a especificação e a descrição de sistemas de telecomunicações. Sua primeira versão foi lançada em 1976, seguida por novas versões em 1980, 1984, 1988 e 1992 [Belina et al., 1991]. Nas suas versões mais recentes, a linguagem foi expandida, onde a versão SDL-92 incorporando conceitos de objetos [Færgemand et. al. 1997].

Uma especificação em SDL apresenta uma descrição geral do sistema, que é feita utilizando-se blocos, canais e tipos abstratos de dados, denominados sorts. A especificação dos blocos se dá através de processos, ou de outros blocos, e de sinais. Os processos, que especificam o comportamento do sistema, são representados através de máquinas de estado finitas.

Por possuir uma sintaxe e uma semântica bem definidas uma especificação em SDL pode ser executada, permitindo a validação ou simulação do sistema especificado

Uma especificação SDL provê diversas visões de um sistema [TELELOGIC AB, 1996a], sendo elas:

1. estrutural, através da:
   - decomposição hierárquica do software em sistema, blocos e processos;
   - hierarquia de tipos, através de herança e especialização (apenas no SDL-92);
2. de comunicação, através de:
   - sinais assíncronos e seus parâmetros opcionais;
   - chamadas remotas a procedimentos (apenas no SDL-92);
3. comportamental:
   - através dos processos;
4. visão de dados através de:
   - tipos abstratos de dados (sorts) definidos;
   - tipos de dados ASN.1

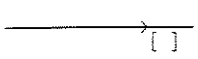
A estrutura estática do sistema é definida em termos de blocos. Os blocos se comunicam por canais. Já a estrutura dinâmica é descrita em termos de um conjunto de

processos que executam em paralelo. Um processo é representado por uma máquina finita de estados. Cada processo é independente do outro, comunicando-se entre si através dos sinais descritos nas rotas de sinais.

Especificação de Sistemas em SDL

System

O sistema SDL representa a estrutura estática do sistema e é definido por um ou mais blocos, que comunicam-se entre si, ou com o ambiente, através de canais de sinais. Todos os sinais e tipos de dados usados na sua descrição, e que serão visíveis aos seus blocos, devem ser especificados neste nível.

Canal 

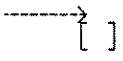
Um canal de sinal é definido para o transporte de sinais entre blocos, ou entre os blocos e o ambiente externo ao sistema. Um canal pode ser uni ou bidirecional, ou seja, pode conter sinais apenas de entrada ou de saída, ou ambos. Na sua definição são descritos os sinais e/ou listas de sinais que contém, e o início ou o fim de um canal sempre tem de ser um bloco ou o ambiente.

Text 

Consiste no elemento usado em SDL para declarações de variáveis e sinais.

Block 

Um bloco é constituído por um ou mais processos que interagem entre si e com os canais do sistema, através de rotas de sinais. Todos os sinais, listas de sinais e tipos de dados usados neste nível, e que não tenham sido declarados no nível do sistema, devem ser especificados.

Signal Route 

As rotas de sinais são definidas para o transporte de sinais entre processos, e da mesma forma que os canais, podem ser uni ou bidirecional. Os sinais definidos nas rotas devem ser os mesmos que foram definidos nos canais que se conectam a estas rotas.

Process 

O comportamento do sistema SDL é especificado através de um conjunto de processos que executam em paralelo. Cada processo é descrito como uma máquina finita de estados com dados, independente dos demais.

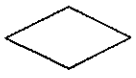
Um processo pode mandar e receber sinais para outros processos, e estes sinais devem ser especificados através das rotas de sinais do processo.

Cada processo possui uma fila de sinais associada, onde todos os sinais recebidos são armazenados e consumidos numa política do tipo FIFO.

Principais elementos para a descrição dos processos.

Task 

A construção *task* é utilizada para definir uma tarefa, podendo ser usada uma descrição informal *'texto'*. Caso mais de um comando seja definido, eles devem ser separados por ';' (ponto e vírgula).

Decision 

A construção de decisão consiste de uma questão e de um determinado número de respostas. Esta estrutura não checa se as respostas cobrem todas as possibilidades possíveis para a questão. Entretanto, devem existir duas ou mais respostas exclusivas, onde apenas uma delas pode ser *else*. As respostas podem ser texto informal ou não.

Start 

O *start* é definido apenas uma vez no processo, indicando o início da sua execução. Este elemento deve ser seguido de uma transição ou de um estado.

Stop 

Responsável por terminar a execução do processo.

Create 

Construção utilizada para criar uma instância de outro processo existente no mesmo bloco deste processo.

State 

Define um estado. A mudança de um estado só é possível através da recepção de um sinal.

 refere-se a todos os estados, estabelecendo ações comuns a todos eles.

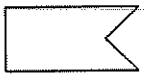
 o estado não se altera após a execução de alguma ação.

Save 

Este elemento vem sempre depois de um estado. É utilizado quando se quer salvar um sinal recebido que não deve ser consumido naquele momento. Esta construção altera a política FIFO da fila de sinais do processo.

Comment 

Utilizado para descrever algum comentário.

Input 

Indica um sinal de entrada. Deve vir sempre depois de um estado, o qual pode apresentar mais de um input.

Output 

Indica um sinal de saída, usado para mandar sinais e dados a outros processos. Os tipos dos dados enviados pelo *output* devem ser os mesmos que os do *input* no processo receptor. O processo destino deve existir no momento do envio do sinal.

O endereço do processo receptor do sinal pode ser

- *explícito*: especifica-se o endereço do receptor ou de saída (*to <end>* indica para qual processo vai o sinal; ou *via* indicando o *gate* ou a *rota de sinal* por onde o sinal passa.)
- *implícito*: omitindo-se o endereço do destino

Procedure Call 

Chamada a procedimento. Só pode aparecer num processo, serviço ou outro procedimento.

Procedure 

Os procedimentos (procedures) funcionam como nas linguagens de programação, sendo úteis quando existem partes repetidas do processo, ou para se ter uma visão melhor do sistema.

Um procedimento é criado quando uma *procedure call* é interpretada. Neste ponto, o processo que a chama é interrompido até que a sua execução termine.

A definição de um procedimento é similar a de um processo, e também tem fila de sinais associada. Todas as variáveis definidas no processo são visíveis ao procedimento, mas os seus estados não.

Um procedimento pode ter parâmetros, que são definidos da seguinte forma

;FPAR.....

xxx yyy zzz;

onde:

xxx especifica o tipo do parâmetro, podendo ser:

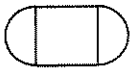
in: parâmetros apenas de entrada

in/out: parâmetros de entrada e saída

returns: define um valor de retorno

yyy define o nome do parâmetro;

zzz define o tipo de dado do parâmetro;

Start 

Indica o início do procedimento, devendo ser definido uma única vez.

Return 

Símbolo de retorno da procedure. Quando é executado, retorna o controle para o processo.

Tipos Abstratos de Dados (TADs) em SDL

Os tipos abstratos de dados da linguagem SDL são denominados SORTS. Existe um conjunto pré-definidos de sorts para permitir o trabalho com dados nas especificações dos processos [TELELOGIC AB, 1996b]. O uso destes sorts se dá através de variáveis. O comando *DCL* é utilizado para a sua declaração, e := para suas atribuições.

Os sorts mais simples, comuns às maioria das linguagens de programação, são:

integer, boolean, real, natural, character, charstring

Além destes, existem alguns *sorts* mais específicos como:

- Pid: sort do tipo *identificador de processo*; cada processo apresenta quatro tipos de Pid, descritos a seguir:

  self - identificador do próprio processo;

  sender - identificador do processo que mandou o último sinal recebido;

  offspring - identificador do último processo criado por ele;

  parent - identificador do seu processo criador

- *Time* e *duration*: sort usados para controle de tempo, onde *time* especifica o tempo absoluto, e *duration* o tempo relativo; existem algumas expressões associadas a estes sorts; são elas:

  timer: consiste em um objeto que gera um sinal de tempo; para ser utilizado, o timer tem de ser declarado na área de declarações através da contrução *DCL*;

  active: ativa o timer;

  reset: desativa o timer;

  set: atribui um determinado tempo ao timer;

  now: expressão pré-definida, usada para ativar o timer com o tempo absoluto no momento.

A linguagem SDL também possui construções para a criação de novos tipos de dados (sorts). Algumas delas são apresentadas a seguir:

- *newtype* : define novo tipo;
- *syntype*: restringe o limite de um sort existente;

- *literal*: define valores enumerados para um sort;
- *struct*: define uma estrutura de dados, composta por outros sorts;
- *array:* consiste num vetor de sorts. A sua declaração apresenta dois sorts: um que será como índice de acesso e o outro que corresponde a cada elemento do array;
- *adding*: adiciona novos operadores a algum sort já definido;

Versão SDL-92

A principal evolução da linguagem SDL-92, com relação às versões anteriores, foi a incorporação de alguns mecanismos para permitir o uso de conceitos da orientação a objetos [TELELOGIC AB, 1995b]. Nas linguagens orientadas a objetos, o conceito de classe é usado para representar um grupo de objetos com propriedades semelhantes e mesmo comportamento, enquanto os objetos são as instâncias das classes. Além disso, também permitem a criação de novas subclasses através da herança/especialização de classes existentes.

Em SDL-92 uma classe corresponde a um tipo (type) e os objetos às instâncias (instances). Uma das principais diferenças entre o SDL-88 e SDL-92 é que os elementos são definidos diretamente em SDL-88 e já no SDL-92 podem ser definidos usando tipos e depois instanciados. Desta forma, as especificações definem um tipo, e os tipos são usados na descrição do sistema através de suas instâncias.

Uma das vantagens do uso de tipos é que torna-se possível que a criação de tantas instâncias de um processo quantas forem necessárias durante a execução do sistema, e principalmente, tornam possível o uso dos conceitos de herança e de especialização.

Um tipo pode ser definido como uma especialização de outro tipo, e neste caso o tipo original é o supertipo e o especializado é o subtipo. O subtipo herda todas as propriedades do supertipo, sendo possível a adição de novas propriedades ou a redefinição de outras existentes.

Tipos Existentes em SDL-92:

System Type

Define um sistema. Basicamente é definido para ser reusado por outro sistema, através do mecanismo de package.

Block Type 

Define um conjunto de blocos com características semelhantes. As instâncias de blocos constituem o principal conceito de estruturação num sistema SDL, normalmente contendo uma ou mais instâncias de processos.

A definição do nome da instância de um bloco é formada por duas partes:

xxx: yyy

onde:

- *xxx* é o nome da instância;
- *yyy* é o nome do *block type* sendo instanciado.

Process Type 

Define um conjunto de processos com características semelhantes.

Um instância de processo é uma máquina de estados que trabalha de forma autônoma e concorrente com outras instâncias de processos, que se comunicam assincronamente usando mensagens discretas chamadas sinais. Os sinais carregam o endereço de quem manda e de quem recebe o sinal, e valores, se necessário. Além disso, uma instância de processo tem um tempo de existência.

A definição do nome da instância de um processo é formada por três partes:

xxx (a,b): yyy

onde:

- *xxx* é o nome da instância do processo
- *(a,b): a indica* o número de instâncias do processo criadas no início da execução do sistema e *b* indica o número máximo de instâncias que podem ser criadas ao mesmo tempo;
- *yyy* é o nome do process type sendo instanciado

Outros conceitos de Orientação a Objetos em SDL-92:

Gate

O gate faz parte da definição do tipo e é usado para conectar as instâncias com a sua estrutura superior, que no caso de um *process type* é o bloco e no caso de um *block type* é o sistema. O gate determina quais os sinais que o tipo pode receber ou enviar.

O gate de blocos é um ponto de conexão de canais, e o de processo é um ponto de conexão de rotas de sinais. Assim, na especificação dos *outputs* dos processos, os sinais passam a ser enviados *via* gate e não mais *via* signal route, como era no SDL-88.

Herança e Especialização de Tipos

No processo de evolução de software é muito comum a especialização de um tipo existente, com a inclusão de novas propriedades ou a redefinição de antigas. Quando isto ocorria na versão SDL-88, era preciso rescrever todo o elemento com as suas alterações. Já na SDL-92, isto é obtido através da herança de tipos, onde todas as propriedades do supertipo são herdadas pelo subtipo, incluindo a definição dos gates, devendo ser especificadas apenas as diferenças existentes entre estes tipos.

A herança de tipos é feita através da construção INHERITS. Quando a herança é feita apenas para a adição de novas funcionalidades, deve se usar a construção INHERITS <sistema herdado> ADDING, que indica que a entidade herda outra adicionando novas propriedades.

Quando a especialização do tipo é feita não apenas com a adição de funcionalidades, mas também com a alteração algumas existentes, denomina-se redefinição. Neste caso, é necessária a utilização de outra construção além da INHERITS. A propriedade a ser redefinida deve ser definida originalmente como VIRTUAL e na sua redefinição é utilizada a construção REDEFINED, indicando que esta propriedade já foi definida em algum lugar.

Os tipos que podem ser redefinidos em um sistema são os *block* e *process types*. Um *system type* não pode ser definido como virtual, e por esta razão, não pode ser redefinido. Uma procedure, apesar de não ser um tipo, também pode ser definida como virtual e ser redefinida. A redefinição de transições se dá sempre através da redefinição de processos.

Redefinição de Block Types

Para que um *block type* possa ser redefinido, ele deve ser definido originalmente como *virtual*. O bloco especializado herda todas as propriedades do supertipo, incluindo os processos, instâncias e sinais.

As instâncias definidas para os seus processos podem ser reusadas, mesmo que exista a necessidade de inclusão de novos sinais. Neste caso a instância é desenhada como pontilhada, indicando o reuso, e a definição dos novos sinais é feita em um nova rota de sinais, nos gates herdados.

Redefinição de Transições

A especialização de processos pode ser feita de duas formas: com a inclusão de novas funcionalidades, que pode ser a inclusão de novos estados, ou de novas transições em estados existentes, ou com a redefinição de transições existentes.

Nas transições, não apenas as transições de *input* podem ser redefinidas, mas também as transições de *start* e de *save*.

Para que seja possível a redefinição de uma transição, ela tem de ser definida no supertipo como VIRTUAL e assim, a propriedade herdada será redefinida, através da construção REDEFINED.

A propriedade redefinida será considerada como VIRTUAL para outro tipo que for reutilizá-la.

Packages

Um dos conceitos mais importantes em SDL-92 é o PACKAGE. O package permite que a definição de um tipo possa ser usada em diferentes sistemas, com um propósito similar a uma biblioteca de classes, levando ao reuso de especificações.

Um package pode conter definições de tipos, de listas de sinais, de sorts, etc, e pode ser reusado integralmente num sistema, ou apenas com algumas de suas definições. Uma vez que um package é usado por um sistema, as suas definições aparecem como se tivessem sido realmente definidas no próprio sistema.

Um package é incluído em um sistema através da cláusula USE, que só pode ser usada na definição do sistema ou de um outro package, neste caso permitindo a criação de uma hierarquia de packages.

Quando um tipo é instanciado, é importante que os canais ou rotas de sinais que estão conectados às instâncias carreguem os sinais corretos, de acordo com os gates do tipo definido.

Procedures em SDL-92

Uma procedure também pode ser considerada um tipo, mas ao contrário dos demais tipos, ela não tem de ser instanciada para ser usada. Em SDL-88 elas tinham de ser usadas no mesmo nível em que foram definidas, ou seja não podiam ser globais.

Já no SDL-92, esta restrição foi removida, e a procedure pode ser definida em qualquer nível do sistema.

Procedures Remotas

Um outro conceito que tornou-se disponível na versão SDL-92 é a chamada remota a procedimento - RPC (Remote Procedure Call) [ITU-T, 1993a]. O uso primário de uma RPC é quando alguma informação é requerida por outro processo e esta informação está confinada a um escopo limitado. A forma comum de resolver esta situação é a definição de um sinal de requisição ao processo onde o procedimento está definido, o seu cálculo e um novo sinal de retorno com a resposta. A RPC é uma forma rápida de executar este processo, pois faz uso de sinais implícitos entre as partes envolvidas, normalmente chamadas de cliente-servidor. Isto significa que não é necessário definir canais ou sinais entre o cliente (entidade onde o procedimento é chamado) e o servidor (entidade na qual o procedimento é definido).

As regras necessárias para realizar uma RPC são as seguintes:

- o procedimento no servidor deve ser definido como *exported*, i.e., o servidor deve ser avisado que é permitido o uso externo do procedimento em questão;
- o procedimento deve ser declarado como *imported* pelo cliente, i.e., o cliente deve ser avisado que a definição do procedimento encontra-se em algum lugar;
- o procedimento deve ser declarado como *remote* num nível visível ao cliente e ao servidor.

APÊNDICE B - A Arquitetura CORBA

O Object Management Group - OMG [Siegel, 1996] é um consórcio internacional formado por empresa de computação, voltado para a especificação de padrões para ambientes orientados a objetos.

A arquitetura Common Object Request Broker Arquitecture - CORBA [Siegel, 1996] é um deles, e provê uma infra-estrutura para a comunicação entre objetos distribuídos em ambiente heterogêneos, permitindo a um objeto usar os serviços disponíveis em outro objeto, numa arquitetura tipo cliente-servidor, onde o cliente é o objeto requisitante, e o servidor é o objeto que provê o serviço requisitado.

Para o acesso aos serviços disponíveis nos objetos do ambiente, a arquitetura CORBA requer que todos as interfaces dos objetos sejam expressas em uma linguagem comum, denominada OMG IDL - Interface Definition Language.

Um objeto em CORBA é visto como um componente de software do tipo "plug and play" que apresenta características de encapsulamento, herança e polimorfismo.

A arquitetura CORBA conecta apenas objetos e não aplicações. Como um ambiente integrado de sistemas requer também a conexão de aplicações, o OMG apresenta a OMA (Object Managment Architecture), que baseia-se em CORBA. A arquitetura OMA engloba a visão do ambiente de componentes de software, e é composta por três componentes principais, como pode ser visto na Figura 5.1.

Os serviços CORBA provêem funcionalidades básicas que quase todos os objetos utilizam como cópia, nomeamento e serviços de diretório, bem como serviços mais sofisticados como o TRADER (algo parecido com as "páginas amarelas").

As facilidades CORBA são serviços voltados para as aplicações.

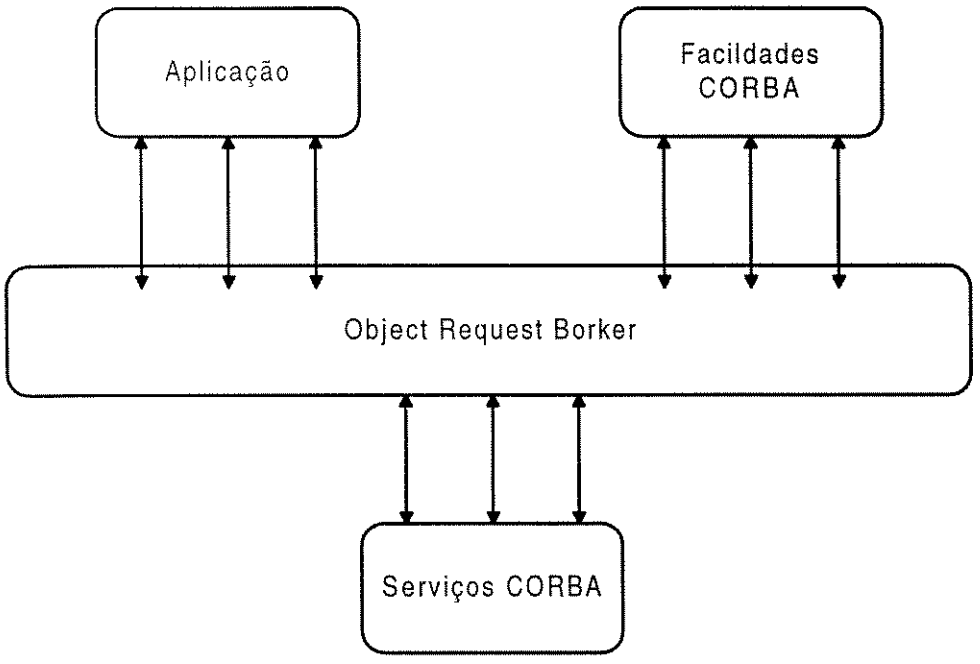


Figura 5.1 – Object Management Arquitecture

As requisições emitidas pelo cliente são sempre gerenciadas pelo Object Request Broker - ORB. Todos os detalhes de distribuição dos objetos ficam no ORB. A Figura 5.2 apresenta a arquitetura de um ORB CORBA, ilustrando os seus principais componentes:

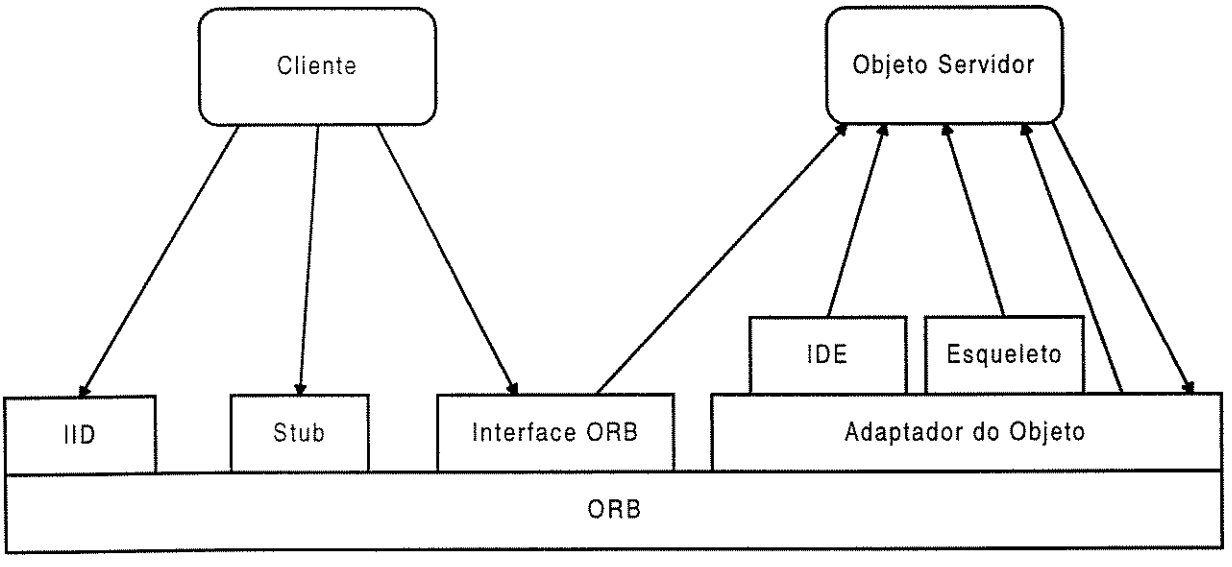


Figura 5.2 – Object Management Arquitecture

O servidor implementa as operações definidas na interface IDL CORBA em uma linguagem de programação como C^{++} ou Java, e que são oferecidas a outros objetos.

Cliente: é a entidade que invoca uma operação do objeto. O acesso remoto a estas operações deve ser transparente ao cliente, funcionando como uma chamada comum ao um método de um objeto.

ORB: provê um mecanismo para a comunicação transparente entre os objetos cliente e servidor. O ORB simplifica a programação distribuída, livrando o cliente e o servidor de detalhes para a invocação de métodos, fazendo as requisições funcionarem como uma chamada local a um procedimento. Quando um cliente invoca um método de outro objeto, cabe ao ORB localizar a implementação deste objeto, ativá-lo, se necessário, enviar a requisição e retornar a resposta ao processo invocador. O cliente não deve fazer nenhuma consideração sobre a forma de invocação ou sobre a referência do objeto, para indicar que o determinado objeto é local ou remoto. Todos os detalhes de invocação são resolvidos pelo ORB.

Interface ORB: provê um conjunto de facilidades, desacoplando as aplicações da implementação do ORB.

CORBA IDL stubs e esqueletos: fazem a ligação entre as aplicações cliente e servidor respectivamente, e o ORB. A compilação do arquivo de interfaces IDL gera o esqueleto da implementação e o stub do cliente. Enquanto o esqueleto contém chamadas a funções, o stub contém declarações de funções. No cliente são escritas as chamadas a funções correspondentes que resolvem as questões de distribuição para o stub.

Interface de Invocação Dinâmica (IID) permite aos usuários usar novos objetos assim que eles tenham sido adicionados a qualquer ORB da sua rede. A IID permite a um cliente no tempo de execução:

- descobrir novos objetos,
- descobrir suas interfaces,
- recuperar suas definições,
- construir e despachar suas invocações e
- receber a resposta resultante ou informação de execução de e para os objetos cujos módulos não estão ligados aos seus stubs clientes.

Interface Dinâmica do Esqueleto (IDE) funciona de maneira análoga à IID, só que no lado do servidor.

Adaptador de Objetos: auxilia o ORB no envio de requisições ao objeto e na sua ativação, associando implementações dos objetos ao ORB.

ORB - Object Request Broker

Existem dois problemas básicos a serem resolvidos pelo ORB para o acesso aos serviços de um objeto: a sua localização - como endereçar uma invocação para um objeto particular e a sua tradução - como a invocação é traduzida para um formato de dados de um ORB estrangeiro e a resposta traduzida de volta? A referência de objeto é definida para solucionar o primeiro problema, e o segundo é resolvido com a linguagem de definição de interfaces IDL.

Referência de Objeto

Todo objeto em CORBA, independente de sua duração no sistema, tem uma referência de objeto própria, que á atribuída pelo seu ORB na sua criação e permanece até que ele seja explicitamente excluído.

Os clientes obtêm as referências de diferentes formas e a associam com a invocação de acordo com a linguagem de mapeamento que estão usando. Esta associação permite ao ORB direcionar a invocação ao objeto determinado.

Ou seja, não é simplesmente um endereço de memória ou de rede do objeto. O consórcio internacional OMG permite que cada vendedor de ORB implemente a tradução da referência para seu objeto alvo da forma que achar melhor para seus sistemas alvos, desde que os requisitos de validade sejam mantidos.

Com o requerimento de que qualquer ORB entende uma referência de objeto em qualquer hora, a referência de objeto tem uma função importante em um sistema distribuído. As referências de objeto podem ser passadas para os ambientes usando arquivos de dados, serviços de nomeamento ou "trading", arquivos públicos de localização ou outras formas. Qualquer aplicação usando um ORB na rede pode recuperá-los e passá-los para seu próprio ORB para invocar o objeto.

O usuário do ORB deve saber como manusear a referência de objeto para emitir uma requisição, e o ORB obtém esta invocação, repassando-a para o objeto alvo.

O IDL e o ORB

A arquitetura CORBA requer que o ORB armazene a definição IDL de todos os seus objetos em um repositório de interface (RI) o que vem a ser a chave do sistema distribuído. Este RI deve estar disponível não apenas para o ORB, mas também para todos os clientes e implementações de objetos e utilidades como browsers de objetos e depuradores. Assim, usando a IDL padrão, interfaces podem ser definidas e adicionadas à RI, modificadas,

recuperadas etc, e através das árvores de hierarquia podem ser obtidos os tipos de objetos. Desta forma, sabendo-se o tipo e a ordem dos argumentos nas mensagens, fica possível a comunicação entre ORBs, onde o formato dos dados e a ordem dos bytes podem ser traduzidos sempre que necessário. Um outro uso potencial do RI é a invocação dinâmica.

Interoperabilidade

A interoperabilidade em CORBA é feita baseada na comunicação ORB-a-ORB. O cliente não faz nada diferente da invocação local. Ele passa sua invocação usual baseada em IDL para o seu ORB local. Se a sua invocação contém a referência de um objeto local, o ORB a roteia para ele. Caso contrário, o ORB roteia para o ORB remoto, que a roteia para o objeto alvo.

Cada ORB é requisitado para manter no mínimo 2 (possivelmente grandes) bases de dados ou sistemas de diretórios: o RI, com sua coleção de definições de interfaces, e um repositório de implementações, com informações sobre as implementações de objetos disponíveis. Detalhes de comunicação devem ser sincronizados: protocolos de redes ORB devem "casar" ou os gateways devem fazer a tradução entre eles.

O cliente não pode informar sobre a maneira de invocação ou sobre a referência do objeto, indicando se o dado objeto é local ou remoto. Todos os detalhes de invocação são resolvidos pelo ORB.

A comunicação Inter_ORB é a característica chave que dá ao CORBA sua flexibilidade sem paralelos. Clientes e implementações de objetos podem residir em ORBs de diferentes vendedores, de diferentes plataformas, de diferentes sistema operacionais, de diferentes redes e serem escritos em diferentes linguagens de programação por diferentes programadores que nunca se viram ou se falaram, e ainda assim vão interoperar perfeitamente desde que o cliente e o objeto usem a mesma sintaxe IDL e semântica.

Tudo o que um programador necessita para escrever um objeto cliente que acesse um objeto remoto, é uma cópia de seu arquivo IDL, com a descrição sobre o que cada operação faz, e a sua respectiva referência de objeto.

Construindo um objeto CORBA

A contrução de um objeto CORBA deve seguir os seguintes passos:. definir o que é o objeto; especificar sua interface; escolher a linguagem de implementação; escolher a plataforma ou sistema operacional; escolher o ORB para conectá-lo; definir se local ou

remoto; definir o hardware e o protocolo a ser usado no caso do objeto ser remoto; definir outros aspectos como níveis de segurança.

Cada conexão requer uma decisão do implementador: no lado do objeto, saber a linguagem de implementação, e no outro, selecionar um ORB para a sua conexão. Para selecionar a linguagem, devem ser consideradas as suas aplicabilidade e disponibilidade. É aplicável se pode fazer o que a aplicação precisa, usando apenas recursos de computação. E disponível se o ORB suporta o mapeamento IDL para esta linguagem. Deve ser checado a disponibilidade do ORB na plataforma de hardware a ser criada.

Para as principais linguagens de programação, uma linguagem de mapeamento padronizada pelo consórcio internacional OMG especifica como as interfaces IDL (tipos, invocações de métodos e outras construções) são convertidas a chamadas às funções. É desta forma que o esqueleto IDL e a implementação do objeto funcionam juntos. O compilador IDL usa a linguagem de mapeamento para gerar um conjunto de chamadas de funções às operações IDL.

O programador, geralmente auxiliado por uma ferramenta, referencia o arquivo IDL e usa a linguagem de mapeamento para gerar o conjunto de comandos das funções correspondentes. Depois da compilação e ligação ("linking"), o esqueleto faz as chamadas corretas para invocar operações nas implementações de seus objetos. Como a linguagem de mapeamento é padrão, qualquer que seja o ORB utilizado, o conjunto de chamadas a funções deve ser sempre o mesmo para uma linguagem particular, já que a implementação de seu objeto acessa o esqueleto usando a mesma linguagem.

Conectando o ORB

Dois pontos do esqueleto de implementação são opostos: a conexão para o cliente, gerenciado pela interface OMG IDL, que é padrão e provê portabilidade, e a conexão do ORB do outro lado, que é proprietária, e permite ao vendedor implementar a conexão com as características de performance que o cliente quer.

As conexões com o ORB, uma para cada interface do arquivo IDL, são geradas automaticamente pelo compilador IDL. Uma vez que a interface do esqueleto ORB é proprietária, os ORB e o compilador IDL vêm em conjuntos casados, ou seja, deve ser usado o compilador IDL e o ORB de um mesmo vendedor. Assim, dado que a linguagem de mapeamento é padrão, provendo uma junção padrão entre o objeto e o ORB, e os stubs são gerados automaticamente pelo compilador IDL, pode-se trocar o ORB e apenas recompilar o seu arquivo IDL, ligando-o ao novo stub produzido.

lo desenvolvimento:

lma vez instalados os objetos, o desenvolvedor deve carregar o arquivo IDL e i-lo com seu compilador IDL local. Isto vai gerar o stub e o esqueleto do objeto, que ser descartado. Apenas o stub será usado pelo cliente para acessar o ORB. Na entação do código do cliente, a referência de objeto servidor deve ser recuperada de serviços CORBA e usada para invocar operações do objeto desejado.