Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e de Computação

Departamento de Telemática

Desenvolvimento de sistemas TINA utilizando a linguagem de especificação formal SDL com geração automática de código Java

por

Rafael Paoliello Guimarães

Engenheiro de Computação — UFES

Banca Examinadora:

Prof. Dr. Walter da Cunha Borelli	(Orientador)
Prof. Dr. Luís Fernando Faina	Univ. Fed. Uberlândia
Prof. Dr. Shusaburo Motoyama	FEEC/UNICAMP
Prof. Dr. Eleri Cardozo	FEEC/UNICAMP

Tese apresentada na Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica.

19 de Dezembro de 2001

Agradecimentos [1]

Em primeiro lugar, gostaria de agradecer ao Prof. Dr. Walter da Cunha Borelli, pela excelente orientação e pelo apoio durante todo este período, não só como orientador, mas como amigo.

Também, agradeço a meus pais, Eumenes e Ana Lúcia, por seu carinho e apoio incondicionais.

Gostaria também de agradecer sinceramente a minha irmã, Lucia, e a minha noiva, Aline, pelo carinho, apoio e paciência comigo neste tempo em que estive ausente de casa.

Finalmente, agradeço a todos meus amigos e colegas que, direta ou indiretamente, contribuíram para a realização deste trabalho.

[1]Este trabalho foi financiado pelo Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES

Resumo

Este trabalho trata do desenvolvimento orientado a objetos de sistemas distribuídos baseados na arquitetura TINA através do uso da linguagem de especificação formal SDL com geração automática de código distribuído Java baseado nas especificações desenvolvidas. Através das especificações em SDL, é conferida uma formalização comportamental aos objetos TINA, tornando possível a validação do sistema e a realização de simulações com o auxílio da ferramenta CASE SDT. As especificações em SDL são utilizadas como base para a geração de código Java, que é realizada por uma ferramenta desenvolvida neste trabalho. O código Java gerado faz uso de CORBA como plataforma distribuída de comunicação. É apresentada a utilização desta metodologia no desenvolvimento de uma proposta de especificação formal do modelo de subscrição e dos objetos da sessão de acesso TINA.

Palavras-chave: Sistemas Distribuídos, Especificação formal, TINA - Telecommunications Information Networking Architecture, Arquitetura de Serviços, Modelo de Subscrição, SDL - Specification and Description Language, MSC - Message Sequence Chart, Validação, Simulação, Orientação a Objetos, CORBA - Common Object Request Broker Architecture, Java, Geração de código.

Abstract

This work deals with the object-oriented development of distributed systems based on TINA architecture by using the SDL formal specification language with automatic distributed Java code generation, based on the developed specifications. The SDL specifications provide TINA objects with a behavioural formalization, making it possible to validate and simulate the system by using the SDT CASE tool. The SDL specifications are used as a basis for the distributed Java code, which is generated by a tool developed in this work. The generated Java code uses CORBA as a distributed communication platform. It is presented the use of this metholodogy for the development of a formal specification proposition for the TINA subscription model and the access session objects.

Keywords: Distributed Systems, Formal specification, TINA - Telecommunications Information Networking Architecture, Service Architecture, Subscription Model, SDL - Specification and Description Language, MSC - Message Sequence Chart, Validation, Simulation, Object Orientation, CORBA - Common Object Request Broker Architecture, Java, Code generation.

Publicações

R.P. Guimarães, W.C. Borelli. Uma proposta de especificação formal em SDL de uma Sessão de Acesso TINA. In: *19o Simpósio Brasileiro de Telecomunicações (SBrT'2001)*, Fortaleza, CE, Setembro 2001.

R.P. Guimarães, W.C. Borelli. Object-Oriented Development of TINA systems using SDL and Java. In: *X Congreso Internacional de Computación*, Ciudad de México, DF, México, Novembro 2001.

Acrônimos e Siglas

ACTS – Advanced Communications Technology and Services
anonUA – anonymous User Agent
ASCII – American Standard Code for Information Interchange
ASN.1 – Abstract Syntax Notation One
as-UAP – access session User Application
BNF – Back-Naur Form
CASE – Computer Aided Software Engineering
CCITT – Comite Consultatif International Telegraphique et Telephonique
CHILL – CCITT High Level Language
ConS – Connectivity Service Reference Point
CORBA – Common Object Request Broker
CSM – Communication Session Manager
DPE – Distributed Processing Environment
DCOM – Distributed Component Object Model
IA – Initial Agent
IDL – Interface Description Language
ISS – International Switching Symposium
ITU-T – International Telecommunications Union - Telecommunication
JavaCC – Java Compiler Compiler
JDK – Java Development Kit
LOTOS – Language of Temporal Ordering Specifications
MSC – Message Sequence Chart
namedUA – named User Agent
NCCE – Native Computing and Communications Enviroment
ODL – Object Description Language
OMG – Object Management Group
ORB – Object Request Broker
PA – Provider Agent
PSTN – Public Switched Telephone Network
QoS – Quality of Service
Ret – Retailer Reference Point
RPC – Remote Procedure Call
SAG – Subscription Assignment Group
SCREEN – Service Creation Engineering Environment Project
SDL – Specification and Description Language
SDT – SDL Design Tool
SF – Service Factory
ss-UAP – service session User Application
SSM – Service Session Manager
SUB – Subscription Component
TCon – Terminal Connection Reference Point
TCSM – Terminal Communication Session Manager
TINA – Telecommunications Information Networking Architecture
TINA-C – TINA Consortium
TOSCA – TINA Open Service Creation Architecture Project
TTCN – Tree and Tabular Combined Notation
UA – User Agent
UML – Unifying Modelling Language
USM – User Session Manager

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Histórico

O desenvolvimento das telecomunicações causou um grande impacto na sociedade mundial nas últimas décadas e, nos dias de hoje, exercem um papel cada vez mais significativo em nossa sociedade. Este desenvolvimento traz consigo uma demanda cada vez maior por novos serviços.

Esses novos serviços necessitam de uma infra-estrutura complexa que suporte os mais variados tipos de serviço, desde os mais simples até os mais complexos. Neste contexto, a infra-estrutura de telefonia tradicional (PSTN - *Public Switched Telephone Network*) não atende mais os requisitos mínimos desta nova realidade. Surge assim a necessidade da criação de um novo ambiente para o provimento destes serviços, um ambiente distribuído.

Uma arquitetura distribuída é necessária por diversos motivos, dentre eles pode-se citar a cooperação entre processos e/ou serviços, o compartilhamento de recursos, o ganho na segurança (confiabilidade) e o ganho de desempenho devido ao processamento paralelo.

Entretanto, uma série de problemas dificultam a implantação de um ambiente distribuído, como a heterogeneidade das redes, dos sistemas operacionais, etc, além da complexidade das relações existentes entre as diversas entidades distribuídas. Além disso, juntamente com estas dificuldades, ainda existe a demanda por ciclos mais rápidos de desenvolvimento de novas aplicações. Ou seja, são necessárias aplicações distribuídas cada vez mais complexas e que devem ser desenvolvidas no menor tempo possível.

Neste contexto, o CORBA [1, 2] (*Common Object Request Broker Architecture*) surge com o intuito de fornecer um meio padronizado de comunicação entre os objetos distribuídos, resolvendo assim o problema da heterogeneidade das redes. CORBA se localiza na camada

OSI de aplicação, realizando o papel de mediador entre as aplicações e as camadas de rede inferiores.

Por outro lado, visando facilitar o processo de desenvolvimento de serviços de telecomunicações, um conjunto de empresas e instituições [1] se uniram e formaram um consórcio chamado TINA-C [3, 4] (*Telecommunications Information Networking Architecture - Consortium*), que tem por objetivo definir uma arquitetura distribuída aberta que suporte a criação, implantação, provimento e gerência de serviços de telecomunicações — a arquitetura TINA [5, 6, 7, 8, 9].

TINA disponibiliza um ambiente distribuído bem estruturado onde podem ser oferecidos todos os tipos de serviço, desde os mais simples até os mais complexos existentes. Para isso, TINA faz uso de uma camada chamada DPE (*Distributed Processing Environment*) que é responsável por prover um ambiente distribuído sobre o qual as aplicações executam. CORBA foi selecionado como o DPE a ser utilizado em ambientes TINA.

Devido a sua enorme complexidade, TINA é dividida logicamente em três arquiteturas: a arquitetura de serviços, a arquitetura de computação e a arquitetura de rede. A primeira, é responsável pelo provimento e gerência dos serviços disponibilizados. A segunda é responsável pelos recursos computacionais necessários para o provimento dos serviços desejados. Já a última é responsável pelos recursos de rede necessários.

Afim de disponibilizar estes serviços de telecomunicações, a arquitetura TINA é baseada em uma série de objetos que se relacionam oferecendo as mais diversas funcionalidades. Estes objetos são definidos nas recomendações TINA através da linguagem ODL [10] (*Object Description Language*). A linguagem ODL é responsável pela especificação das interfaces destes objetos, ou seja ela especifica o padrão utilizado por estes objetos para trocar mensagens, entretanto, não há nenhuma formalização acerca do funcionamento interno de cada objeto, o que é deixado a cargo de cada implementação.

Durante o processo de desenvolvimento dos sistemas TINA, entretanto, é interessante que se tenha a possibilidade de formalizar os aspectos comportamentais dos objetos (que não são contemplados pela linguagem ODL), principalmente quando se tratam de sistemas de maior complexidade, onde as relações entre os diversos objetos não são triviais. Neste contexto, a linguagem SDL [11] (*Specification and Definition Language*) entra como uma alternativa bastante interessante para o desenvolvimento de sistemas TINA, possibilitando uma formalização de seus aspectos comportamentais, o que além de permitir uma melhor análise de sua

[1] AT&T, Bellcore, France Telecom, MCI WorldCom, Sprint, Lucent, Nortel, Siemens, China Information Industry Ministry, etc

dinâmica, possibilita a realização de simulações com as especificações construídas e de sua validação.

Alguns trabalhos já foram realizados sobre o uso de SDL para o desenvolvimento orientado a objetos de serviços de telecomunicações baseados em TINA.Dentre estes trabalhos, pode-se citar o projeto TOSCA [12] (*TINA Open Service Creation Architecture*), desenvolvido pelo programa europeu de pesquisa ACTS (*Advanced Communications Technology and Services*), que concentrou seus esforços na criação de um *framework* para o desenvolvimento rápido de serviços de telecomunicações em SDL. Já no projeto SCREEN [13] (*Service Creation Engineering Environment*), também desenvolvido pelo programa de pesquisas ACTS, criou-se um ambiente para o desenvolvimento de serviços de telecomunicações voltados para a Internet, para CORBA e para TINA, utilizando SDL como linguagem de especificação formal. Ambos os projetos concentraram seus esforços na especificação formal da sessão de serviços TINA.

Neste contexto, esta dissertação propõe uma metodologia para a especificação formal orientada a objetos de sistemas TINA com o uso da linguagem SDL. A metodologia proposta se baseia naquelas apresentadas nos projetos TOSCA e SCREEN, porém algumas contribuições são realizadas no intuito de aprimorar alguns dos passos da metodologia, melhorando o resultado final do sistema que venha a ser desenvolvido. Além disso, diferentemente dos projetos citados, este trabalho se concentra na sessão de acesso TINA, propondo uma especificação formal para o modelo de gerência de subscrição de clientes. Este modelo consiste na parte do ambiente TINA responsável pelo gerenciamento e pela disponibilização das informações acerca dos usuários cadastrados no sistema e dos serviços contratados pelos mesmos. Conseqüentemente, também são especificados os objetos da sessão de acesso que fazem uso destas informações. Apesar do trabalho se concentrar na sessão de acesso, a metodologia proposta pode ser aplicada para o desenvolvimento de qualquer parte da arquitetura TINA.

A metodologia proposta consiste na combinação das técnicas de especificação formal em SDL e de geração de código em Java [14]. A partir das especificações em ODL dos objetos TINA (disponibilizadas nas recomendações do consórcio), desenvolveu-se um modelo equivalente em SDL. Para isso, uma série de mapeamentos para a conversão de estruturas ODL em estruturas SDL foi proposto. Estes mapeamentos foram baseados nos desenvolvidos pelo projeto TOSCA, entretando algumas contribuições foram realizadas visando facilitar o processo de geração de código, além de representar mais adequadamente alguns conceitos.

O modelo SDL gerado confere uma formalização comportamental bastante interessante ao sistema, permitindo a sua validação, bem como a simulação de suas principais funcionalidades

e casos críticos, o que auxilia na identificação e na correção de quaisquer erros existentes antes de uma eventual implementação.

Para o desenvolvimento das especificações em SDL, bem como para a realização dos processos de validação e simulação, foi utilizada a ferramenta CASE (*Computer Aided Software Engineering*) SDT [2] [15, 16] (*SDL Design Tool*) para o desenvolvimento profissional em SDL.

De posse do sistema em SDL, simulado e validado, é gerado código distribuído em Java para os objetos especificados. Para isso, foi desenvolvida uma ferramenta geradora de código Java que recebe como entrada qualquer arquivo com especificações em SDL para objetos TINA (desde que o sistema em SDL tenha sido especificado seguindo a metodologia proposta, ou seja, com o uso do mapeamento de ODL para SDL criado) e tem-se como resultado um sistema distribuído em Java que faz uso de CORBA como DPE.

Para a construção da ferramenta geradora de código Java foi utilizada a ferramenta construtora de compiladores JavaCC [17] (*Java Compiler Compiler*), que pode ser obtida gratuitamente na Internet. A plataforma CORBA utilizada foi o JavaIDL que é disponibilizado gratuitamente pela Sun como parte integrante das distribuições mais recentes do kit de desenvolvimento em Java (JDK - *Java Development Kit*).

A metodologia proposta pode reduzir significativamente o tempo dispendido no desenvolvimento de objetos TINA. Desta forma, empresas do ramo de telecomunicações podem se beneficiar das vantagens da formalização dos aspectos comportamentais do sistema TINA em um passo anterior à implementação, além de detectar e corrigir de erros através de simulações e validação das especificações. O processo de geração automático de código em Java também agiliza bastante o processo de implementação, o que é de fundamental importância no mercado competitivo dos dias atuais.

1.2 Descrição do Trabalho

Esta dissertação está dividida em seis capítulos e dois apêndices.

O **capítulo 2** traz uma introdução à arquitetura TINA e à sua Arquitetura de Serviços [18], com ênfase nos pontos mais relevantes para o trabalho desenvolvido. No final do capítulo é apresentada a metodologia de desenvolvimento a ser utilizada nos capítulos 3, 4 e 5.

O **capítulo 3** apresenta uma proposta de especificação formal orientada a objetos para o modelo de gerência de subscrição para ambientes TINA e para a sessão de acesso que

[2] Pacote SDT (*SDL Design Tool - Telelogic Suécia*) versão 4.2: adquirido pelo DT/FEEC/UNICAMP através do Projeto Temático FAPESP (Proc.91/3660-0).

faz uso desse modelo. São mostrados diagramas em SDL do sistema desenvolvido a partir das descrições ODL existentes, utilizando para isso as regras de mapeamento introduzidas no capítulo 2. É mostrada também a possibilidade de reuso das especificações através do conceito de herança. Com isso é possível simplificar o processo de desenvolvimento utilizando especificações já existentes como base para a construção de novos blocos SDL (especialização de tipos).

No **capítulo 4** são apresentadas os resultados da validação do sistema desenvolvido e exemplos de simulações das principais funcionalidades e casos críticos através de diagramas MSC [19] (*Message Sequence Chart*). Também são discutidas técnicas utilizadas para a identificação e correção de erros através da simulação e da validação.

O **capítulo 5** apresenta a ferramenta desenvolvida para a geração de código distribuído Java a partir das especificações SDL e os principais conceitos envolvidos. O código Java gerado faz uso de CORBA como plataforma distribuída de comunicação. São apresentados também exemplos de construções em SDL e seus equivalentes em Java.

No **capítulo 6** são apresentadas as conclusões finais e sugestões para eventuais trabalhos futuros.

O **apêndice A** apresenta uma introdução à linguagem SDL, mostrando suas principais características e os pontos relevantes para o trabalho desenvolvido. Também é apresentada a ferramenta CASE SDT, utilizada para o desenvolvimento das especificações em SDL.

No **apêndice B** são introduzidos conceitos de interpretação de código utilizados na construção da ferramenta geradora de código através do auxílio da ferramenta JavaCC.

O **apêndice C** apresenta os dois artigos publicados no período.

Capítulo 2

TINA e a Metodologia Utilizada

Neste capítulo são introduzidos conceitos relacionados à arquitetura TINA e é apresentada a metodologia proposta para o desenvolvimento de objetos TINA. Na seção 2.1 é apresentado um histórico do consórcio responsável pela definição da arquitetura. As seções 2.2, 2.3 e 2.4 apresentam uma introdução à arquitetura TINA, mostrando o seu funcionamento. A seção 2.5 apresenta com maiores detalhes a arquitetura de serviços TINA, onde se encontra o modelo de subscrição de clientes e a sessão de acesso que serão especificadas nos capítulos seguintes através do uso da metodologia proposta. Finalizando o capítulo, na seção 2.6 é proposta a metodologia para o desenvolvimento de objetos TINA através do uso combinado da linguagem de especificação formal SDL e da geração automática de código distribuído em Java, que faz uso de CORBA como plataforma de comunicação.

2.1 Histórico

No mundo de hoje, as operadoras de telecomunicações vêm enfrentando, cada vez mais, um ambiente de grande competição. Sendo assim, a habilidade de prover os serviços corretos no momento correto, de forma eficiente e pouco onerosa, tem se tornado vital para estas empresas. Com o constante surgimento de novas tecnologias, a cada instante pode-se prover novos serviços e pode-se alterar a forma de se prover estes serviços visando maior eficiência.

Nesse mundo de constantes inovações, as operadoras sentem cada vez mais a necessidade de oferecer novos serviços de forma rápida e de poder personalizá-los para cada cliente diferente de acordo com sua necessidade. Além disso, é notório o fato de que ainda hoje a especificação de novos serviços, o projeto desses serviços, a sua introdução na rede e as suas modificações, bem como a gerência do seu funcionamento são práticas que consomem muito

tempo, muitos recursos e nem sempre possuem a eficiência desejada.

Neste contexto, surge a proposição da arquitetura TINA [5, 6, 7, 8] (*Telecommunications Information Networking Architecture*), uma arquitetura aberta para o provimento e a gerência serviços de telecomunicações, que vem ao encontro das necessidades dos novos serviços e dos tradicionais e da necessidade de flexibilidade para utilização de novas tecnologias que venham a surgir.

A arquitetura TINA surgiu recentemente através do trabalho de um consórcio multinacional criado em 1992, o TINA-C [3, 4] (*Telecommunications Information Networking Architecture Consortium*).

Logo após o anúncio formal da criação do consórcio TINA (TINA-C) no ISS'92 (*International Switching Symposium*), engenheiros de mais de 30 empresas de grande porte iniciaram trabalhos conjuntos no projeto TINA. Desta forma, ao final de 1993 foi produzido o primeiro conjunto de recomendações da arquitetura TINA.

Em 1996, o TINA-C entrou em fase de consolidação dos resultados das colaborações obtidas até então e, ao final do ano de 1997, finalmente liberou um conjunto de recomendações para a arquitetura, como resultado dos 5 anos de colaborações. O seu trabalho, entretanto, ainda continua, com o objetivo de refinar as especificações da arquitetura através da resolução de algumas questões ainda em aberto.

2.2 Objetivos da Arquitetura TINA

A arquitetura TINA tem como objetivo resolver os problemas no que diz respeito ao provimento e à gerência de serviços de telecomunicações nos dias de hoje. Sendo assim, alguns objetivos mais diretos podem ser destacados, levando-se em conta algumas de suas características [20]:

- Interoperabilidade (de serviços de gerência, de aplicações e de plataforma) — o software de forma geral deve se comunicar com os demais, mesmo que tenham sido desenvolvidos por fabricantes distintos;

- **Reusabilidade (de especificações, de serviços e de aplicações)** — o software de forma geral deve ser reutilizado sempre que possível/necessário. A utilização de SDL para a especificação de objetos TINA é bastante útil neste ponto, por permitir o reuso de especificações construídas para o desenvolvimento de especializações que acrescentem novas funcionalidades, reduzindo assim o tempo de desenvolvimento de novos

objetos/aplicações e simplificando o processo de testes, uma vez que as especificações originais já se encontram testadas;

- Execução distribuída de aplicações — deve-se construir um ambiente distribuído, onde as aplicações/serviços são executados de forma transparente em qualquer local que estejam (a localização não é um fator significativo);

- Suporte a novos tipos de serviços — todos os tipos de serviço devem ser suportados, sejam baseados em chamadas (serviço telefônico) ou serviços de informação ou multimídia, serviços móveis (celulares) ou fixos, bem como a possibilidade de se criar qualquer outro tipo de serviço que venha a surgir;

- Suporte ao gerenciamento — deve haver gerência das versões dos serviços, gerência de dados replicados, gerência/monitoramento em tempo real, possibilidade de reconfiguração dinâmica (recuperação de falhas, upgrade de serviços, etc), gerência de contabilização e por último, deve haver uma coordenação entre a gerência dos serviços e dos recursos;

- **Independência do ambiente de computação e dos equipamentos** — as aplicações devem ser independentes dos recursos de rede/computação existentes. A especificação dos objetos através de uma linguagem de especificação com um alto nível de abstração (como é o caso do SDL) permite a total independência em relação aos recursos disponíveis. Mesmo que se altere o ambiente de execução, as especificações continuam válidas;

- Qualidade de serviço ou QoS (*Quality of Service*) — deve ser possível garantir diversos níveis QoS (dependendo do serviço desejado), bem como lidar com congestionamentos e indisponibilidade de recursos;

- Escalabilidade — a arquitetura deve acompanhar a evolução dos recursos disponíveis;

- Segurança — suporte a autenticação, realização de coleta e manutenção de dados para auditoria, proteção de informações, seleção de níveis de segurança;

- Compatibilidade com sistemas de telecomunicações existentes — deve permitir a troca de informações com os sistemas existentes (legados);

- Flexibilidade contra regulamentação — deve suportar, dentro do possível, as diversas regulamentações que possam existir ao redor do mundo na área de telecomunicações;

- **Testes de conformidade** — deve definir regras para a validação de soluções que venham a ser criadas, bem como de novos serviços. A linguagem de especificação formal SDL traz uma contribuição bastante grande no que diz respeito à validação das especificações criadas. Com o uso de ferramentas CASE profissionais para o desenvolvimento em SDL, como o SDT (utilizado neste trabalho), as especificações podem ser completamente validadas e ter suas principais funcionalidades e casos críticos simulados, evitando assim a presença de erros.

2.3 Estruturação do Ambiente TINA

O mercado de telecomunicações dos dias de hoje vem se mostrando bastante dinâmico, com mudanças aceleradas e novos serviços sendo oferecidos a cada dia que passa. Sendo assim, os serviços providos pela arquitetura TINA devem seguir essa dinâmica, devendo ser criados rapidamente. O mesmo deve ocorrer com os processos administrativos e de gerência.

Além disso, a complexidade do mercado vem aumentando ao longo do tempo. Existem diversos participantes no mundo das telecomunicações, os quais devem manter relações comerciais, visando uma possível interoperabilidade entre seus sistemas de comunicação. Essas relações de negócios (*business relationships*) entre os participantes devem, então, possuir a mesma dinâmica apresentada pelos serviços TINA. Visando suprir essa necessidade, o consórcio TINA elaborou um modelo de negócios (*business model*) que modela estas relações entre os participantes e que é capaz de lidar com toda essa dinâmica de mercado.

2.3.1 Relacionamento entre os participantes

A base para a operação de um ambiente TINA é a interação entre objetos "possuídos" pelos diversos participantes da rede. Sendo assim, o domínio destes participantes é muito bem delimitado e seus objetos se comunicam por pontos de referência [21] cujas características são definidas por um contrato entre as partes (figura 2.1).

Baseado nesse conceito de pontos de referência foi proposta uma estrutura composta de cinco funções que podem ser assumidas pelos participantes, baseando-se no mercado de telecomunicações e Internet existente, conforme pode ser visto no modelo de negócios apresentado na figura 2.2.

Um mesmo participante pode assumir mais de uma das funções do modelo de negócios (o provedor de serviços pode também atuar como varejista, por exemplo).

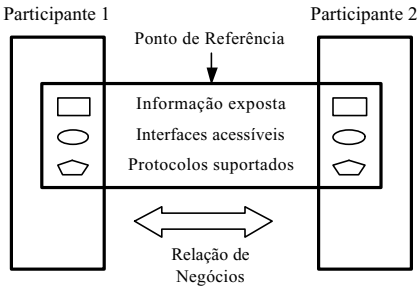


Figura 2.1: Relação entre participantes no modelo de negócios

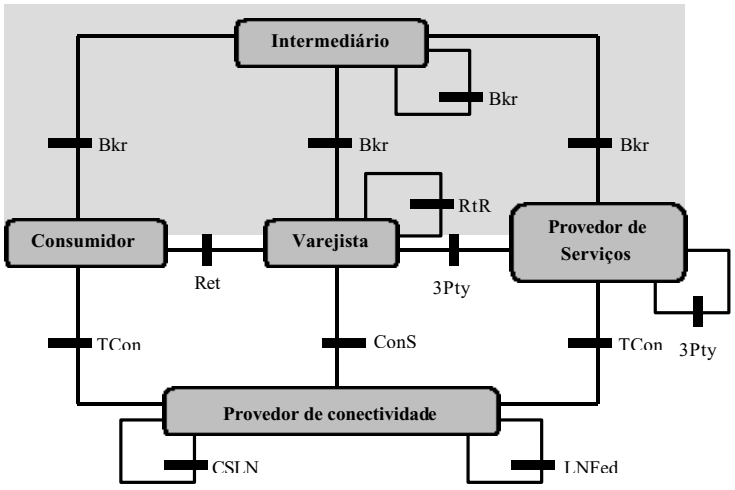


Figura 2.2: Modelo de Negócios

Os seguintes tipos de participantes foram identificados :

- Consumidor (*Consumer*) — talvez seja o participante mais importante de todo o conjunto, pois é o único que realmente consome os serviços TINA, sendo a sustentação econômica de toda a estrutura;

- Provedor dos serviços (*3Pty Service Provider*) - é o participante que efetivamente produz os serviços TINA, além de ser responsável pela manutenção dos mesmos;

- Varejista (*Retailer*) — é aquele que negocia diretamente com o consumidor a "venda" dos serviços que foram produzidos pelo provedor de serviços;

- Intermediário (*Broker*) — permite que os consumidores possuam acesso a informações sobre como e onde encontrar os serviços desejados;

- Provedor de conectividade (*Connectivity Provider*)- é responsável pelos recursos de transporte necessários para acesso aos serviços TINA.

As relações entre os diversos participantes podem ser de curta duração, quando um serviço é requerido e provido encerrando assim a relação comercial (por exemplo, a realização de uma ligação através de um orelhão), ou de longa duração (como a assinatura de uma linha telefônica residencial).

Essas relações são completamente descritas pelos pontos de referência mostrados na figura 2.2 e, dentre essas relações, três são consideradas de maior importância, são elas:

- Relação de varejista (*Ret*) — provê uma interface entre diferentes varejistas, permitindo a gerência e o acesso aos serviços destinados ao usuário final e gerenciando o ciclo de vida dos usuários;

- Relação de serviço de conectividade (*ConS*) — provê uma interface entre o provedor de conectividade e o varejista, possibilitando o controle e a gerência da conectividade (reservando e liberando recursos necessários);

- Relação de conexão terminal (*TCon*) — provê uma interface entre o provedor de conectividade e o consumidor ou o provedor de serviços (ambos, nesse caso chamados de consumidores de conectividade), possibilitando a gerência do acesso entre o provedor de conectividade e o consumidor dessa conectividade.

Sendo assim, o modelo de negócios TINA se mostra bastante útil na identificação das funções e pontos de referência de uma família de serviços de telecomunicações. É um ponto de partida a ser seguido no desenvolvimento de componentes para a arquitetura TINA.

2.3.2 Estruturação do software

TINA define uma arquitetura de software para sistemas de telecomunicações. Sendo assim, é definida uma estruturação básica do software bem como um ambiente de operação. Para isso, a arquitetura TINA faz uso de dois princípios básicos: sistemas distribuídos - uma vez que os serviços de telecomunicações são essencialmente distribuídos - e orientação a objetos - já que todos os serviços e os recursos de rede podem ser modelados como objetos que interagem uns com os outros.

Para a criação de um ambiente distribuído, foi definido um ambiente de processamento chamado DPE (*Distributed Processing Enviroment*). O DPE é responsável pela disponibilização do ambiente distribuído, sobre o qual as aplicações de telecomunicações irão funcionar. Desta forma a natureza não-distribuída e heterogênea das redes de telecomunicações é completamente escondida das aplicações de telecomunicações, que passam a "enxergar" um único grande ambiente distribuído - o DPE. O DPE da arquitetura TINA é o CORBA [22] (*Common Object Request Broker Architecture*) da OMG (*Object Management Group*).

A figura 2.3 mostra uma rede de telecomunicações e as diversas camadas necessárias para a construção da "superfície" DPE, que disponibiliza um ambiente de execução distribuído.

Em cada nó da rede de telecomunicações (representado pelo hardware na figura) há uma camada de software chamada NCCE (*Native Computing and Communications Enviroment*) que é a camada onde se encontra o sistema operacional do nó, bem como os sistemas de comunicação e outros sistemas de suporte.

Em cada nó, sobre esta camada, é implementado o DPE que se comunica com as implementações nos demais nós (utilizando os sistemas de comunicação da camada NCCE) criando uma "superfície" distribuída, sobre a qual irão funcionar os sistemas de telecomunicações da arquitetura TINA. Desta forma, os objetos de uma aplicação podem estar localizadas em diversos nós (de acordo com as necessidades operacionais) e o DPE passa a ser responsável pela localização dos objetos e interação entre os mesmos, permitindo que as aplicações sejam construídas sem que se necessite saber onde estarão localizadas as suas partes

Uma vez que os sistemas TINA são compostos por diversos objetos que se relacionam em um ambiente distribuído provido pelo DPE, estes objetos devem possuir interfaces padronizadas de forma a facilitar o relacionamento entre diferentes implementações. Visando essa padronização, TINA faz uso da linguagem ODL [10] (*Object Definition Language*) para a descrição formal dos objetos e de suas interfaces. A linguagem ODL é bastante similar à IDL [1] (*Interface Definition Language*), utilizada nos ambientes CORBA. Através dela é possível a descrição completa das interfaces dos objetos, padronizando assim a forma de comunicação entre os mesmos. Entretanto, o funcionamento interno do objeto não é descrito, deixando sua especificação a cargo de cada implementação.

Desta forma, os objetos TINA essenciais para o funcionamento da arquitetura possuem suas interfaces descritas com o uso de ODL e estas especificações estão contidas nas diversas recomendações que descrevem a arquitetura TINA.

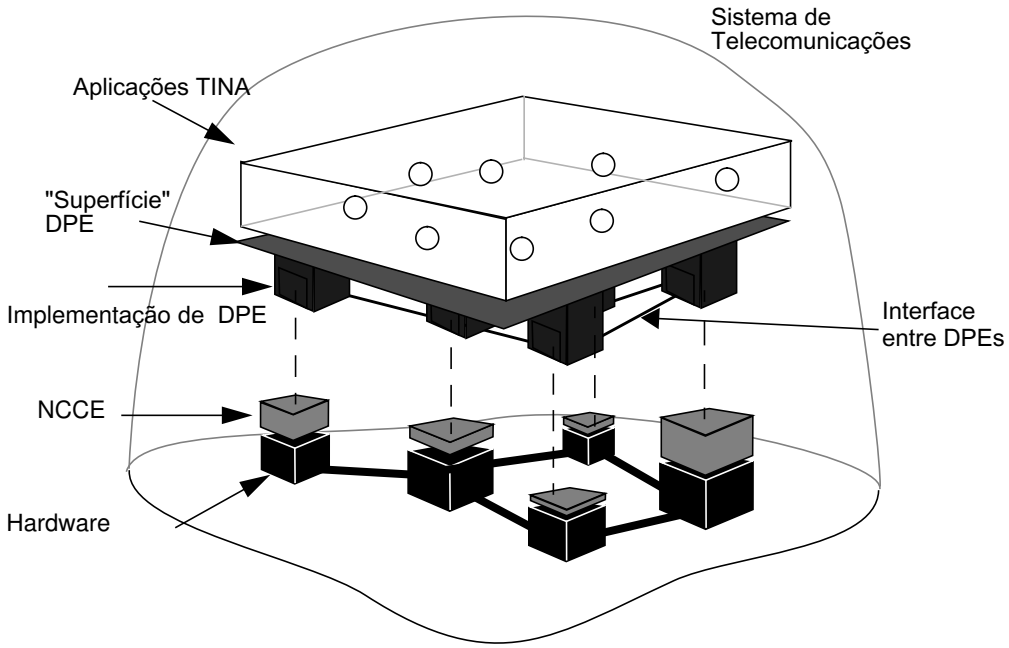


Figura 2.3: Disponibilização de um ambiente de execução distribuído através do DPE

2.4 Divisão de TINA em sub-arquiteturas

A arquitetura TINA é bastante ampla e é composta de um conjunto relativamente complexo de conceitos e princípios. A gama de serviços suportados é bastante extensa, incluindo serviços de transmissão de voz, serviços multimídia, serviços de informação e serviços de gerência. Novos tipos de serviço que possam surgir podem ser facilmente incorporados, bem como novas tecnologias de redes utilizadas para suportar estes serviços.

Um ponto importante a se destacar é o fato da arquitetura TINA não ter sido projetada para existir isoladamente. Sendo assim, os sistemas TINA devem possui a capacidade de se relacionar com sistemas não-TINA já existentes (sistemas legados).

Dessa maneira, visando dar um melhor tratamento a toda esta complexidade, é de fundamental importância a divisão da arquitetura em sub-conjuntos para a melhor compreensão da mesma. Cada um destes sub-conjuntos possui escopos muito bem definidos e se relaciona com os demais, dando forma à arquitetura TINA.

Sendo assim, TINA é dividida em 3 áreas técnicas, também chamadas arquiteturas. São elas:

- Arquitetura de Computação — define uma série de conceitos e princípios para a criação

de sistemas distribuídos e de um ambiente de suporte aos sistemas.

- Arquitetura de Serviços — define uma série de conceitos e princípios para a criação, implementação e gerência de serviços de telecomunicações;

- Arquitetura de Rede — define uma série de conceitos e princípios para a criação, implementação e gerência de redes de transporte.

Cada uma dessas arquiteturas pode ainda ser decomposta em outros sub-conjuntos mais detalhados de conceitos e princípios,conforme a figura 2.4.

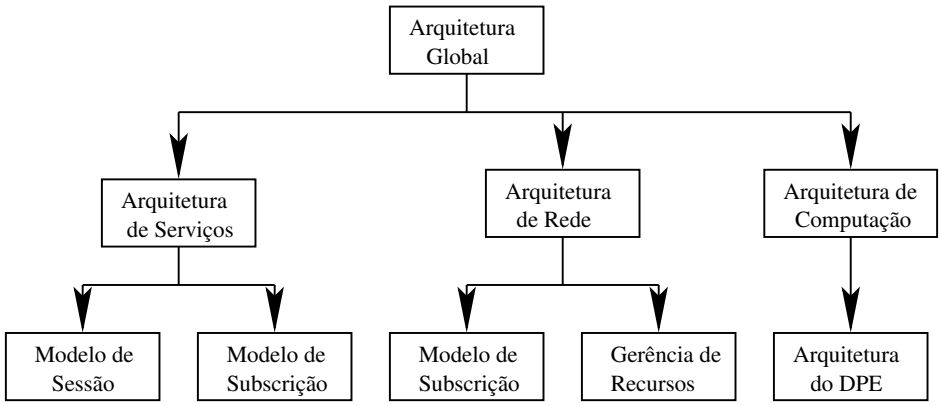


Figura 2.4: Decomposição da Arquitetura TINA

Essas arquiteturas se relacionam entre si de forma a garantir a consistência da arquitetura TINA. A arquitetura de serviços, por exemplo, requer a utilização de abstrações de recursos de rede que são fornecidas pela arquitetura de rede. A arquitetura de computação, por sua vez, serve como base para acesso aos serviços definidos na arquitetura de serviços. Sendo assim, como se pode perceber, apesar da arquitetura TINA ser dividida em sub-conjuntos, estes são altamente dependentes uns dos outros.

As arquiteturas de rede e de computação estão intimamente ligadas à disponibilização do meio de transporte necessário para a comunicação entre os objetos que proveêm os serviços TINA (objetos da arquitetura de serviços). Ou seja, estas duas arquiteturas estão relacionadas ao gerenciamento do DPE e de seus serviços básicos (ciclo de vida, persistência, etc). Diversos trabalhos relacionados a estas arquiteturas foram desenvolvidos ([23], [24], etc) e após a adoção de CORBA como DPE padrão da arquitetura TINA, muitos destes esforços foram deixados a cargo da OMG.

Já a arquitetura de serviços não teve seu desenvolvimento influenciado pela adoção de CORBA como DPE. Trata-se de uma arquitetura mais independente das questões de transporte e ainda demanda bastante esforço de desenvolvimento para que seja melhor estruturada. Com este intuito, diversos trabalhos vem tratando questões diretamente relacionadas a esta arquitetura ao longo dos últimos anos (questões de gerência de serviços, de contabilização de utilização, etc).

Dentre os trabalhos mais relacionados a esta dissertação, pode-se destacar os do projeto SCREEN [13] e do projeto TOSCA [12]. Ambos os trabalhos focalizam a sessão de serviço da arquitetura de serviços TINA, visando a criação de ambientes para o desenvolvimento de serviços de telecomunicações baseados na linguagem SDL. Estes projetos foram responsáveis pela especificação formal de objetos relacionados com a sessão de serviço.

O projeto TOSCA concentrou seus esforços no desenvolvimento de uma metodologia e um conjunto de ferramentas para a criação rápida e validação de serviços TINA. Para isso foi desenvolvido um mapeamento da linguagem ODL para SDL e, baseado nestes mapeamentos, foi implementada uma ferramenta para auxiliar a criação de serviços TINA através do uso de SDL [25, 26]. Esta ferramenta disponibiliza um *framework* com as características comuns a todos os serviços já especificadas, deixando em aberto apenas as partes específicas de cada serviço para que sejam desenvolvidas de acordo com as características do serviço a ser criado. Desta forma, visa-se facilitar o trabalho de desenvolvimento de novos serviços, uma vez que grande parte do serviço já se encontra pronto e pode ser utilizado em todos os novos serviços a serem criados.

O projeto SCREEN, por sua vez, foi responsável pelo desenvolvimento de especificações formais para a sessão de serviço TINA (tal qual o projeto TOSCA) além de serviços baseados em CORBA para a Internet. Em uma etapa complementar, foi desenvolvido um gerador de código capaz de criar código Java com base nas especificações formais em SDL desenvolvidas pelo projeto em questão. Entretanto, o código gerado por esta ferramenta necessita de algumas adaptações manuais para que funcione corretamente [27].

Visando complementar os estudos relacionados à Arquitetura de Serviços TINA, esta dissertação se concentra em outras parte da sessão de serviço TINA não exploradas em nenhum dos trabalhos anteriores — o modelo de subscrição de serviços e a sessão de acesso que faz uso deste modelo. O modelo de subscrição possui um papel bastante importante na gerência das informações dos assinantes e dos serviços contratados e, além disso, ainda possui muitos pontos em aberto que necessitam de um maior aprofundamento.

Sendo assim, a próxima seção apresenta a arquitetura de serviços e o papel da sessão de

acesso dentro da mesma, bem como o funcionamento do modelo de subscrição de serviços. Em seguida, na seção 2.6 é apresentada a metodologia proposta para o desenvolvimento de objetos TINA utilizando as técnicas de especificação formal SDL e de geração automática de código Java, a qual será utilizada para o desenvolvimento dos objetos do modelo de subscrição e da sessão de acesso nos capítulos 3, 4 e 5.

2.5 A Arquitetura de Serviços

A arquitetura de serviços [18, 28] é responsável por definir um conjunto de conceitos e princípios que possibilitem a concepção, implementação, uso e operação de serviços de telecomunicações. Para isto, esta arquitetura define um conjunto de componentes reusáveis com os quais se constrói os serviços de telecomunicações.

Trata-se da arquitetura mais nova e talvez a mais importante das especificações TINA. Seu escopo é delimitado pela parte cinza da figura 2.2.

Os serviços em TINA podem ser divididos em 3 grupos distintos:

- Serviços de Telecomunicações — são basicamente os serviços responsáveis pelo transporte de informações entre dois pontos da rede de telecomunicações, através do estabelecimento de conexões;
- Serviços de Gerência — são responsáveis pelo gerenciamento dos recursos no ambiente TINA. Incluem as funcionalidades de gerência de falha, segurança, configuração, desempenho e contabilização além de controle de ciclo de vida de serviços e usuários;
- Serviços de Informação — são os serviços que processam informações, como vídeo, áudio ou documentos. São responsáveis pelo armazenamento e manipulação destas informações.

A arquitetura de serviços provê, desta forma, mecanismos para garantir o acesso e o uso destes serviços. Para que isso seja possível, são definidos alguns conceitos que servem de base para a definição de toda a estrutura da arquitetura de serviços, são eles:

- Conceitos de Sessão
- Conceitos de Acesso
- Conceitos de Gerência

2.5.1 Conceitos de Sessão

O conceito de sessão está intimamente ligado ao tempo. Uma sessão nada mais é que um período de tempo no qual um conjunto de atividades é executado visando atingir algum objetivo. Por ser um conceito bastante amplo, pode ser aplicado a diversos casos e sendo assim, 4 diferentes tipos de sessão podem ser identificados na arquitetura de serviços: sessão de serviço, sessão de usuário, sessão de comunicação e sessão de acesso.

A **sessão de serviço** é a simples ativação do serviço. Ela relaciona os usuários do serviço para que os mesmos possam interagir uns com os outros, trocando informações. Computacionalmente, uma sessão de serviço é representada por um gerente de sessão de serviço (figura 2.5). Este gerente é um objeto que oferece dois tipos de interfaces operacionais: uma que diz respeito ao controle da sessão (entrada e saída de usuários, etc) e outra que diz respeito à operação do serviço.

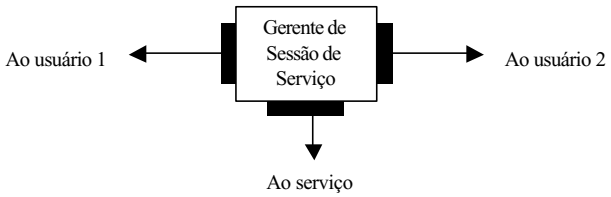


Figura 2.5: Interfaces do Gerente de Sessão de Serviço

Em alguns casos é interessante que o gerente de sessão de serviço ofereça a possibilidade de se suspender/retomar o envolvimento dos usuários com o serviço. Desta forma, alguns serviços (como vídeo-conferência) que sejam utilizados por mais de um dia podem liberar recursos de comunicação durante a noite (quando o serviço não é utilizado) armazenando no gerente o estado atual do serviço (usuários envolvidos, recursos utilizados, etc). Desta forma, o serviço pode ser facilmente retomado no dia seguinte sem que haja a necessidade de recriá-lo.

A **sessão de usuário**, por sua vez, armazena informações acerca de suas atividades e dos recursos alocados para a utilização de uma sessão de serviço. A sessão de usuário é criada sempre que um usuário se liga a uma sessão de serviço - armazenando dados como a tarifação do usuário, estado do serviço (página atual sendo editada pelo usuário), etc - e é removida quando o usuário deixa a sessão. É representada computacionalmente pelo gerente de sessão de usuário (figura 2.6)

A **sessão de comunicação** é uma abstração orientada aos serviços das conexões da rede de transporte, sendo necessária apenas quando há uma interface de fluxo de dados (stream).

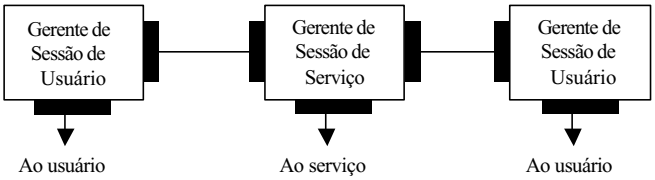


Figura 2.6: Relação entre a sessão de serviço e a sessão de usuário

Desta forma, mantém informações sobre as conexões de uma determinada sessão de serviço (como qualidade de serviço, caminhos de comunicação, etc). Computacionalmente, uma sessão de comunicação é representada por um gerente de sessão de comunicação (figura 2.7).

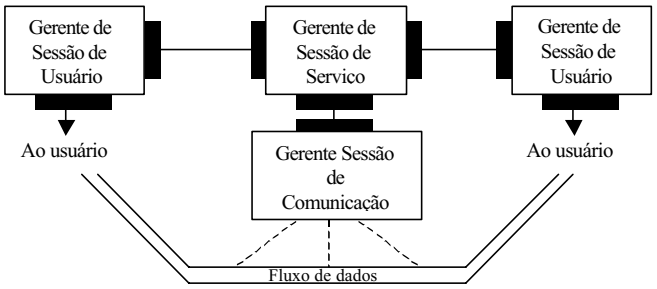


Figura 2.7: Gerente de sessão de comunicação e o fluxo de dados

Por último, **a sessão de acesso** mantém informações acerca da ligação do usuário com o sistema e o seu envolvimento com os serviços. É responsável pelo processo de autenticação do usuário e de disponibilização de informações essenciais para o acesso do usuário, como a lista de serviços disponíveis por exemplo. Computacionalmente é representada por uma série de objetos que juntos possibilitam o acesso do usuário ao ambiente TINA.

A definição destes conceitos distintos de sessão concede à arquitetura TINA uma maior flexibilidade na disponibilização dos seus serviços. A separação das sessões de serviço e de usuário, por exemplo, permite a distribuição de funções e estados, onde a sessão de usuário provê uma visão local enquanto a sessão de serviço provê uma visão do conjunto. Além disso tal separação permite a suspensão e restauração da ligação do usuário com o serviço. Já a separação entre as sessões de comunicação e de serviço permite que sejam criados serviços que usem ou não a rede de transporte (para fluxo de dados, como áudio ou vídeo). Para os serviços que utilizem este recurso é criado um gerente de sessão de comunicação, já para os serviços que não utilizem, não é necessária a criação do mesmo, utilizando-se apenas das interfaces operacionais dos objetos envolvidos.

2.5.2 Conceitos de Acesso

Visando permitir que os usuários tenham um acesso mais flexível aos serviços, ou seja, que possam acessar os serviços de diferentes localizações e a partir de diferentes terminais, o acesso do usuário é distingüido do acesso do terminal na arquitetura de serviços. Sendo assim, existem dois agentes (objetos computacionais) que se relacionam com os serviços: agente de usuário e agente de terminal (figura 2.8).

O agente de usuário é o objeto computacional que representa o usuário em si. Ele recebe requisições do usuário para estabelecer sessões de serviço ou se juntar a sessões já existentes. Da mesma forma, o agente de usuário é que recebe as requisições para se juntar às sessões de serviço provenientes das sessões de serviço propriamente ditas.

Já o agente de terminal é o objeto computacional que representa o ponto a partir do qual o usuário está acessando o sistema TINA. É responsável por obter a localização precisa do terminal.

Para acessar um serviço, o agente de usuário deve se associar a um agente de terminal, completando o processo de *login*.

Dependendo do tipo de serviço, um usuário pode estar associado a mais de um agente de terminal. No caso de uma vídeo-conferência, cada usuário pode estar conectado a uma estação de trabalho e a um telefone (um usuário, dois terminais). Da mesma forma, um agente de terminal pode estar associado a diversos usuários. No caso de uma reunião, todos os participantes podem associar seus agentes de usuário ao telefone da sala de reuniões.

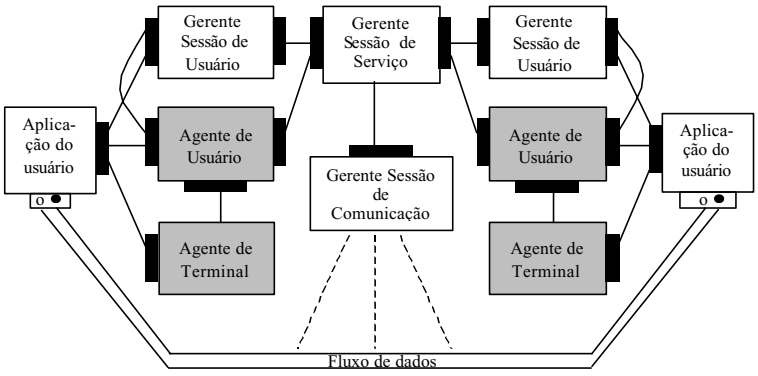


Figura 2.8: Componentes de acesso e de sessão

2.5.3 Componentes da Arquitetura de Serviços

Inicialmente deve-se diferenciar os objetos utilizados na construção de uma arquitetura de serviços, separando-os em 3 conjuntos, conforme mostrado pela tabela 2.1: os responsáveis pelo acesso ao serviços, os responsáveis pelo uso do serviço propriamente dito e os responsáveis pelo estabelecimento das sessões de comunicação (fluxos de dados, por exemplo).

Categoria	**Componente (Objeto)**	**Sigla**
Sessão de Acesso	*User Application*	as-UAP
	Provider Agent	PA
	Initial Agent	IA
	User Agent	UA
	Named User Agent	namedUA
	Anonymous User Agent	anonUA
Sessão de Serviço	*User Application*	ss-UAP
	Service Factory	SF
	User Session Manager	USM
	Service Session Manager	SSM
Sessão de Comunicação	*Terminal Communication Session Manager*	TCSM
	Communication Session Manager	CSM

Tabela 2.1: Objetos da arquitetura de serviços

Um exemplo de como estes componentes (ou objetos) podem interagir pode ser visto na figura 2.9. Ela ilustra um caso simplificado onde dois usuários utilizam um serviço do domínio do provedor - o qual atua tanto como provedor de acesso como provedor de uso do serviço.

Os componentes relacionados com o acesso provêem acesso seguro e personalizado aos serviços além de possibilitar a mobilidade do usuário. O cenário que descreve o estabelecimento de uma sessão de acesso pode ser observado na figura 2.10. A aplicação de Usuário relacionada à Sessão de Acesso (as-UAP - *Access Session related User Application*), provê uma inteface ao usuário para que este possa interagir com o provedor do serviço. Para isso o as-UAP se comunica com o representate do provedor no domínio do usuário, o Agente do Provedor (PA - *Provider Agent*), repassando as requisições do usuário (passo 1). O PA se comunica então com os objetos no domínio do Provedor. O contato inicial (passo 2) é realizado com o Agente Inicial (IA - *Initial Agent*). Após o processo de autenticação do usuário junto ao provedor, o IA informa ao PA a localização do Agente do Usuário (UA - *User Agent*) com o qual ele deve se comunicar (passo 3), estabelecendo assim uma sessão de acesso. Em seguida PA informa seu contexto para UA (passo 4), ou seja informações acerca do terminal

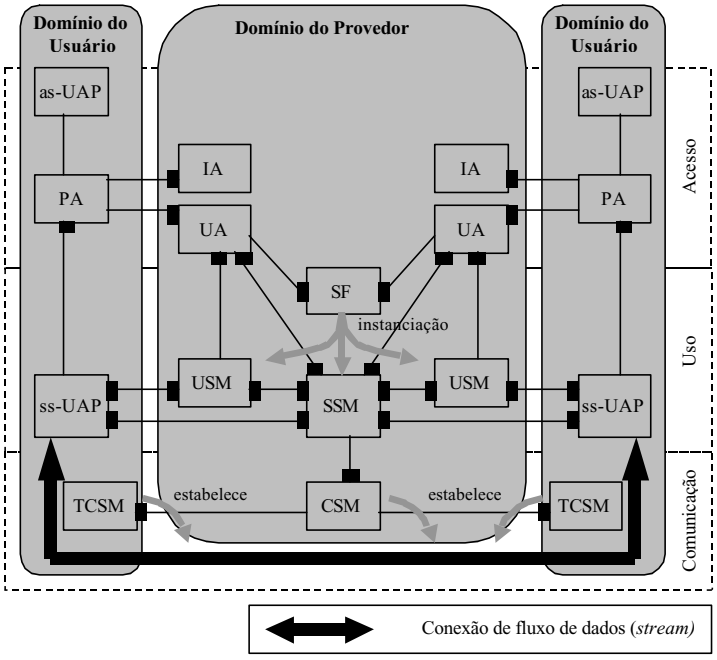


Figura 2.9: Exemplo de interação entre os componentes de serviço

de acesso, referências para suas interfaces, etc. A partir daí o usuário pode realizar diversas tarefas (como obter a lista de serviços contratados, por exemplo) ou mesmo fazer uso dos serviços disponibilizados.

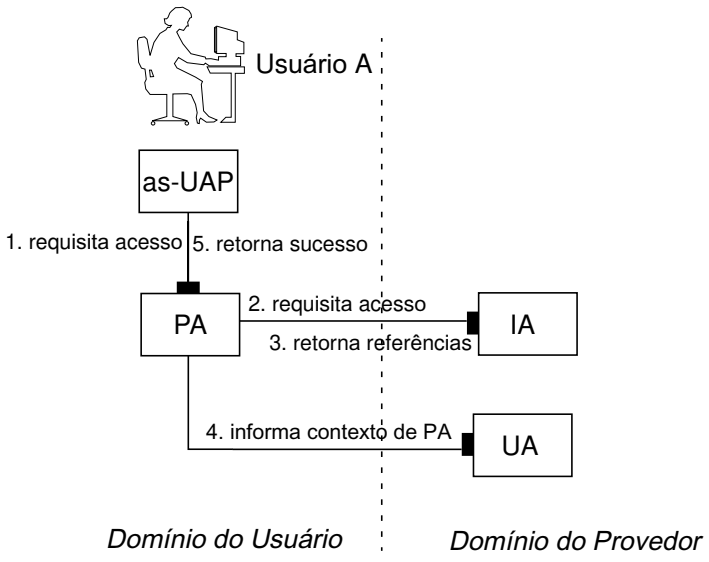


Figura 2.10: Cenário que descreve o estabelecimento de uma sessão de acesso [18]

A tabela 2.1 mostra que o UA possui duas especializações - o Agente de Usuário Anônimo (anonUA - *Anonymous User Agent*) e o Agente de Usuário Identificado (namedUA - *Named User Agent*). Isso se deve ao fato do usuário poder se identificar ao provedor ou preferir se manter anônimo. O anonimato, entretanto, pode não ser tolerado em determinados serviços, ficando a cargo de cada um deles aceitar ou não usuários anônimos. Entretanto, os provedores podem fazer uso do acesso anônimo para disponibilizar aos usuários uma forma de cadastramento *online*, ou seja, o usuário que ainda não é cliente do provedor se conecta anonimamente e se cadastra diretamente no provedor. A partir daí já pode acessar normalmente, identificando-se com os dados cadastrados.

Os componentes relacionados com a sessão de serviço (uso do serviço), são responsáveis por disponibilizar serviços que possam ser acessados e gerenciados através de diversos domínios. Neste nível, têm-se o Gerente de Sessão de Usuário (USM - *User Session Manager*) e o Gerente de Sessão de Serviço (SSM - *Service Session Manager*) que permitem o controle de uma sessão de serviço; o USM controla as partes que dizem respeito a um certo usuário enquanto o SSM controla as partes relativas ao serviço como um todo (comuns a todos os usuários). Instâncias de ambos são criadas pela Fábrica de Serviços (SF - *Service Factory*), de acordo com solicitações dos UAs. Por último, no domínio do usuário, tem-se a Aplicação de Usuário relacionada à Sessão de Serviço (ss-UAP - *Service Session User Application*), que permite que o usuário interaja com a sessão de serviço.

Finalmente, os componentes relacionados com a sessão de comunicação são responsáveis por prover uma comunicação fim-a-fim. O Gerente de Sessão de Comunicação (CSM - *Communication Session Manager*) e o Gerente de Sessão de Comunicação de Terminal (TCSM - *Terminal Communication Session Manager*) estabelecem uma conexão de fluxo de dados entre os UAPs dos usuários.

2.5.4 O modelo de Subscrição

Para que seja possível o acesso dos usuários aos serviços disponibilizados em um ambiente TINA, é necessário que haja uma forma de gerenciamento dos dados destes usuários e dos serviços assinados pelos mesmos. Visando suprir esta necessidade, o consórcio TINA propôs um modelo de gerenciamento da subscrição de serviços.

A subscrição é baseada em um modelo de negócios proposto, o qual explicita as relações existentes entre os participantes do processo de provimento de serviços (consumidor ou assinante e varejista ou *retailer*). A figura 2.11 mostra a relação existente entre estas entidades.

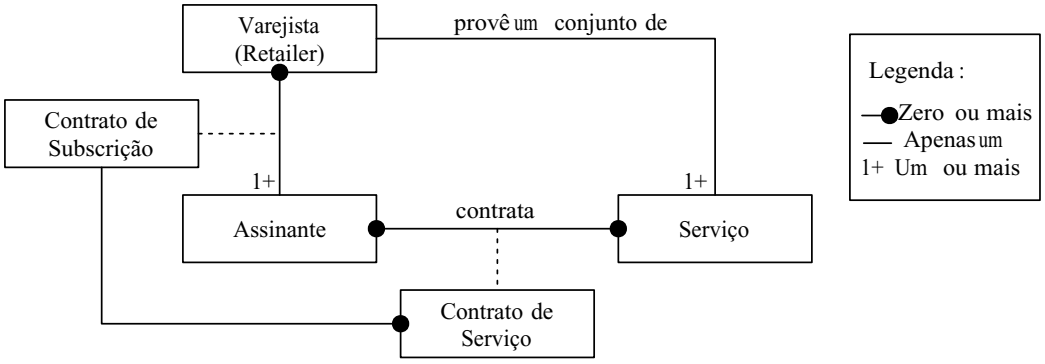


Figura 2.11: Modelo de Negócios da Subscrição

Um assinante pode realizar contratos de subscrição com diversos varejistas. Estes contratos possuem informações independentes dos serviços, como forma de pagamento, pessoa de contato e informações sobre o assinante. Após assinar um contrato de subscrição, pode-se assinar um ou mais serviços dentre os disponibilizados pelo varejista. Para cada assinatura é gerado um contrato de serviço, o qual contém informações relativas ao serviço em questão, como condições e características do serviço (perfis de serviço) que são acordadas entre o assinante e o varejista.

No contexto do modelo de subscrição há uma diferenciação entre assinante e usuário. O assinante é o papel daquele que compra serviços do varejista enquanto o usuário é o papel exercido por aquele que fará uso destes serviços. Desta forma, um assinante pode estar associado a diversos usuários. Por exemplo, uma empresa exerce o papel de assinante ao contratar serviços para que seus empregados, que são os usuários, façam uso. Este relacionamento entre o assinante, o usuário e os serviços pode ser compreendido através da figura 2.12.

Podem então ser identificadas algumas das principais entidades deste modelo:

Assinante (*Subscriber*): representa a pessoa física ou jurídica que assina o contrato com o varejista para que seus usuários possam fazer uso de uma série de serviços;

Usuário: representa a entidade que efetivamente fará uso dos serviços;

Grupo de Atribuição de Assinante (*SAG - Subscription Assignment Group*): representa um grupo de usuários aos quais são atribuídos o mesmo perfil de serviço;

Serviço (*Service*): descreve um serviço em particular através de um formato de especificação padrão;

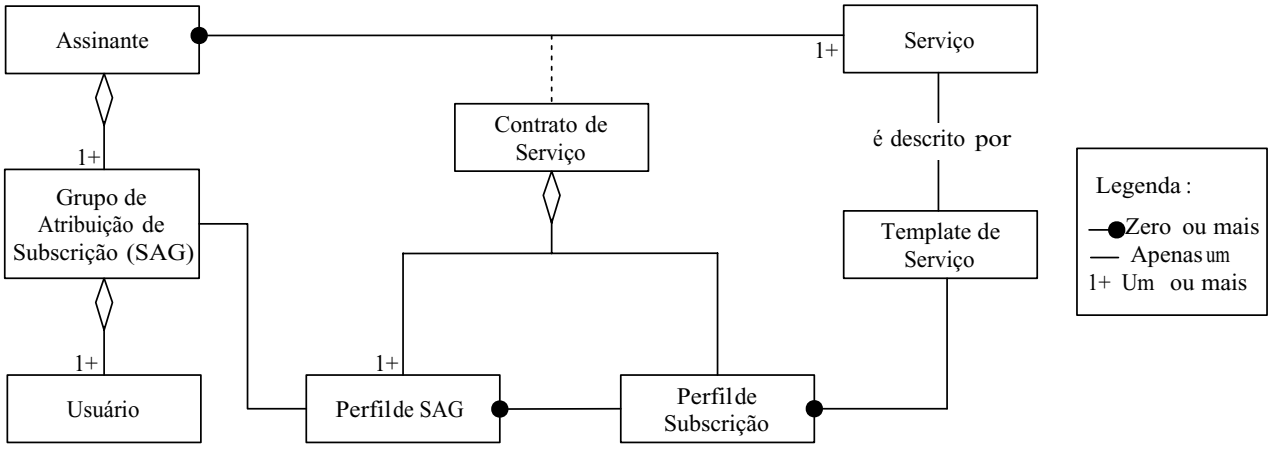


Figura 2.12: Modelo de Informação da Subscrição

Template de Serviço (*Service Template*): representa algumas características genéricas de um determinado serviço;

Perfil de Subscrição (*Subscription Profile*): representa algumas características do serviço desejadas pelo assinante;

Perfil de Serviço do SAG (*SAG Service Profile*): representa uma maior "personalização" do serviço de acordo com as necessidades de um grupo de atribuição de assinante;

Contrato de Serviço (*Service Contract*): especifica as condições de uso do serviço em questão. Indica o perfil de subscrição e os perfis de serviço dos SAGs que permitem uma melhor adequação do serviço de acordo com o uso desejado;

Pelo modelo descrito acima, pode-se então verificar que um assinante contrata um ou mais serviços. Para cada assinatura de serviço há um contrato de serviço que especifica as configurações desejadas. O template do serviço descreve as características de cada serviço disponibilizado. Cada assinante pode ainda selecionar parâmetros específicos do serviço contratado (através do perfil de subscrição), os quais serão disponibilizados para todos os seus usuários. O assinante ainda pode atribuir parâmetros específicos do serviço a grupos de usuários através da criação de SAGs que englobam um conjunto de usuários e possuem perfis de serviço do SAG associados.

Com o intuito de gerenciar estas informações e disponibilizá-las para os diversos objetos da arquitetura de serviços TINA, um modelo computacional da gerência de subscrição é exibido na figura 2.13.

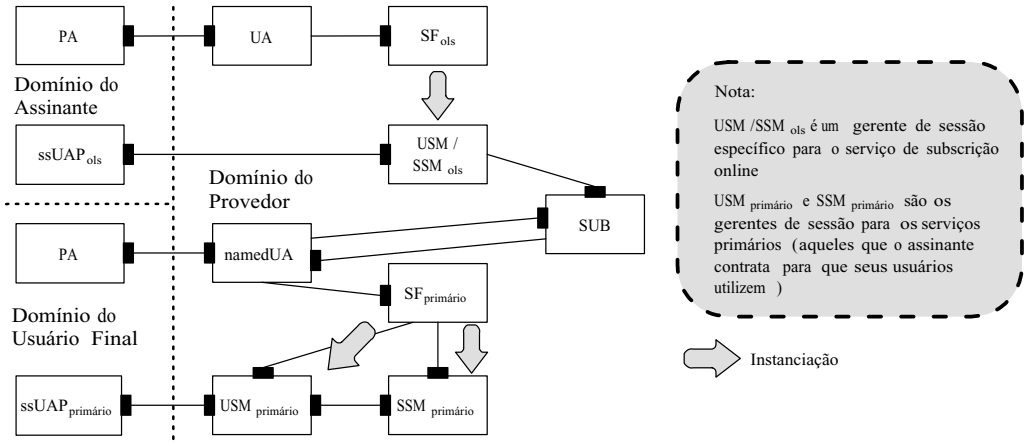


Figura 2.13: Modelo Computacional da Subscrição

Nesta figura pode-se identificar a presença do objeto responsável pelo gerenciamento das informações relativas à subscrição de serviços — o objeto SUB. Suas funcionalidades incluem a criação, modificação e exclusão de assinantes, informações relacionadas a assinantes, usuários, contratos de serviços e perfis de serviço e subscrição. O objeto SUB também é responsável por prover informações acerca dos assinantes, dos usuários e dos serviços assinados pelos mesmos para os demais objetos, como o UA por exemplo. O UA faz uso das informações fornecidas pelo SUB para autenticar os usuários, bem como para prover de forma adequada o acesso aos serviços contratados.

Ainda pode-se notar a presença de três novos objetos, o $ssUAP_{ols}$ (no domínio do consumidor), o USM/SSM_{ols} e o SF_{ols} (no domínio do provedor). Tratam-se de objetos auxiliares específicos para o acesso aos serviços de subscrição. Ou seja, auxiliam o acesso ao SUB para que sejam efetuados as interações relativas à contratação de novos serviços, alteração de contratos existentes ou cancelamento destes contratos.

Além disso, os objetos $SF_{primario}$, $USM_{primario}$ e $SSM_{primario}$ são os objetos relacionados com o provimento dos serviços primários, ou seja, aqueles serviços que efetivamente serão utilizados pelos usuários. Suas funcionalidades já foram descritas anteriormente.

A figura 2.14 descreve o cenário que exemplifica o processo de contratação de um novo serviço através da utilização do serviço de subscrição. Uma vez que o serviço de subscrição tenha sido ativado, ou seja, todos os seus objetos tenham sido criados pela fábrica de serviços (SF_{ols}), o usuário solicita a contratação de um novo serviço através de uma interface (gráfica ou textual) disponibilizada pelo objeto $ssUAP_{ols}$. Este objeto, por sua vez, envia a solicitação de criação (passo 1) para o gerente de sessão do serviço de subscrição (USM/SSM_{ols}) no

domínio do provedor. Este objeto gerencia a sessão de serviço criada e, portanto, possui referências para às interfaces desejadas do objeto SUB. A solicitação de contratação de um novo serviço é então repassada ao objeto SUB através da interface adequada (passo 2) e este objeto se encarrega de efetivar a contratação do serviço gravando estas informações na base de dados de subscrição. Após a contratação do serviço desejado, USM/SSM_{ols} ainda requisita o template do serviço recém-contratado (passo 3), visando assim oferecer maiores informações para o usuário solicitante. Finalmente uma mensagem de sucesso é retornada ao objeto $ssUAP_{ols}$ (passo 4) informando ao usuário que a operação foi concluída.

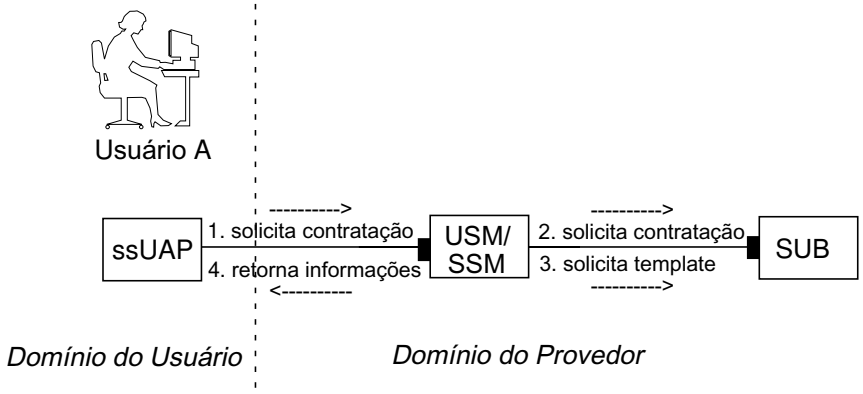


Figura 2.14: Cenário que descreve a contratação de um novo serviço

2.6 Metodologia para o desenvolvimento de objetos TINA em SDL com geração de código Java

Este trabalho propõe a utilização de uma metodologia que combina as técnicas de especificação formal em SDL com a geração de código em Java para o desenvolvimento de objetos TINA que se comunicam através de um ambiente distribuído baseado em CORBA.

A idéia básica desta metodologia (figura 2.15) é utilizar a técnica de especificação formal em SDL em uma fase inicial do desenvolvimento visando prover uma formalização comportamental aos objetos TINA (capítulo 3), o que não é possível com a utilização da linguagem ODL (com a qual os objetos são descritos nas recomendações do TINA-C). É possível, então, realizar a validação do sistema especificado e simular suas principais funcionalidades e casos críticos, com o intuito de detectar e corrigir os erros existentes (capítulo 4).

Em uma fase posterior, é gerado código distribuído Java a partir das especificações SDL desenvolvidas (capítulo 5). Este código Java faz uso da arquitetura CORBA como plataforma

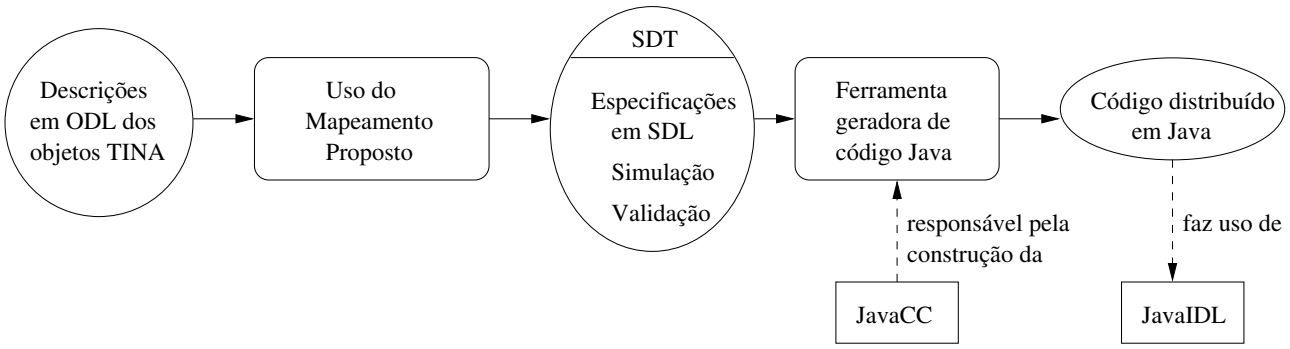


Figura 2.15: Metodologia combinando o uso de SDL e geração de código em Java

distribuída de comunicação entre os objetos e, mais especificamente faz uso do JavaIDL, que é uma implementação da plataforma CORBA distribuída gratuitamente juntamente com as versões mais recentes do kit de desenvolvimento Java (JDK - *Java Development Kit*).

Para o processo de geração automática de código, foi desenvolvida uma ferramenta para a conversão das especificações em SDL para código em Java com o auxílio do JavaCC (*Java Compiler Compiler*), que é um programa que ajuda na construção de ferramentas interpretadoras de código, ou seja, de compiladores.

Como decorrência desta metodologia, têm-se grandes vantagens decorrentes do uso da linguagem SDL (formalização comportamental, possibilidade de validação e simulação do sistema) e da geração automática de código em Java (agilização do processo de implementação, redução na incidência de erros devido à automatização do processo e portabilidade do sistema devido ao uso da linguagem Java).

Para a automação desta metodologia, utilizou-se o SDT (*SDL Design Tool*), uma ferramenta CASE composta por módulos responsáveis por:

- especificar sistemas SDL;
- simular, verificar e validar as especificações;
- editar e gerar diagramas MSC como resultado de simulações.

O uso do ambiente SDT (apresentado no apêndice A) e do JavaCC (apresentado no apêndice B) tornaram o trabalho de especificação e de geração automática de código, respectivamenta, atividades mais simples e eficientes.

2.6.1 A linguagem ODL

A linguagem ODL [10] (*Object Definition Language*) foi definida pela consórcio TINA como uma extensão da linguagem IDL (*Interface Definition Language*), definida pela OMG (*Object Management Group*) e utilizada para a definição de objetos CORBA. O ODL suporta algumas funcionalidades não cobertas pelo IDL, dentre elas pode-se destacar: objetos com múltiplas interfaces, interfaces de fluxo de dados e descrições de qualidade de serviço.

A linguagem ODL é composta de cinco partes principais: tipos de dados e constantes, interfaces de fluxo de dados, interfaces operacionais, objetos e grupos de objetos. Os tipos de dados e constantes podem estar contidos em interfaces (operacionais e de fluxo de dados), que por sua vez podem estar contidas em objetos, os quais podem compor grupos de objetos.

A figura 2.16 mostra a descrição em ODL dos diversos componentes da linguagem. É definido um grupo *G1* composto pelos objetos *O1* e *O2*. O objeto *O1* suporta as interfaces *I1* e *I2* enquanto o objeto *O2* suporta as interfaces *I1* e *I3*. A interface *I1* é uma interface operacional e disponibiliza os métodos *f1* e *f2*. A interface *I2* é uma interface de fluxo de dados, sendo assim, define o fluxo de dados do tipo *VoiceFlowType*. Além disso, é capaz de receber um fluxo deste tipo através da entrada *voiceUpStream* e pode enviar fluxos deste tipo pela saída *voiceDownStream*. Por último, a interface I3 é uma interface operacional e define o tipo de dados *DataType1* que é utilizado como parâmetro de entrada para o método *f3*.

Além disso, a linguagem ODL permite o uso do conceito de herança. Assim sendo, podem ser construídas interfaces, objetos ou grupos de objetos que sejam especializações de outros. Ou seja, que herdem todas as suas características e acrescentem novas características particulares da interface, objeto ou grupo criado.

Os parâmetros passados para os métodos definidos em ODL podem ser de três tipos:

- **in**: Parâmetro de entrada. Utilizado para a passagem de valores para o método chamado. Seu valor não é alterado pelo método.

- **out**: Parâmetro de saída. Utilizado para como forma de retorno de valores. Seu valor é alterado pelo método.

- **inout**: Paramêtro de entrada/saída. Utilizado ao mesmo tempo para a passagem de valor para o método e para retorno. Seu valor pode ser alterado pelo método.

A linguagem ODL se preocupa basicamente com a formalização das interfaces dos objetos componentes de um sistema. Ou seja, da forma através da qual eles se comunicam, provendo

```
group G1 {
    ...
    components
        O1, O2;
}; // end of Group G1

object   O1 {
    ...
    supports
        I1, I2;
}; // end of Object O1

object   O2 {
     ...
     supports
          I1,I3;
}; // end of Object O2
```

```
interface   I1 {
    ...
    void f1(in long a);
    boolean f2(in string a, out string b);
} // end of Interface I1;

interface   I2 {
    ...
    // Flow types
    typedef  ... VoiceFlowType;

    source    VoiceFlowType voiceDownStream;
    sink   VoiceFlowType voiceUpStream;
}; // end of Interface I2;

interface   I3 {
    ...
    // Data types
    typedef  ... DataType1;

    void f3(in DataType1 a);
}; // end of Interface I3;
```

Figura 2.16: Exemplos de especificações em ODL

assim um meio de padronização da comunicação dos objetos. Não há preocupação com o aspecto comportamental do objeto. Seu funcionamento interno é deixado sob responsabilidade da implementação. Por isso, para o desenvolvimento de uma implementação destes objetos, surge a necessidade da utilização de uma linguagem de especificação formal que seja capaz de prover um formalismo comportamental — o SDL, no caso deste trabalho.

2.6.2 A linguagem SDL e a ferramenta SDT

A linguagem de especificação formal SDL [11] (*Specification and Description Language*) teve seu desenvolvimento iniciado pela CCITT, hoje ITU-T, em 1972, visando a especificação e descrição de sistemas telefônicos, sendo depois utilizada também para a especificação de protocolos em redes de telecomunicações. Atualmente sua aplicação é muito mais abrangente, sendo utilizada na especificação de sistemas de informação [29, 30], sistemas distribuídos [31, 27], sistemas de tempo real, etc. Trata-se de uma linguagem que incorpora conceitos de orientação a objetos, tais como tipos, reuso e herança.

A linguagem SDL é estruturada em diversos níveis de abstração, desde um nível mais alto até um nível mais detalhado do sistema especificado. Um sistema é dividido em blocos, que se comunicam uns com os outros e com o ambiente. Cada bloco é composto por um conjuto de processos que também se comunicam entre si e até mesmo com outros blocos. O processo, por sua vez, trata-se do nível de abstração onde é especificado o comportamento do sistema através da definição de estados e transições. Sempre que necessário, podem ser especificados procedimentos (dentro dos processos) que realizem alguma tarefa específica.

Os processos se comunicam entre si de duas formas diferentes: através da troca de sinais (modo assíncrono) ou através da chamada a procedimentos exportados (modo síncrono). Através destes sinais e procedimentos exportados, os processos podem trocar dados e trabalhar cooperativamente, dando a funcionalidade desejada ao sistema especificado.

Os processos podem ser definidos como tipos instanciáveis, ou seja, é especificado um comportamento padrão para um determinado processo e podem ser criadas inúmeras instâncias deste processo. Estas instâncias são executadas concorrentemente e podem se comunicar umas com as outras.

Os blocos e processos SDL podem herdar suas características de um ancestral. Neste caso, as funcionalidades do ancestral também estão presentes no bloco/processo herdeiro, além de novas funcionalidades que podem ser incorporadas. Além disso, algumas funcionalidades podem ser redefinidas através da construção *REDEFINED*, desde que isso seja permitido pelo ancestral através do uso da construção *VIRTUAL*. Blocos, processos, procedimentos e transições de estado devido a recepção de sinais podem ser declarados como *VIRTUAL* e, portanto, redefinidos.

Uma das principais características do SDL é que, diferentemente de grande parte das linguagens de especificação formal (como Estelle e LOTOS, por exemplo), ele apresenta uma representação gráfica, chamada SDL-GR, que facilita bastante a compreensão do usuário e o desenvolvimento das especificações. Esta representação gráfica possui uma representação textual equivalente, chamada SDL-PR.

Neste trabalho, todo o desenvolvimento em SDL foi realizado fazendo-se uso da ferramenta CASE SDT, que permite o desenvolvimento das especificações em SDL através de sua representação gráfica (SDL-GR), facilitando assim a compreensão do sistema como um todo. O SDT disponibiliza também módulos capazes de realizar a validação da especificação desenvolvida e simulações de suas funcionalidades e casos críticos. Estas facilidades foram utilizadas para a detecção e correção de erros (ou falhas de especificação), o que garante a corretude das especificações geradas.

O SDT também disponibiliza uma representação textual das especificações (SDL-PR), que é utilizada como entrada para a ferramenta geradora de código distribuído Java desenvolvida neste trabalho.

O Apêndice A apresenta maiores detalhes das principais construções SDL e da estruturação da linguagem. É também apresentada a ferramenta SDT e as principais facilidades presentes na versão 4.2, utilizada neste trabalho.

2.6.3 Criando especificações em SDL a partir de descrições em ODL

Os objetos TINA são descritos nas recomendações disponibilizadas pelo TINA-C através da linguagem ODL, que tem como objetivo formalizar as interfaces dos objetos deixando seus aspectos comportamentais em aberto, a cargo de cada implementação. Assim, o SDL entra como alternativa para o desenvolvimento destes objetos, com o objetivo de formalizar os aspectos comportamentais, em passo anterior à implementação dos mesmos.

Para que isso seja possível, entretanto, é necessário que se utilize algumas regras de mapeamento entre as estruturas ODL e as estruturas SDL. As regras utilizadas neste trabalho estão descritas na tabela 2.2. Alguns dos aspectos da linguagem ODL, como por exemplo os parâmetros de qualidade de serviço não são contemplados por este mapeamento, por não terem sido necessários para o desenvolvimento das partes especificadas no trabalho (não há requisitos de qualidade de serviço para os objetos descritos nas especificações em ODL disponibilizadas para estes objetos).

Este mapeamento se destaca dos propostos pelos projetos TOSCA e SCREEN por apresentar a possibilidade de representação do tipo *Union*, através da construção de um novo tipo com o gerador *choice*. Através do uso do *choice* pode-se construir uma estrutura com diversos componentes, onde apenas um deles pode assumir algum valor em um dado instante de tempo.

Uma das maiores vantagens do mapeamento proposto neste trabalho sobre o apresentado em [32] é o fato dele permitir a existência de exceções no modelo SDL, o que é de fundamental importância em um sistema distribuído, como é o caso da arquitetura TINA.

Além disso, quando comparado ao mapeamento desenvolvido pelo projeto SCREEN [25] (que também provê suporte a exceções, porém mapeia uma chamada a um método em uma troca de sinais), este se mostra mais próximo da realidade por utilizar procedimentos exportados para representar os métodos dos objetos, uma vez que uma chamada a um procedimento

Estruturas ODL	**Mapeamento em SDL**
Object Type*	Block Type
Interface Type*	Process Type
Referência de objeto*	Pid
Operação oneway (assíncrona)*	Sinal precedido por pCALL_
Operação (síncrona)	Procedimento exportado onde, caso haja a possibilidade de geração de exceção, o valor retornado é do tipo choice, ou seja, pode assumir valores de tipos diferentes, um para o resultado e um para cada exceção que possa ser gerada
Enum*	NewType com os literais correspondentes
Struct*	NewType com a estrutura correspondente
Sequence, Array*	NewType com o uso do gerador String
Union	NewType com o uso do gerador Choice
Typedef*	Syntype
Constant*	Synonym
boolean	Boolean
char, wchar	Character
octet	Octet
string, wstring	Charstring
short, long, long long	Integer
unsigned short	Natural
unsigned long	Natural
unsigned long long	Natural
float, double	Real
parâmetros in	parâmetros in
parâmetros out e inout	parâmetros in/out

* Mapeamentos obtidos do projeto TOSCA

Tabela 2.2: Mapeamento de ODL para SDL

exportado possui a mesma característica síncrona de uma chamada a um método. Esta característica além de tornar o modelo SDL mais próximo do real, facilita bastante o processo de geração automática de código Java (capítulo 5), uma vez que as especificações comportamentais que representam o corpo de um método estão completamente separadas dentro de um procedimento. Já no mapeamento apresentado em [25], como uma chamada a um método é mapeada em um par de sinais (um de chamada e um de resposta/exceção), as especificações equivalentes aos corpos de todos os métodos de uma mesma interface encontram-se misturadas umas com as outras dentro do mesmo escopo (o do processo), ou seja, os comportamentos

de todos os métodos são especificados como o comportamento do processo, sendo assim, nem sempre é fácil delimitar o que faz parte de um método e o que faz parte de outro.

Os resultados dos métodos (procedimentos exportados) que podem gerar exceções são mapeados em tipos de dados definidos com o gerador *choice*. O gerador *choice* permite a criação de tipos compostos por diversos componentes (de qualquer tipo), sendo que uma variável deste tipo somente pode assumir o valor de um destes componentes de cada vez. Sendo assim, para o resultado de um procedimento exportado que possa gerar exceção, o tipo *choice* possui um componente (*result*) que representa o resultado propriamente dito e um componente para cada exceção que pode ser gerada. Portanto, se o procedimento exportado não gerar nenhuma exceção durante sua execução, a variável de retorno tem seu componente *result* definido como o valor a ser retornado, entretanto, se uma exceção for gerada, o componente relativo a esta exceção é ativado.

Com o desenvolvimento das especificações em SDL de sistemas TINA, obtém-se uma formalização dos aspectos comportamentais dos objetos envolvidos. Esta formalização permite a realização de simulações das principais funcionalidades do sistema além de possibilitar sua completa validação.

O capítulo 3 apresenta a aplicação desta metodologia na especificação formal dos objetos TINA responsáveis pelo modelo de subscrição e pela sessão de acesso TINA e o capítulo 4 apresenta os resultados das simulações e da validação realizada no sistema SDL desenvolvido.

2.6.4 A geração de código distribuído Java

Após a especificação do sistema em SDL com o uso da ferramenta SDT e da realização de simulações e validação, pode-se utilizar o modelo construído como base para o desenvolvimento de uma implementação do sistema. É proposto então, neste trabalho, a utilização de uma ferramenta desenvolvida para a geração automática de código distribuído Java, com uso de CORBA como plataforma distribuída, a partir das especificações criadas em SDL.

Java foi escolhido por diversas razões, mas a mais importante delas é a independência de plataforma. Esta característica traz muita liberdade ao código gerado, uma vez que somente é necessário direcionar a aplicação para uma plataforma (a máquina virtual Java) e ela funcionará em qualquer sistema operacional. Além disso, CORBA foi escolhido como DPE por estar se tornando um padrão de ambiente distribuído.

Desta forma, qualquer especificação desenvolvida em SDL seguindo a metodologia proposta, ou seja, com o uso das regras apresentadas na tabela 2.2, pode ser convertida automa-

ticamente para uma implementação em Java utilizando CORBA. Como saída da ferramenta geradora de código Java tem-se o código em Java dos objetos envolvidos e suas definições em IDL (necessárias para o uso do CORBA).

Para isso, algumas regras de mapeamento de estruturas SDL para estruturas Java e estruturas IDL são propostas na tabela 2.3. Este mapeamento não é baseado no desenvolvido no projeto SCREEN, tendo sido desenvolvido levando-se em consideração a recomendação da OMG (*Object Management Group*) para o uso de CORBA juntamente com Java [33]. O desenvolvimento de um novo mapeamento, diferente de um já existente teve como único objetivo a obtenção de um código que seguisse as normas impostas pela OMG para o desenvolvimento de aplicações Java com uso de CORBA e que seguisse a padronização da arquitetura TINA, o que permite a interoperabilidade do código gerado com qualquer outro sistema TINA.

Os procedimentos definidos nas especificações SDL são divididos em duas categorias: os procedimentos exportados e os não-exportados. Os não-exportados só podem ser acessados pelo processo ao qual ele pertence, sendo assim, não é necessário representá-lo no arquivo IDL gerado (uma vez que o arquivo IDL representa a interface do objeto para que outros objetos acessem suas informações). Da mesma forma, a classe gerada com a funcionalidade de um temporizador (*Timer*) é de uso exclusivo da classe à qual pertence, não necessitando de uma representação equivalente em IDL.

Na geração de código Java realizada pelo projeto TOSCA [27], como os métodos ODL são representados por trocas de sinais em SDL, a implementação Java gerada também faz uso de chamadas assíncronas (métodos *oneway*) para representar uma chamada síncrona. Neste trabalho, um grande avanço foi obtido neste sentido, pois o uso de procedimentos exportados como forma de representação para métodos ODL (tabela 2.2), permite que durante o processo de geração de código Java, estes procedimentos exportados sejam mapeados em métodos públicos Java, fazendo com que a implementação possua a característica esperada, ou seja, que os métodos ODL sejam representados por métodos Java equivalentes.

Os parâmetros in e in/out são mapeados, respectivamente, para uma passagem comum de parâmetros em Java e para uma passagem através do uso de *Holders*. *Holders* são classes definidas com a funcionalidade única de conter um objeto. Apesar de, em Java, o valor de um parâmetro não poder ser alterado pelo método, o conteúdo de um de seus campos (no caso de um objeto) pode ser modificado. Sendo assim, ao se passar um *Holder* contendo um determinado objeto, este objeto pode ser modificado pelo método, pois se trata de um campo do *Holder*. Desta forma, pode-se obter a mesma funcionalidade dos parâmetros in/out.

Estruturas SDL	**Mapeamento em Java**	**Mapeamento em IDL**
Block Type	Package	Module
Process Type	Classe Java derivada de org.omg.CORBA.Object	Interface
Pid	Referência a um objeto genérico org.omg.CORBA.Object	Tipo Object
Sinal assíncrono	Método	Método *oneway*
Procedimento Exportado	Método público com a definição de geração de exceção, se necessário	Método com a definição de geração de exceção, se necessário
Procedimento (não-exportado)	Método privado com a definição de geração de exceção, se necessário	Não representado
NewType	Definição dos tipos definidos segundo norma da OMG [33]	Definição dos tipos correspondentes em IDL
Syntype	Uso do tipo original	Typedef
Synonym	Atributo da interface (public static)	Constant
Timer	Classe com as funcionalidades de um timer	Não representado
Boolean	boolean	boolean
Character	char	char
Octet	byte	octet
Charstring	String	string
Integer	int	long
Natural	int	unsigned long
Real	double	double
parâmetros in	parâmetros comuns	parâmetros in
parâmetros in/out	uso de Holders [33]	parâmetros inout

Tabela 2.3: Mapeamento de SDL para Java e IDL

O capítulo 5 propõe uma ferramenta para a conversão das especificações em SDL para código Java, com a discussão de alguns conceitos envolvidos e apresentação de exemplos relacionados à especificação do modelo de subscrição e da sessão de acesso desenvolvida no capítulo 3.

Capítulo 3

Especificação em SDL dos objetos do modelo de subscrição e sessão de acesso

Neste capítulo, diferentemente dos trabalhos de especificação formal de objetos TINA anteriores (TOSCA [12] e SCREEN [13]) que se concentraram na sessão de serviço TINA, são propostas especificações formais para os objetos componentes da sessão de acesso [34] e para o modelo de subscrição (o qual disponibiliza informações para os objetos da sessão de acesso), integrantes da arquitetura de serviços TINA, cujos objetivos e funcionalidades foram descritas na seção 2.5 do capítulo 2. O modelo de subscrição exerce um papel bastante importante, em conjunto com os objetos da sessão de acesso, no controle dos assinantes aos serviços disponibilizados pelo provedor. Além disso, disponibiliza um serviço específico para cadastro de novos assinantes e de novos usuários, contratação de novos serviços, cancelamento de serviços contratados, alteração em contratos de serviço existentes, etc.

Na seção 3.1 é apresentado o sistema composto pelos objetos da sessão de acesso e do modelo de subscrição e a sua especificação em SDL realizada com o uso da metodologia proposta na seção 2.6 do capítulo 2. Através do uso do mapeamento proposto na tabela 2.2 do capítulo 2, as descrições em ODL dos objetos do sistema foram mapeados em estruturas SDL equivalentes, permitindo assim a formalização comportamental destes objetos.

Na seção 3.2 são apresentados três blocos SDL criados para auxiliar a especificação fornecendo algumas funcionalidades que se encontram fora do escopo do sistema especificado (serviço de resolução de nomes, por exemplo), mas que são bastante importantes. A seção 3.3 apresenta a organização interna dos blocos relativos à sessão de acesso TINA, mostrando

os processos que os compõem e descrevendo suas funcionalidades. É também mostrado como o conceito de herança pode auxiliar no processo de desenvolvimento do sistema, permitindo o reuso de especificações.

Apesar de sua importância, o modelo de subscrição ainda possui diversas questões em aberto. Sendo assim, a seção 3.4 apresenta a organização interna dos blocos pertencentes ao modelo de subscrição TINA, dando maior destaque ao bloco SUB (gerenciador das informações de subscrição), por ser o bloco de maior importância no modelo de subscrição. São propostas soluções para algumas questões que se encontram em aberto, tal como a organização interna do objeto SUB e conseqüentemente do bloco que o representa. Alguns métodos não previstos pelas especificações disponibilizadas em ODL pelo TINA-C são adicionados aos blocos especificados em SDL visando lhes conferir funcionalidades necessárias para o perfeito funcionamento do sistema.

Na seção 3.5 é apresentada a organização dos blocos do sistema especificado em SDL em *packages*. O uso desta forma de organização permite a reutilização das especificações do sistema em extensões que possam ser eventualmente construídas. Caso tenha-se o interesse de expandir o sistema — provendo também as funcionalidades de sessões de serviço, por exemplo — pode-se criar especializações das especificações desenvolvidas através do uso do conceito de herança.

3.1 Sistema de acesso e subscrição TINA em SDL

A parte da arquitetura de serviços TINA composta pela sessão de acesso e pelo modelo de subscrição é composta de diversos objetos, os quais foram apresentados nas figuras 2.9 e 2.13 respectivamente. O modelo de subscrição é de fundamental importância para o gerenciamento das informações relacionadas com os contratos de serviço estabelecidos entre os assinantes e o provedor. Essas informações são utilizadas pelo objetos da sessão de acesso TINA para que possam ser disponibilizadas para o usuário informações acerca dos serviços contratados, de forma que eles possam eventualmente ser ativados, iniciando assim uma sessão de serviço. Além disso, através dos objetos da sessão de acesso, pode-se iniciar um serviço de subscrição, responsável por permitir a manipulação dos dados relativos à contratação de serviços. Este serviço de subscrição é parte integrante da modelo de subscrição TINA.

Devido a forte interação existente entre os objetos componentes da sessão de acesso e do modelo de subscrição, seus objetos apresentados nas figuras 2.9 e 2.13 foram agrupados, formando o sistema de acesso e subscrição apresentado na figura 3.1.

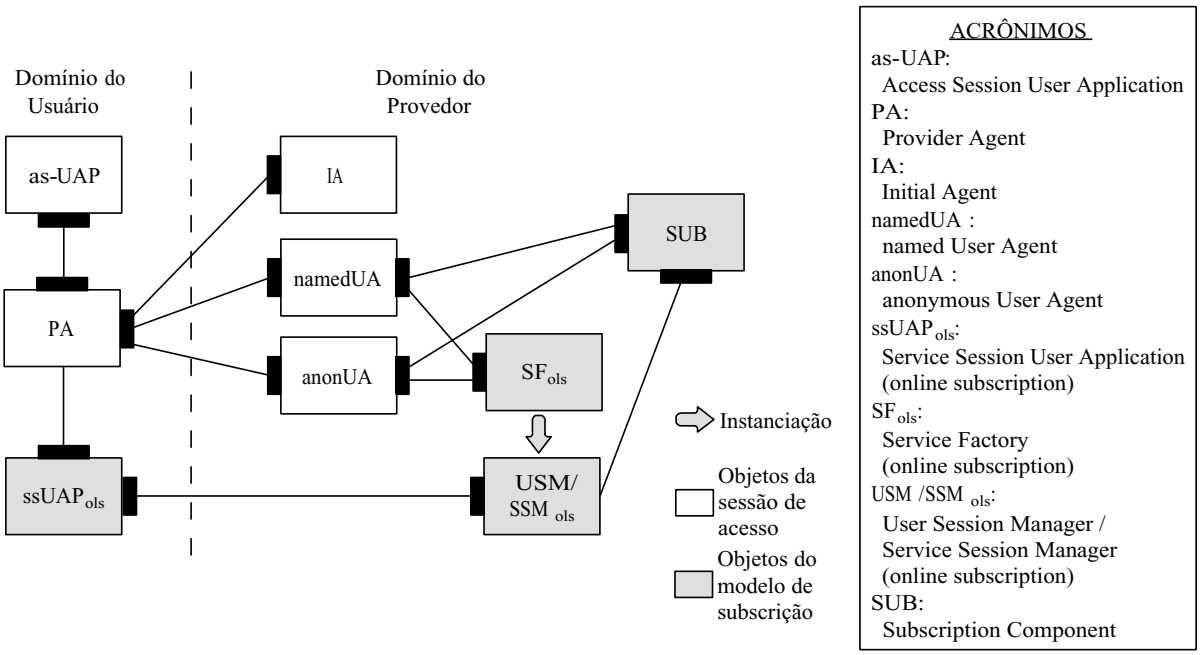


Figura 3.1: Relacionamentos entre objetos do Modelo de Subscrição e da Sessão de Acesso TINA

Esta figura mostra os objetos responsáveis por prover o acesso do usuário ao provedor (as-UAP e PA no domínio do usuário e IA, named e anonUA no domínio do provedor). São mostrados também os objetos responsáveis por prover o serviço de subscrição ($ssUAP_{ols}$ no domínio do usuário e SF_{ols} e USM/SSM_{ols} no domínio do provedor). O objeto SUB merece destaque por se tratar do objeto responsável pela gerência das informações de subscrição, fornecendo informações para os objetos da sessão de acesso (namedUA e anonUA) e permitindo a consulta e a alteração de suas informações por intermédio do objeto USM/SSM_{ols} do serviço de subscrição.

Com o intuito de prover uma formalização comportamental a estes objetos, complementando assim a formalização conferida pelo ODL (que se limita à especificação das interfaces), este sistema foi especificado através do uso da linguagem SDL (figura 2.13) [34].

Cada um dos objetos componentes do sistema da figura 3.1 é convertido em um bloco SDL com funcionalidade equivalente (figura 3.2). As interações entre estes objetos são representadas em SDL através de canais de comunicação, utilizados para trocas de sinais. As diversas interfaces que compõem um objeto TINA são convertidas para processos SDL que se comunicam entre si, fornecendo o comportamento desejado ao bloco que compõem. Os métodos disponibilizados por estas interfaces são representadas por procedimentos exportados, que

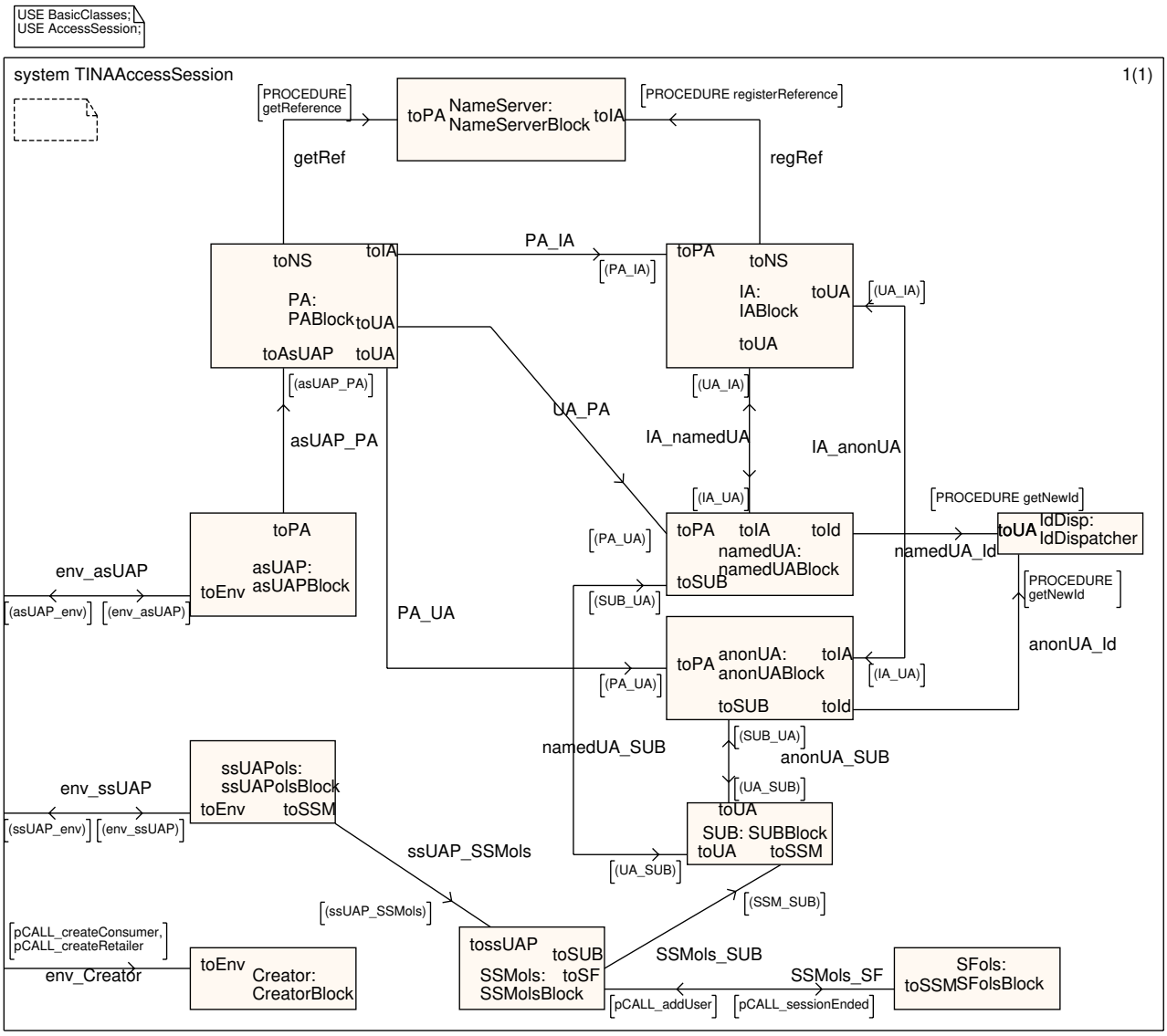


Figura 3.2: Sistema SDL representando o modelo de subscrição e a sessão de acesso TINA

podem ser acessados por outros processos (do mesmo bloco ou de outros blocos).

Adicionalmente, no modelo SDL foram implementados os blocos NameServer, IdDispatcher e Creator. Estes objetos não fazem parte das especificações TINA, porém são necessários na especificação do sistema em SDL para prover o serviço de resolução de nomes (bloco NameServer), fornecer identificadores de sessão (bloco IdDispatcher) e ajudar na criação de instâncias do consumidor e do provedor (bloco Creator). A especificação de um serviço de nomes visa simular o serviço disponibilizado pela plataforma CORBA em um sistema real. As sessões de acesso necessitam de identificadores que sirvam para distingüir as sessões umas

das outras, sendo assim um bloco capaz de fornecer estes identificadores é necessário. Já a presença de um bloco que simplifique a criação de um consumidor ou de um provedor através da criação de instâncias de seus blocos é bastante útil durante o processo de simulação.

3.2 Especificação dos blocos auxiliares

O **bloco NameServer** (figura 3.3) é um Servidor de Nomes, cuja única funcionalidade é fornecer ao usuário uma referência ao objeto IA do provedor desejado quando solicitado, para que se possa estabelecer uma comunicação entre o usuário e o provedor. Para isto é disponibilizado ao objeto PA um método do Servidor de Nomes chamado *getReference*. Para que o Servidor de Nomes possua referências para os objetos IA dos diferentes provedores existentes, sempre que uma nova instância de IA é criada, esta se registra junto ao Servidor de Nomes através do método *registerReference* representado em SDL pelo procedimento exportado de mesmo nome.

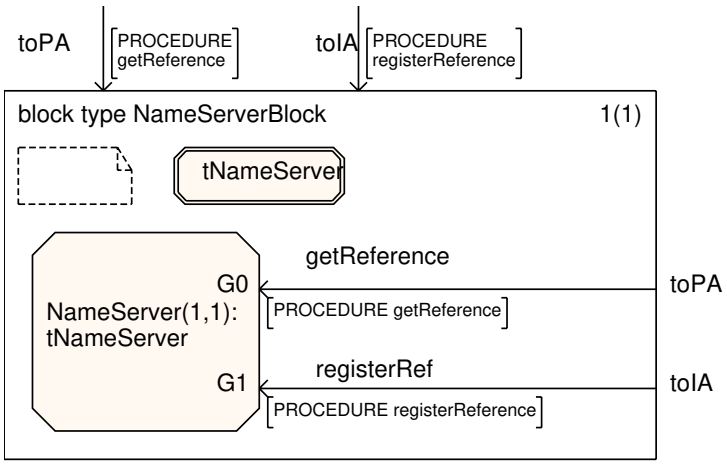


Figura 3.3: Bloco NameServer

O **bloco IdDispacher** (figura 3.4), por sua vez, é responsável apenas por gerar números seqüenciais (através do método *getNewId*) que serão utilizados pelo namedUA e anonUA como identificadores de sessões de acesso.

Finalmente, tem-se o **bloco Creator** (Figura 3.5), cujo único objetivo é tornar mais prática a criação de instâncias de novos consumidores e provedores durante a simulação do modelo. Desta forma, ao invés de se criar manualmente os objetos necessários para que inicie o funcionamento do provedor (objeto IA) e do consumidor (objetos as-UAP e PA), faz-se isso através apenas de dois sinais: *pCALL_createConsumer* e *pCALL_createRetailer*.

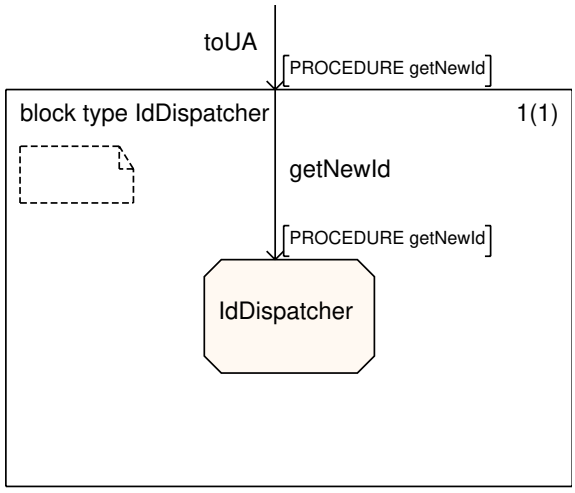


Figura 3.4: Bloco IdDispatcher

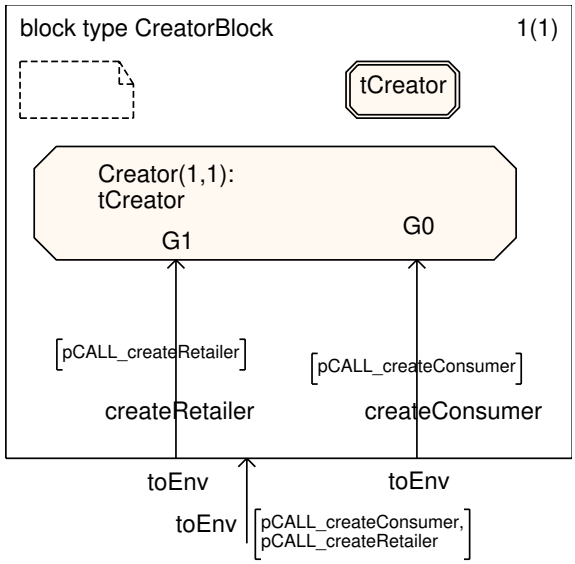


Figura 3.5: Bloco Creator

3.3 Especificação dos blocos referentes aos objetos ODL da sessão de acesso TINA

Os objetos pertencentes à arquitetura de serviços TINA foram organizados segundo as regras de mapeamento de ODL para SDL propostas. Eles foram mapeados em blocos SDL, suas interfaces em processos SDL e seus métodos em procedimentos exportados.

Entretanto, como um objeto especificado em ODL pode ser criado dinamicamente, en-

quanto uma instância de um bloco SDL não pode (apenas processos SDL podem), para cada bloco SDL, tem-se além dos processos que representam suas interfaces, um processo pré-existente que é responsável pela criação de instâncias do bloco — um processo criador. A figura 3.6 mostra o processo criador do bloco SUB (*SUB_Creator*). O par (1,1) após o nome do processo indicando que este processo possui uma instância inicialmente e que pode haver no máximo uma instância deste processo. Este processo cria instâncias de cada um dos processos do bloco e, após a criação dos mesmos, envia a cada um deles as referências de todos os outros processos componentes deste bloco, de forma que os processos possam se comunicar (uma vez que conhecem os *Pid* uns dos outros), e assim atuar como uma unidade, um objeto. Na figura 3.6, (0,) após o nome do processo *SUB_Manager* indica que inicialmente não há nenhuma instância deste processo e que não há um número máximo de instâncias possíveis para este processo.

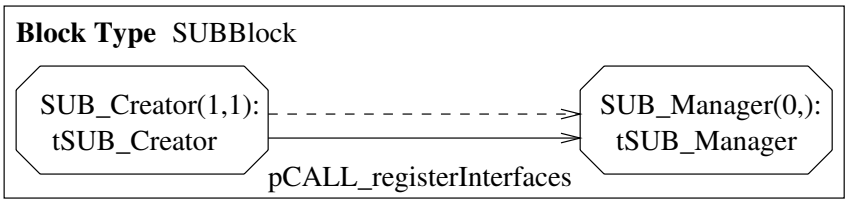


Figura 3.6: Solução para a criação de instâncias de blocos SDL

O **bloco as-UAP** é responsável por prover uma interface ao usuário, através da qual ele tem acesso ao ambiente TINA. Em sua especificação formal em SDL (figura 3.7) nota-se a presença do processo criador e de um processo, *i_Api*, que é responsável por prover esta interface para o usuário, representada pelos sinais que o bloco as-UAP troca com o ambiente através do canal *asUAP_to_env*.

O **bloco PA** (figura 3.8), que é responsável pelo contato entre o domínio do usuário e o domínio do provedor, recebe então solicitações de as-UAP. Cada um dos processos do bloco PA representa uma interface do objeto ODL equivalente. O processo *PA_i_Initial* é responsável por possibilitar ao bloco as-UAP o contato inicial com o provedor. O processo *PA_i_ProviderAuthenticate* provê meios para a autenticação do usuário e *PA_i_Access* permite o estabelecimento de sessões de acesso ao provedor.

O **bloco IA** (figura 3.9) é o ponto de contato inicial do bloco PA com o domínio do provedor. É o primeiro bloco com quem PA se comunica. O processo *PA_i_ProviderInitial* provê o contato inicial do usuário com o provedor, enquanto *PA_i_ProviderAuthenticate* é responsável pela autenticação do usuário (para isso são obtidos dados junto ao bloco SUB,

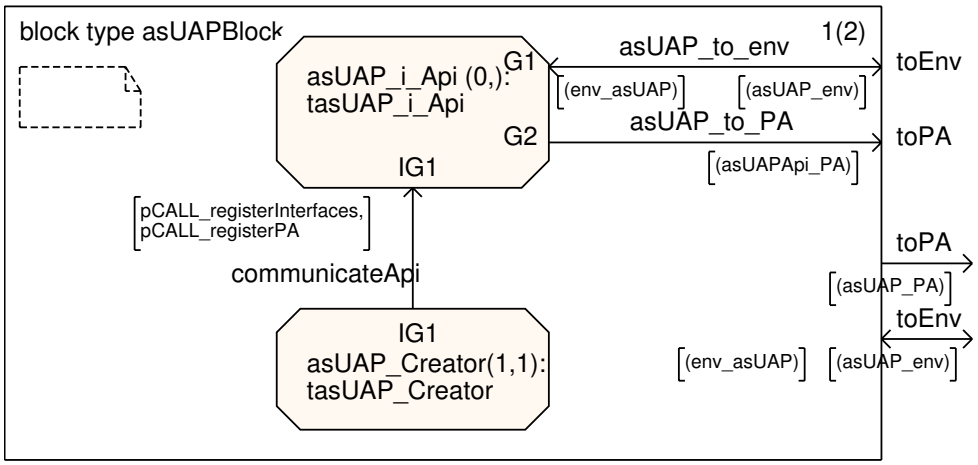


Figura 3.7: Especificação do objeto as-UAP (*access session User Application*)

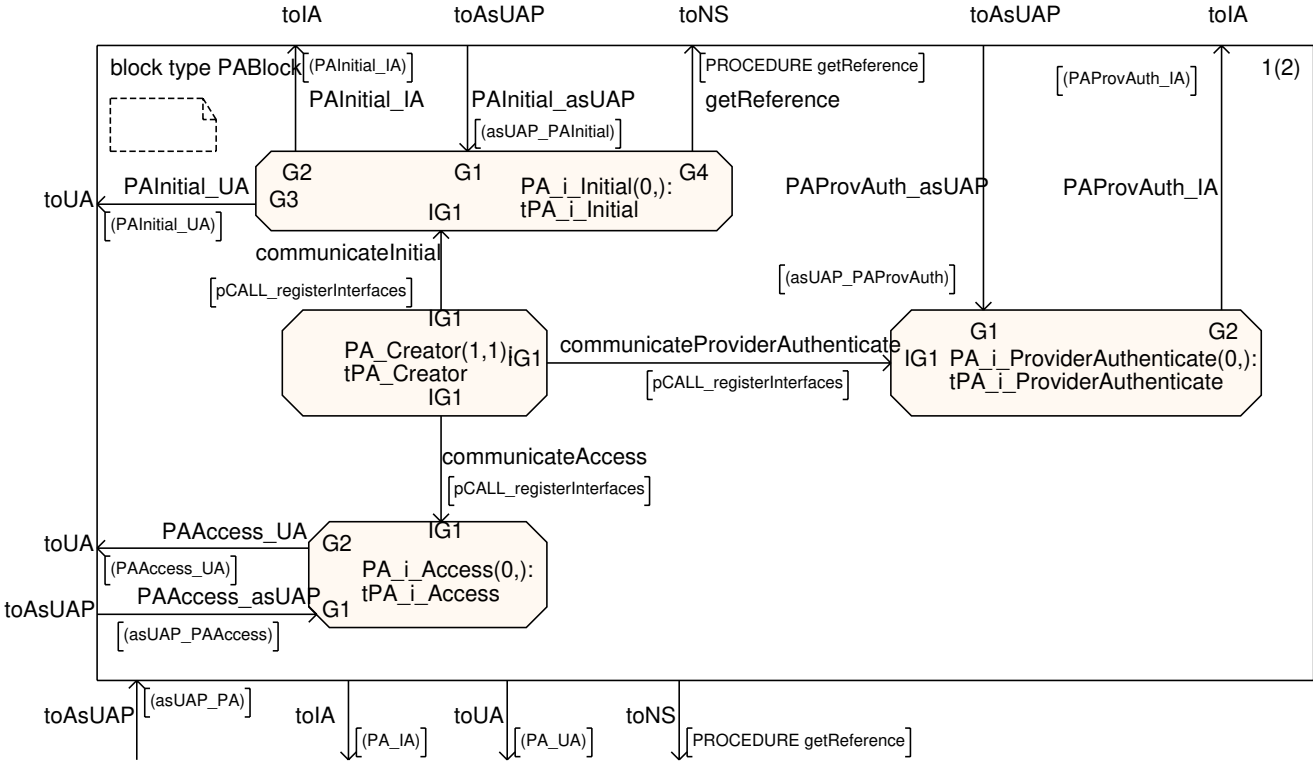


Figura 3.8: Especificação do objeto PA (*Provider Agent*)

que contém informações acerca de todos os usuários).

Após a autenticação do usuário, o bloco IA retorna para o bloco PA uma referência ao agente de usuário a ser utilizado no domínio do provedor.

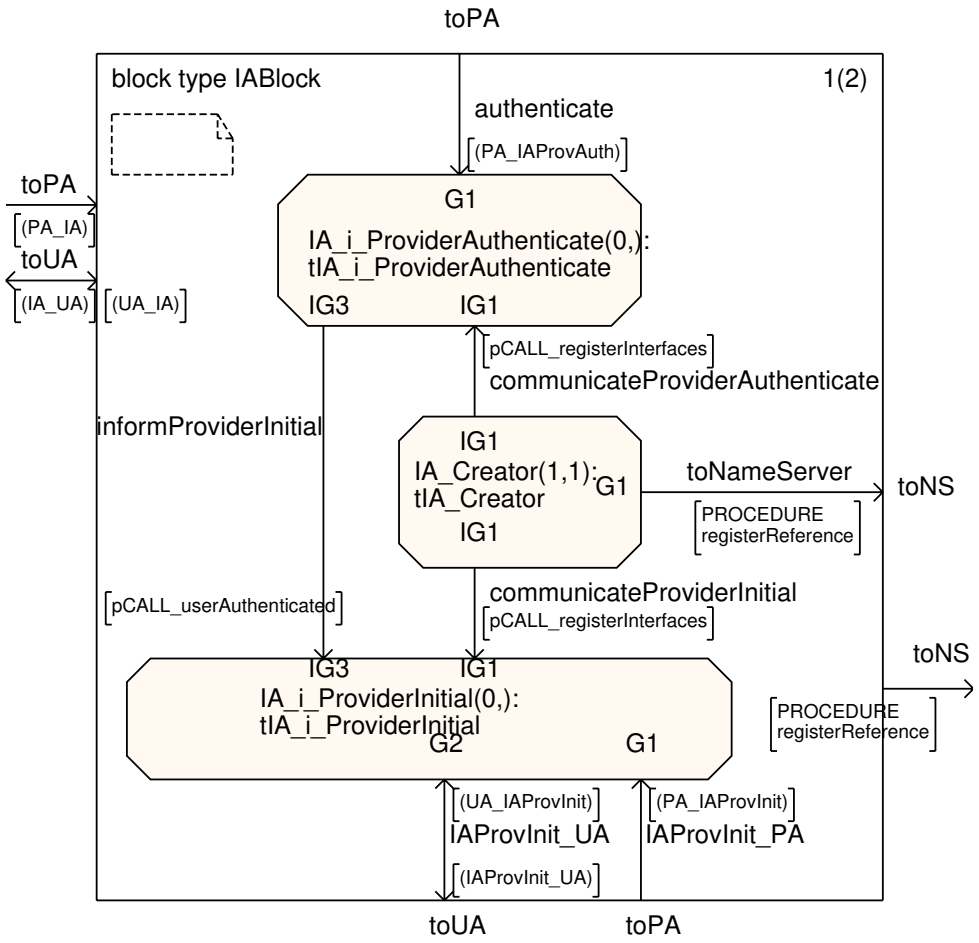


Figura 3.9: Especificação do objeto IA (*Initial Agent*)

O **bloco namedUA** (figura 3.10) representa o agente de usuário utilizado quando o usuário se identifica junto ao provedor. O processo *namedUA_i_Initial* permite que o usuário estabeleça sessões de acesso e, assim, retorne para o bloco PA uma referência ao processo *namedUA_i_ProviderNamedAccess* que provê serviços relativos ao acesso, tal como o acesso à lista de serviços contratados. Já o processo *namedUA_i_SubscriptionNotify* permite que o bloco SUB atualize a lista de serviços contratados pelo usuário sempre que essa lista seja alterada através do serviço de subscrição.

O **bloco anonUA** (figura 3.11), por sua vez, permite que o usuário se comunique com o provedor de forma anônima.Como usuário anônimo, é possível por exemplo iniciar o serviço de subscrição para realizar o cadastramento *online* de um assinante no provedor. Caso o usuário seja anônimo, o bloco IA retorna uma referência ao bloco anonUA (ao invés de

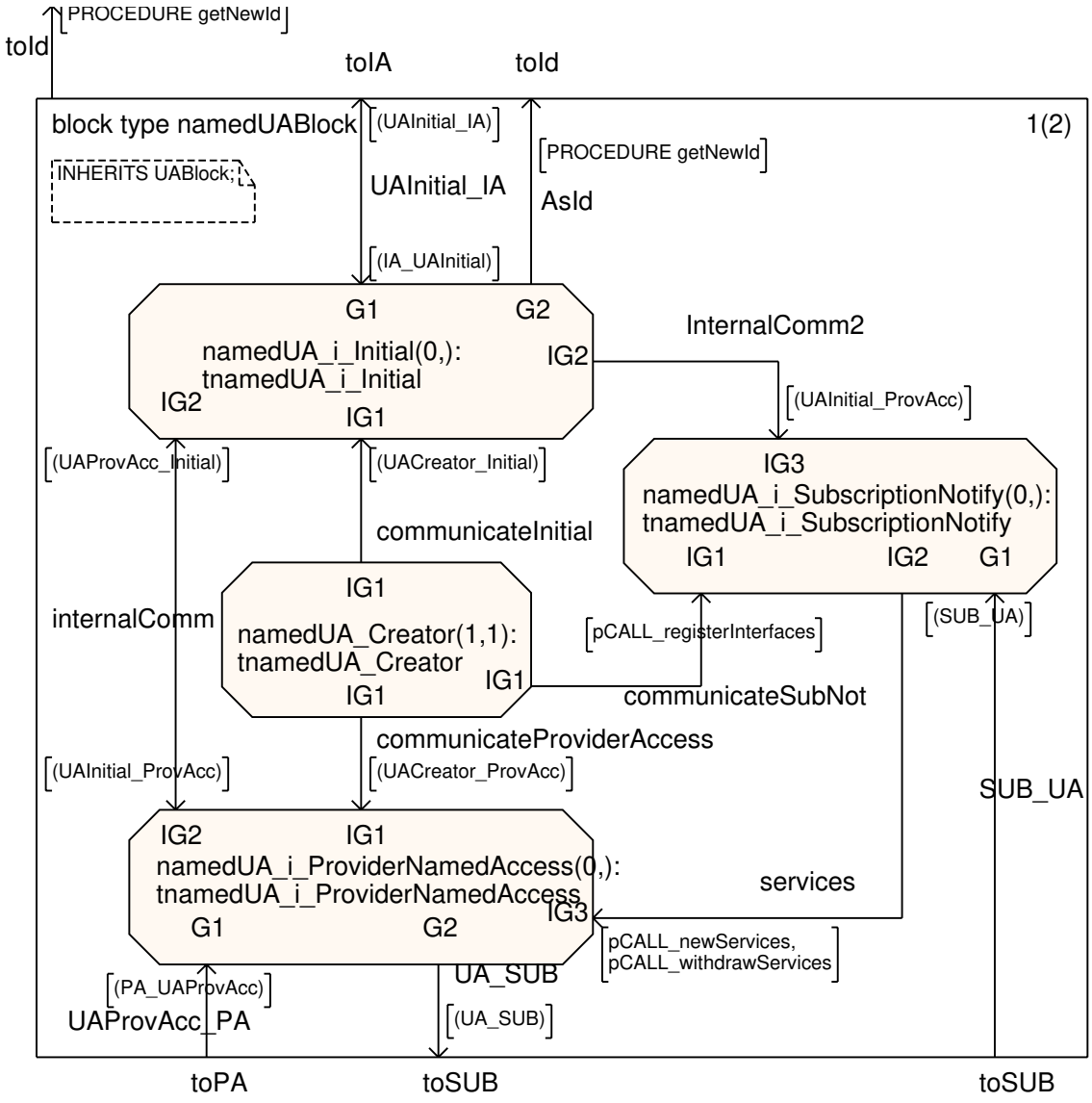


Figura 3.10: Especificação do objeto namedUA (*named User Agent*)

namedUA). Os processos que o compõem são equivalentes aos existentes no bloco namedUA, no entanto, os seus comportamentos são ligeiramente diferentes, uma vez que no neste caso não é necessária a realização do processo de autenticação do usuário.

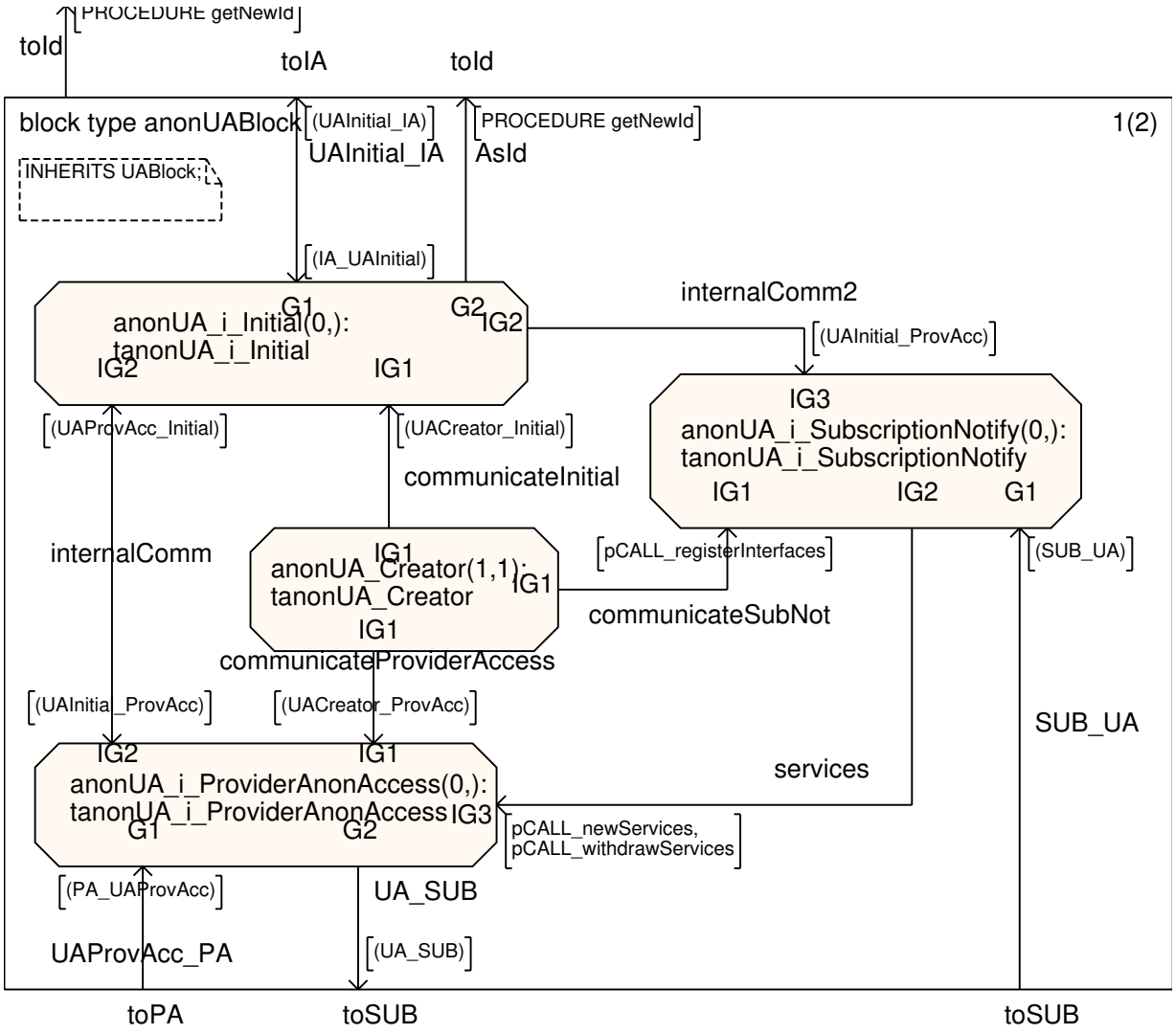


Figura 3.11: Especificação do objeto anonUA (*anonymous User Agent*)

3.3.1 O uso de herança na especificação dos blocos anonUA e namedUA

Como os blocos anonUA (figura 3.11) e namedUA (figura 3.11) são bastante similares, visando uma simplificação do processo de desenvolvimento dos mesmos, foi definido um bloco ancestral chamado UA (figura 3.12). Este bloco possui todas as funcionalidades comuns aos dois blocos e alguns pontos específicos são deixados para serem complementados durante a especificação formal de anonUA e namedUA.

O processo *UA_i_Initial* dá origem às especializações *namedUA_i_Initial* e *anonUA_i_Initial*

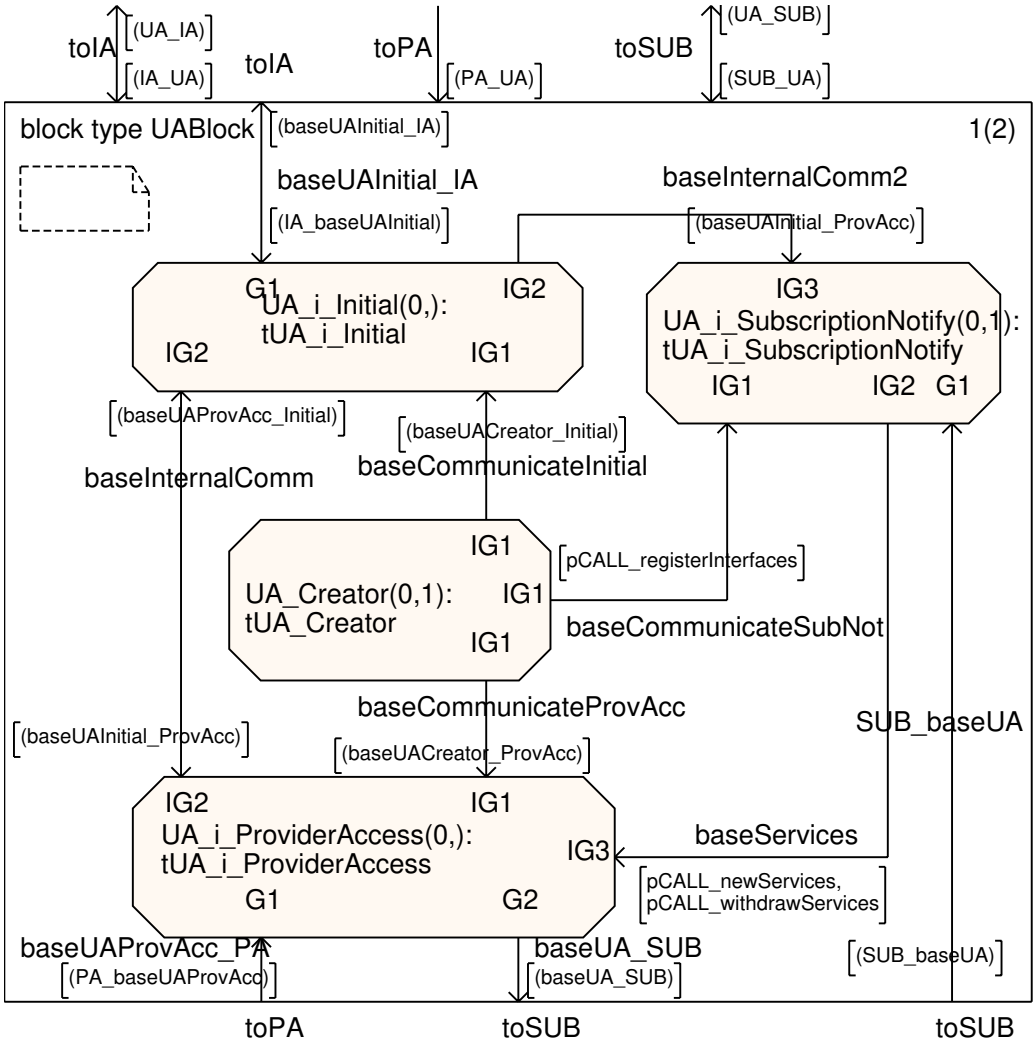


Figura 3.12: Especificação do objeto UA (*User Agent*)

dos blocos namedUA e anonUA respectivamente. Da mesma forma, o processo *UA_i_SubscriptionNotify* é especializado em *namedUA_i_SubscriptionNotify* e *anonUA_i_SubscriptionNotify*, o processo *UA_i_Creator* é especializado em *namedUA_i_Creator* e o processo *anonUA_i_Creator* e o processo *UA_i_ProviderAccess* é especializado em *namedUA_i_ProviderNamedAccess* e *anonUA_i_ProviderAnonAccess*.

Nestes últimos dois processos estão concentradas as principais complementações de especificação em relação ao bloco ancestral UA. Estes processos são responsáveis pelo estabelecimento de sessões de acesso e, para isso, é necessária a autenticação do usuário, no caso do acesso identificado fornecido pelo bloco namedUA. Já no bloco anonUA esta autenticação não existe, uma vez que o usuário é anônimo. Sendo assim, o procedimento exportado que

lida com o estabelecimento de sessões de acesso (*UA_setupAccessSession*, definido no processo *UA_i_ProviderAccess*) é redefinido em cada especialização.

O conceito de herança (figura 3.13) possibilita também o uso do conceito de polimorfismo. O procedimento que lista os serviços contratados pelo usuário, por exemplo, é definido no bloco UA, o que permite que o usuário (através do bloco PA) faça uma chamada a este procedimento da mesma forma, seja ele direcionado ao bloco namedUA ou ao bloco anonUA, não havendo necessidade de duas chamadas diferentes, uma para cada caso.

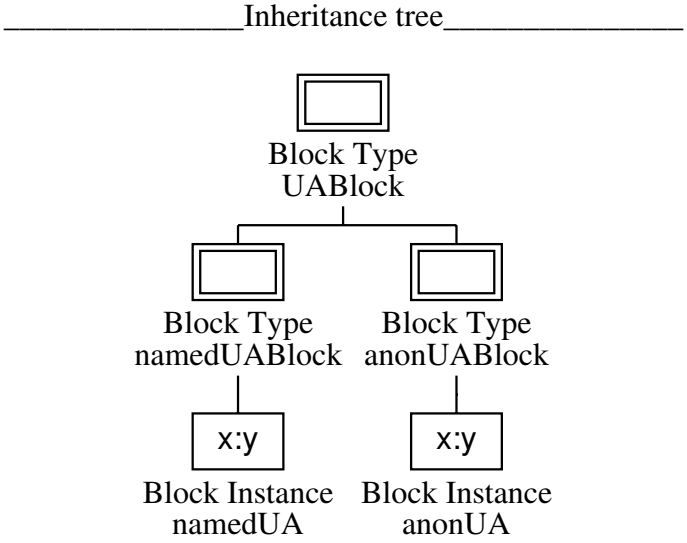


Figura 3.13: Blocos namedUA e anonUA como herdeiros do ancestral UA

3.4 Especificação dos blocos referentes aos objetos ODL do modelo de subscrição TINA

Através dos blocos anonUA e namedUA, o usuário pode iniciar sessões de serviço para que então possa fazer uso dos serviços contratados por ele. Dentre estes serviços, pelo menos um é disponibilizado para todos os usuários (até mesmo para o usuário anônimo), o serviço de subscrição. Através do uso deste serviço, pode-se contratar novos serviços, cancelar serviços contratados, cadastrar novos usuários, etc.

O **bloco SFols** (figura 3.14) é então responsável pela criação de uma instância do bloco SSMols no domínio do provedor, uma vez que seja solicitado o acesso a este serviço. Um único processo ($SFols\_i\_SSCreate$) é disponibilizado para os agentes de usuário (blocos anonUA

e namedUA) para que a criação dos blocos SSMols seja solicitada.

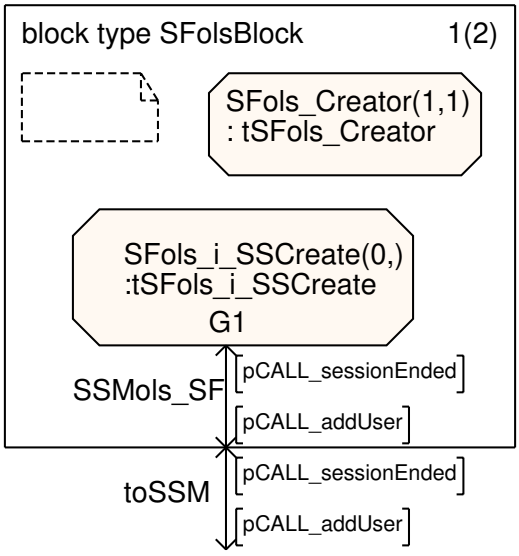


Figura 3.14: Especificação do objeto SF_{ols} (*Sevice Factory - online subscription*)

O **bloco SSMols** (figura 3.15) representa o objeto USM/SSM_{ols}. Apenas um processo é disponibilizado para que o usuário (através do bloco ssUAPols) tenha acesso a todas as funcionalidades disponibilizadas pelo serviço de subscrição. Este processo se comunica com o bloco que representa o objeto SUB, de onde obtém as informações necessárias. Este bloco é um intermediário entre o usuário em o bloco SUB, sendo assim responsável por controlar a sessão de serviço de subscrição.

O **bloco ssUAPols** (figura 3.16) é então criado no domínio do usuário assim que uma sessão de serviço de subscrição é solicitada. Este bloco é responsável por prover uma interface para que o usuário tenha acesso ao serviço de subscrição. O processo *ssUAPols_i_Api* provê a interface com o usuário, que é representada pelo sinais trocados com o ambiente através do canal de comunicação *Environment*. O processo *ssUAPols_i_Api* se comunica com o bloco SSMols presente no domínio do provedor, fornecendo assim o acesso ao serviço de subscrição.

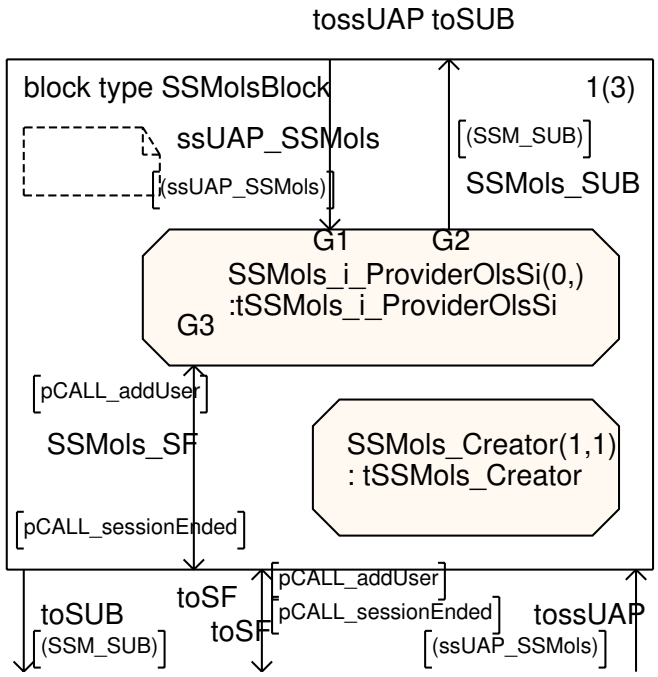


Figura 3.15: Especificação do objeto USM/SSM_{ols} (*User Session Manager/Service Session Manager - online subscription*)

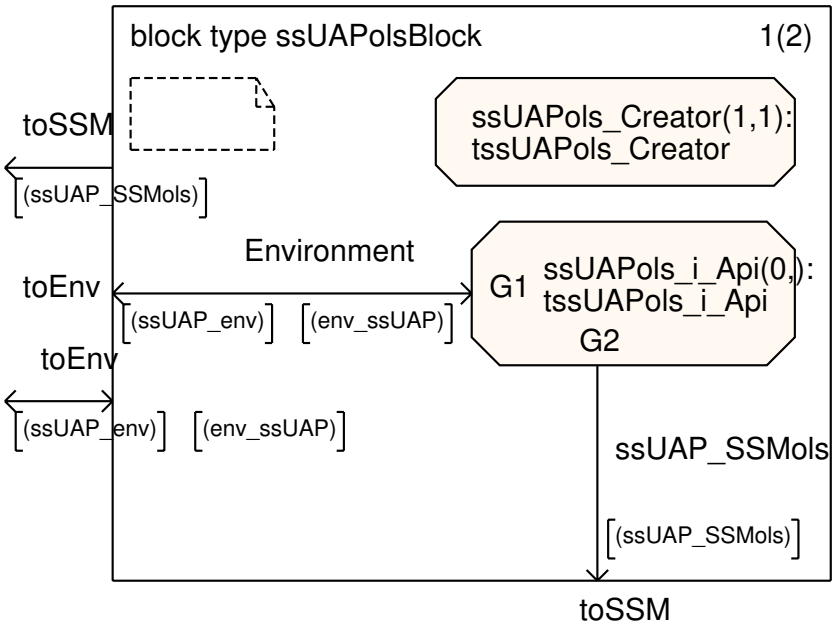


Figura 3.16: Especificação do objeto $ssUAP_{ols}$ (*service session User Application - online subscription*)

3.4.1 Especificação objeto SUB (Subscrição)

O objeto SUB da figura 3.1 é o principal componente do modelo de subscrição. Ele é o responsável pelo gerenciamento das informações relacionadas aos assinantes e aos serviços contratados pelos mesmos. Para isso algumas interfaces são disponibilizadas para os outros objetos com os quais se relaciona. Nas recomendações TINA, as seguintes interfaces são descritas (utilizando ODL) para o objeto SUB:

i_InitialAccess: ponto de contato inicial ao objeto SUB. Retorna as referências às interfaces que operam com informações relacionadas aos assinantes e serviços contratados.

i_Subscribe: interface oferecida ao componente USM/SSM_{ols} para a criação de novos assinantes, a contratação de serviços e o cancelamento de contratos existentes;

i_SubscriberInfoQuery: interface oferecida aos componentes $namedUA$ e $anonUA$ para que possa obter informações associadas ao usuário representado por estes objetos;

i_SubscriberInfoMgmt: interface oferecida ao componente USM/SSM_{ols} para que possa adicionar, remover ou modificar informações associadas a assinantes;

i_ServiceContractInfoMgmt: interface oferecida ao componente USM/SSM_{ols} para que possa adicionar, remover ou modificar informações associadas aos serviços contratados.

Para uma melhor organização destas interfaces, o objeto SUB foi modelado neste trabalho da maneira apresentada na figura 3.17. Nesta figura é apresentada a organização interna do objeto. As interfaces *i_SubscriberInfoQuery* e *i_SubscriberInfoMgmt* estão diretamente relacionandas à gerência de informações de assinantes. As interfaces *i_Subscribe* e *i_InitialAccess* atuam como coordenadores da subscrição, permitindo a contratação de novos serviços e fornecendo referências para as outras interfaces. A parte relacionada à gerência das informações dos serviços contratados é composta por diversas instâncias das interfaces *i_ServiceContractInfoMgmt*, uma para cada contrato de serviço. As referências para estas interfaces são gerenciadas pela interface interna *Manager*.

Por último, existe uma interface interna chamada *DBMSBroker* que é responsável pela mediação entre todas as interfaces e a base de dados que armazena as informações de subscrição. Desta forma é definida uma interface padrão para a obtenção dos dados, o que faz com que o objeto seja independente do banco de dados utilizado. Caso o gerenciador de banco de dados seja modificado, altera-se apenas a implementação do *DBMSBroker*, sendo

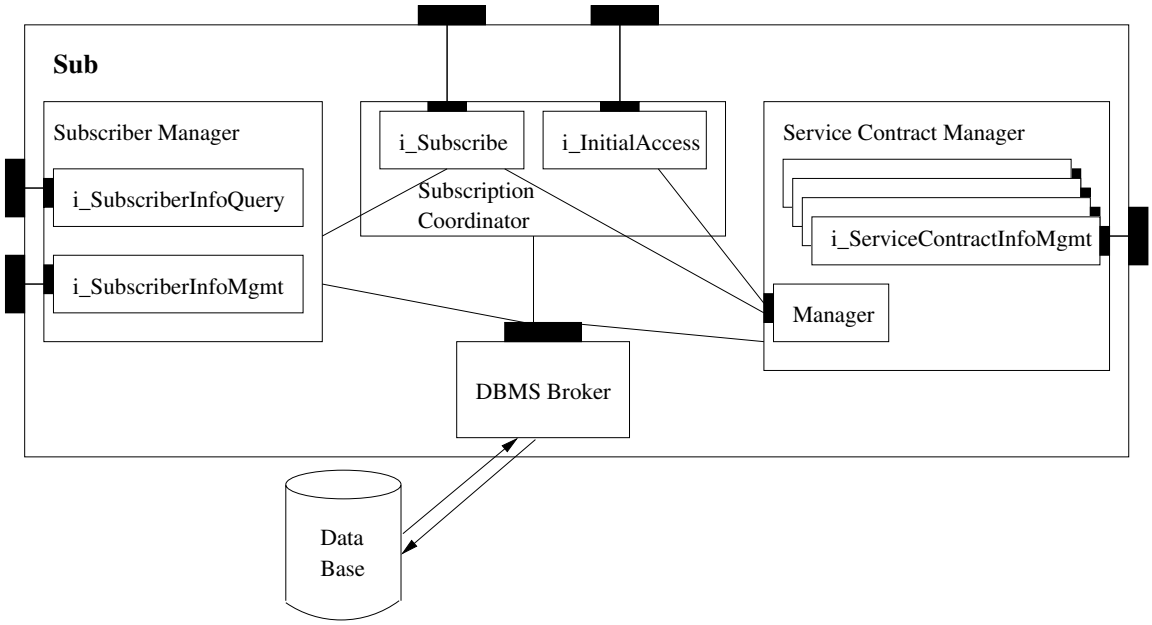


Figura 3.17: Organização interna do objeto SUB

essa alteração transparente para os demais componentes do objeto SUB. Além disso, uma vez que este objeto será especificado em SDL, após o término das especificações, no caso de uma eventual implementação, independente do banco de dados escolhido, as especificações de todas as interfaces continuam válidas, necessitando apenas de modificações no mediador *DBMSBroker*.

O *bloco SUB* que, em SDL, representa o objeto SUB, é apresentado na figura 3.18. Neste bloco estão presentes os processos que representam todas as interfaces descritas acima, além do processo criador.

O **processo criador SUB_Creator** (figura 3.19) contém apenas um procedimento responsável pela criação de instâncias do objeto, ou seja, dos processos componentes do bloco que representa o objeto SUB. Este procedimento (*CreateSUB*) cria instâncias das interfaces do objeto e envia a cada uma delas referências aos processos *SUB_Manager* e *SUB_DBMSBroker*. Além disso, envia ao processo *SUB_Manager* referências a todos os outros processos (através do sinal *pCALL_registerInterfaces*).

Desta forma, o processo *SUB_Manager* é o único com conhecimento das referências a todos os processos e, sempre que algum processo necessitar de referências para outros, essa informação pode ser obtida através do recurso de variável revelada do SDL. Através deste recurso, o valor da variável que contém as referências pode ser visualizado em todos os

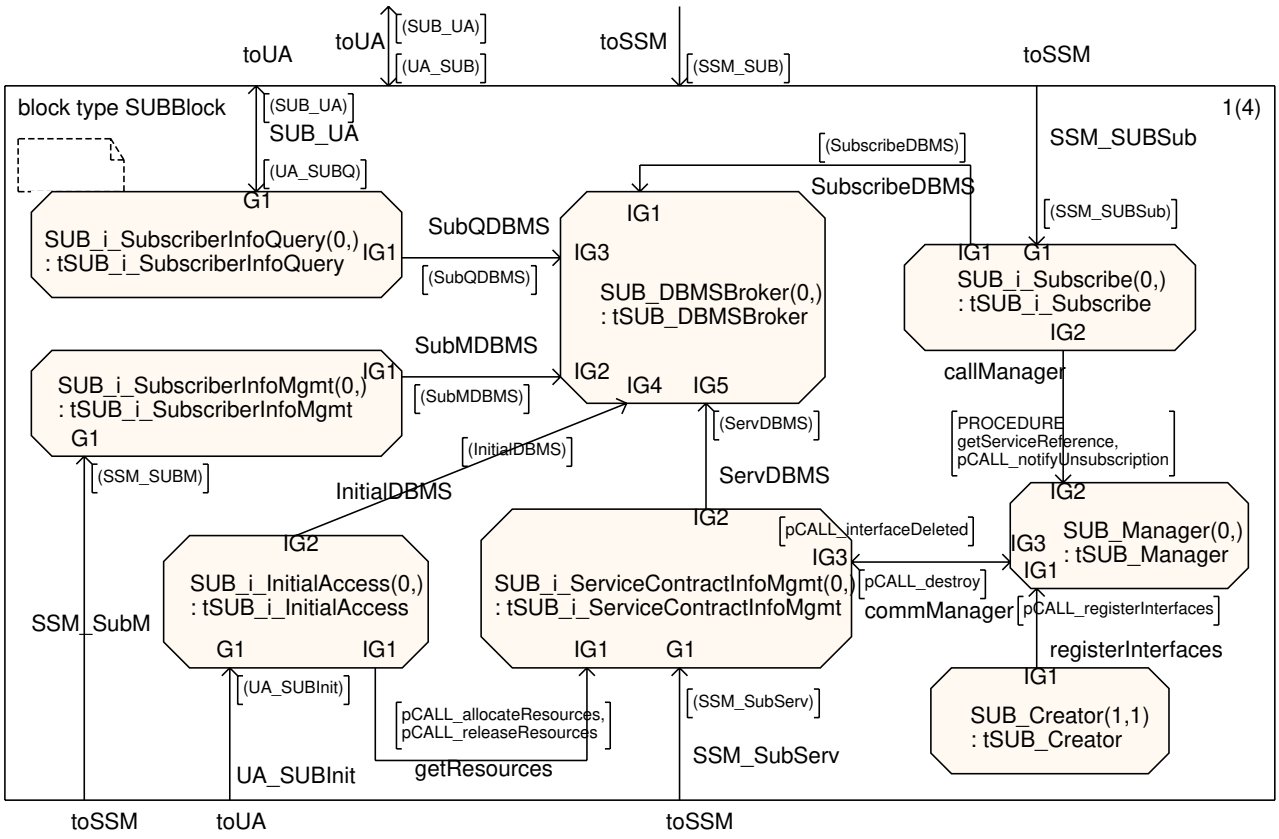


Figura 3.18: Especificação do objeto SUB (*Subscription Component*)

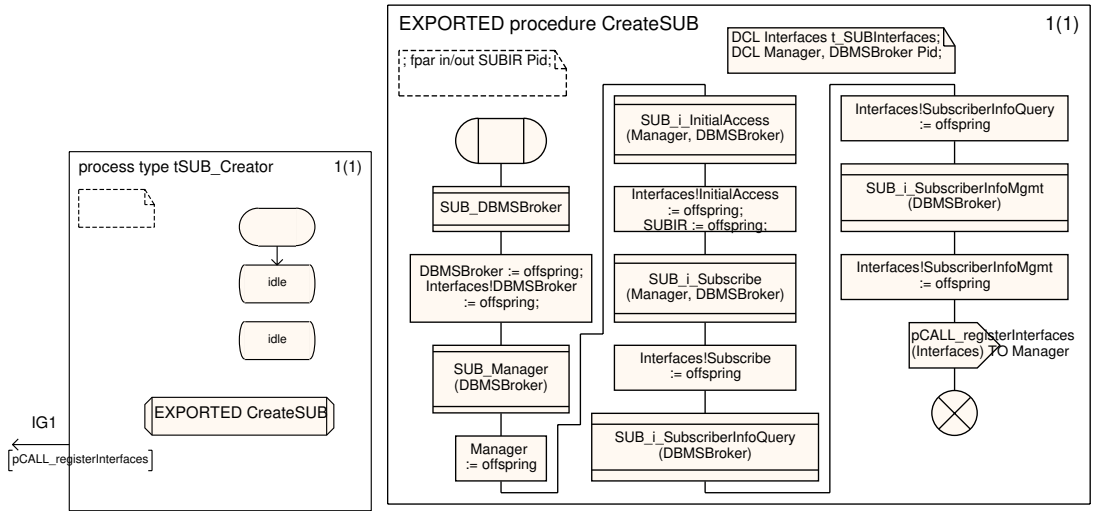


Figura 3.19: Processo *SUB_Creator* (Bloco SUB) e seu único procedimento

outros processos do bloco SUB, desde que eles possuam uma referência para o processo *SUB_Manager*, onde a variável foi declarada.

O **processo SUB_DBMSBroker** (figura 3.20), que representa a interface interna *DBMS-Broker*, provê diversos procedimentos responsáveis pela obtenção/modificação de informações sobre os usuários e os serviços contratados junto a uma base de dados.

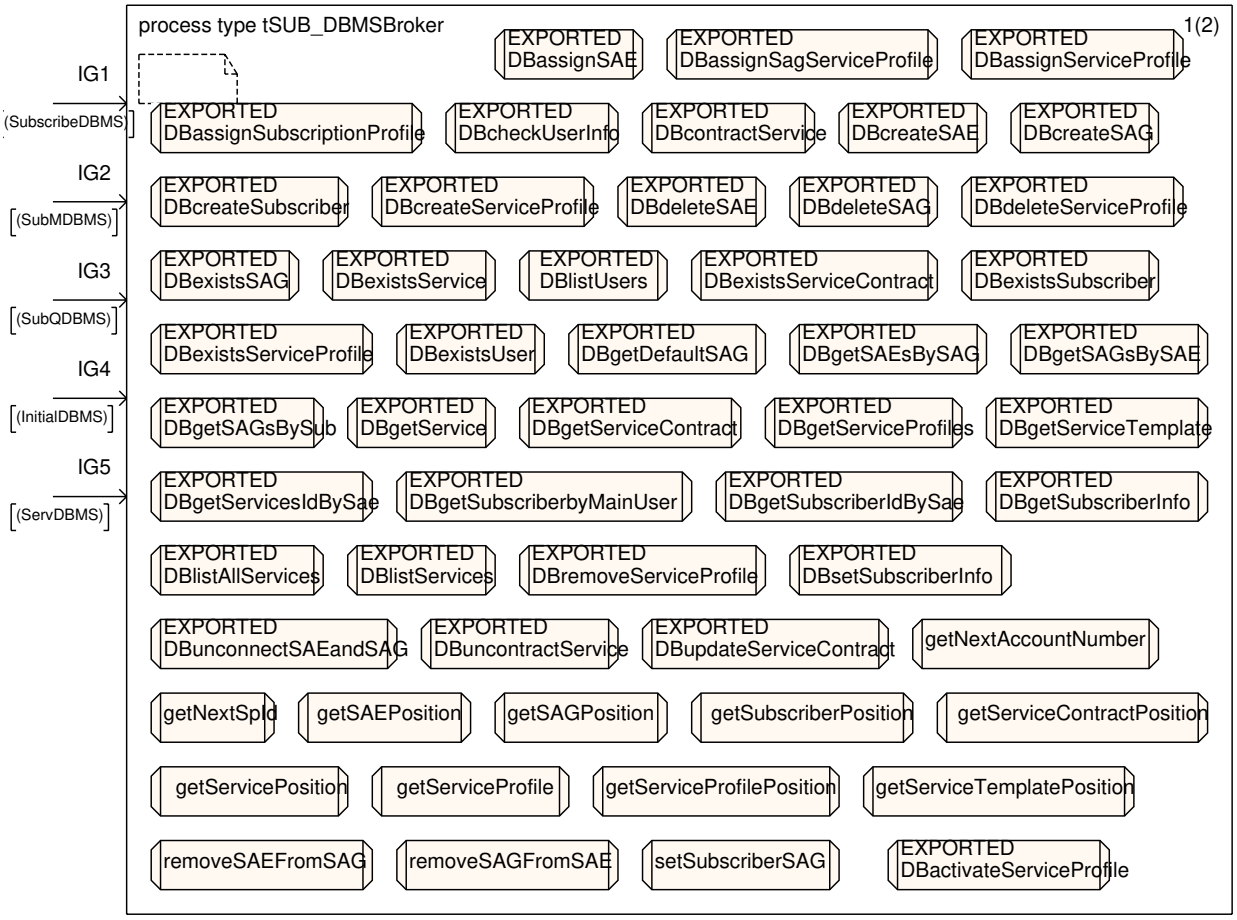


Figura 3.20: Especificação do processo *SUB_DBMSBroker* (Bloco SUB)

A base de dados acessada pelo processo *SUB_DBMSBroker* representa o modelo de informações de subscrição apresentado na figura 2.12. Sua especificação em SDL (figura 3.21) é representada por uma série de estruturas que representam cada um dos objetos componentes do modelo (Assinante, Serviço, Contrato de Serviço, Perfis de Serviço, Perfis de Subscrição, SAGs, Usuários e Templates de Serviço) e estas estruturas são armazenadas em listas, ou seja, tipos criados a partir do gerador *String* do SDL.

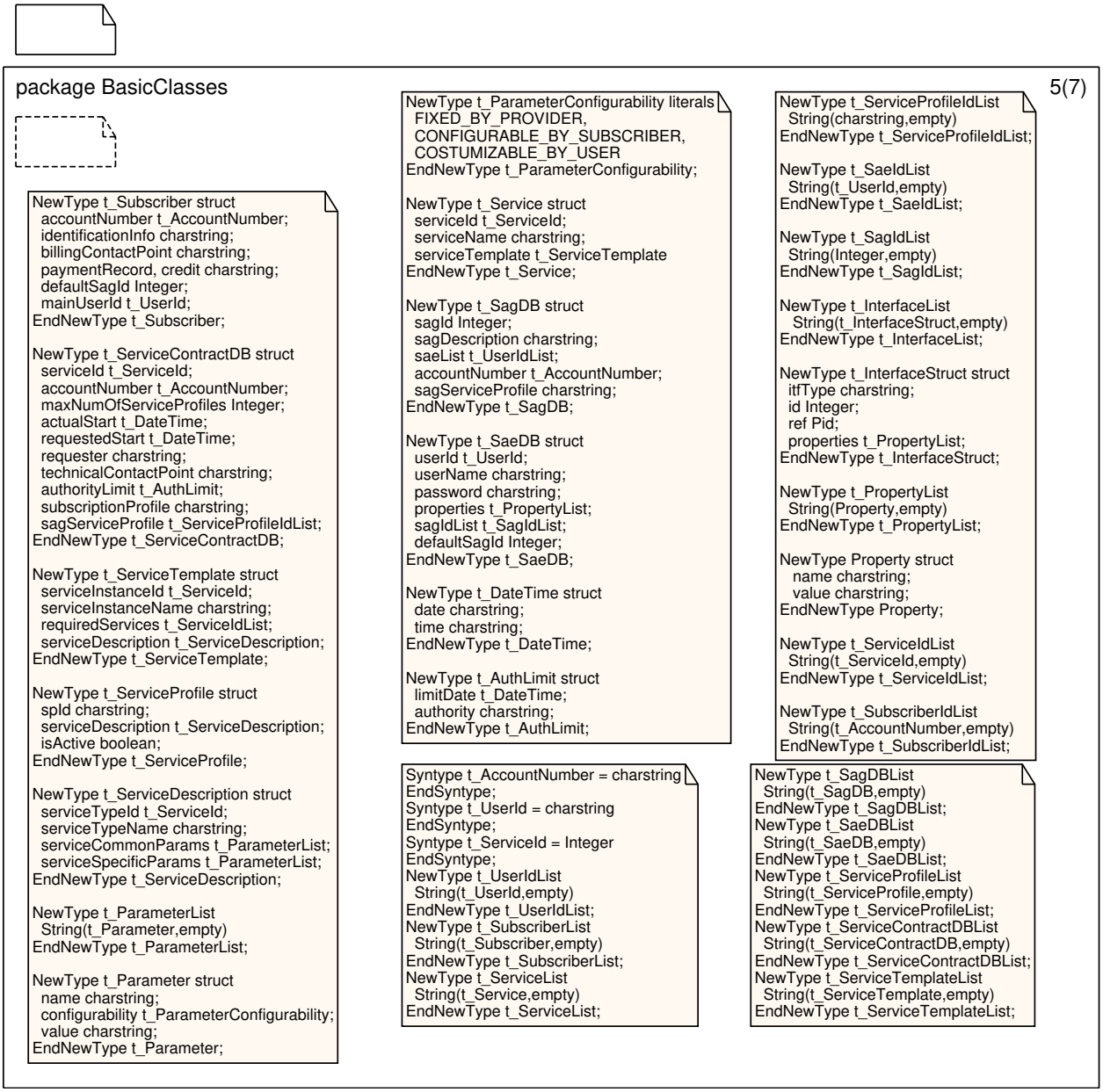


Figura 3.21: Especificação em SDL do modelo de informações de subscrição

O **processo SUB_Manager** (figura 3.22), que representa a interface interna *Manager*, é responsável pelo gerenciamento das referências aos diversos processos *SUB_i_ServiceContract-InfoMgmt* criados.

Este processo pode receber alguns sinais de outros processos que compõem o bloco SUB. Além do sinal *pCALL_registerInterfaces* que é recebido do processo criador portando re-

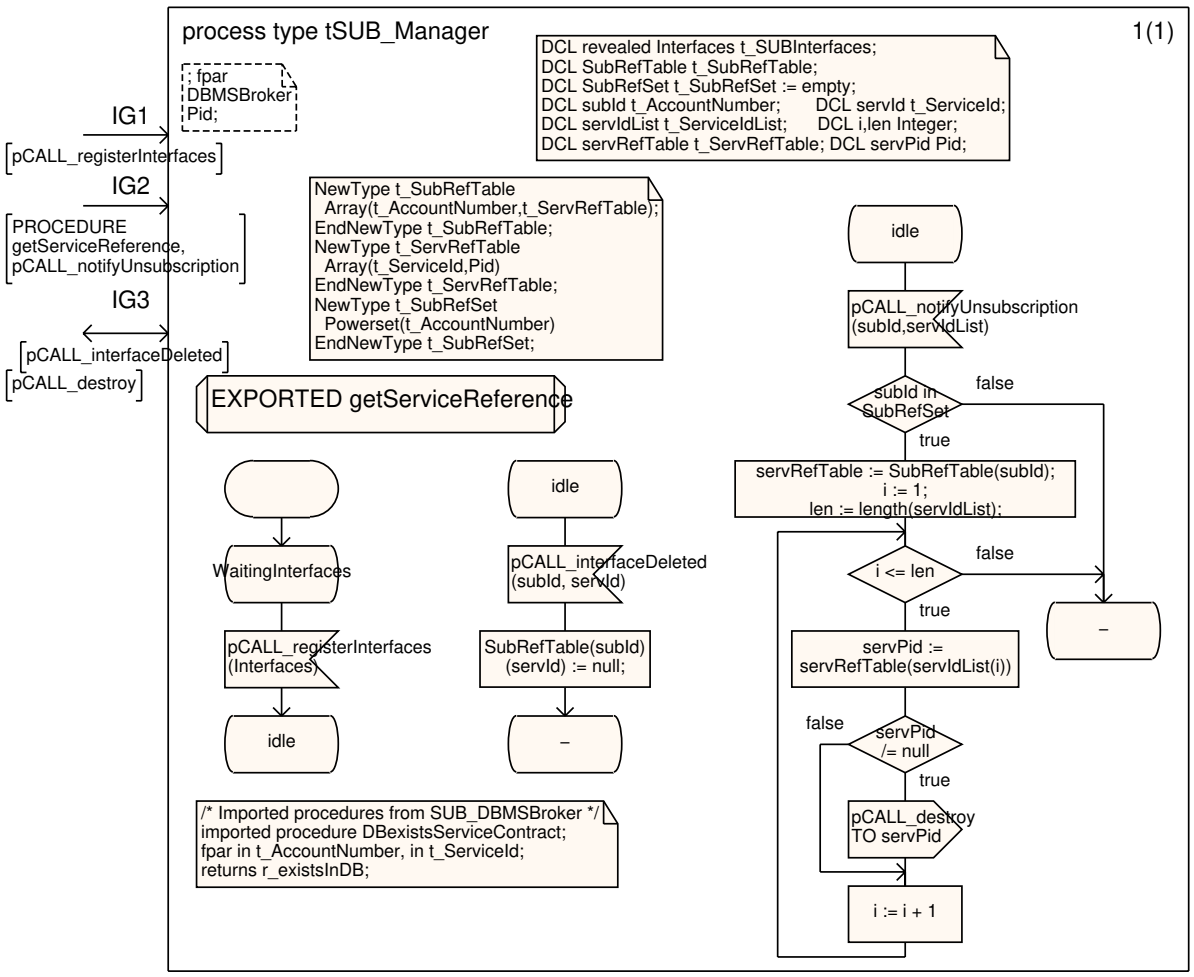


Figura 3.22: Especificação em SDL do processo *SUB_Manager* (Bloco SUB)

ferências a todos os processos componentes do bloco, pode-se receber o sinal *pCALL_interfaceDeleted* de um processo *SUB_i_ServiceContractInfoMgmt* informando que o mesmo foi destruído, para que então essa informação possa ser atualizada junto à tabela de controle de referências responsável pela gerência das instâncias destes processos. Também pode ser recebido o sinal *pCALL_notifyUnsubscription* da interface *SUB_i_Subscribe* sempre que a contratação de um serviço é cancelada por um assinante. Assim, pode-se solicitar a destruição do respectivo processo *SUB_i_ServiceContractInfoMgmt* através do envio do sinal *pCALL_destroy* e atualizar as tabelas de gerência das instâncias dos processos de contratos de serviço.

O **procedimento getServiceReference** (figura 3.23), o único deste processo, tem por objetivo fornecer a referência do processo *SUB_i_ServiceContractInfoMgmt* responsável por um certo contrato de serviço, dado o identificador deste contrato. Nesta figura, pode-

se também perceber a presença de um procedimento importado *DBexistsServiceContract*. Isto indica que este processo (bem como os procedimentos que o compõem) pode fazer uso deste procedimento que foi exportado por outro processo — neste caso, pelo processo *SUB_DBMSBroker*.

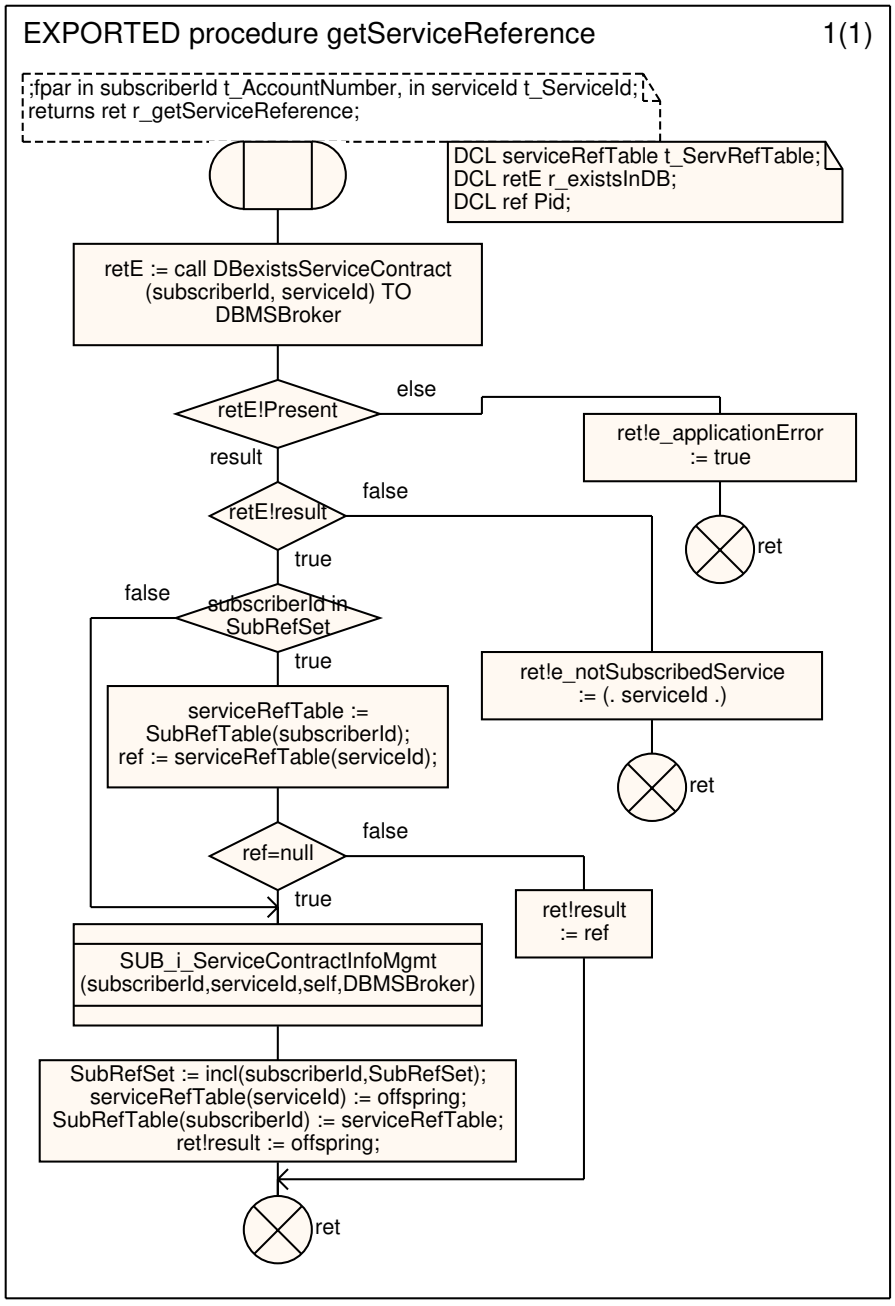


Figura 3.23: Especificação do procedimento *getServiceReference* (Processo *SUB_Manager* / Bloco SUB)

Na especificação formal deste procedimento exportado pode-se notar a representação dada à geração de uma exceção através da atribuição realizada à variável de retorno (*ret*) do procedimento. A definição do tipo desta variável é apresentada na figura 3.24. Trata-se de um tipo construído através do uso do gerador *choice* e, sendo assim, apenas um dos componentes da variável pode ser atribuído algum valor em um determinado instante. Caso se atribua um valor ao componente *result*, isso significa que o procedimento finalizou sem a ocorrência de exceções e que o resultado é retornado neste componente. No caso do procedimento *getServiceReference*, o resultado é uma referência à interface relativa ao contrato de serviço indicado pelos parâmetros *subscriberId* e *serviceId*. Assim sendo, o componente *result* é do tipo *Pid*. Entretanto, no caso de um procedimento que pode gerar exceções mas que não retorna nenhum resultado, o campo *result* é definido como sendo do tipo *null*. Para indicar que nenhuma exceção foi gerada bastaria fazer $ret!result := null$.

```
NewType r_getServiceReference
choice  result null;
  e_notSubscribedService t_notSub;
  e_applicationError boolean;
EndNewType r_getServiceReference;

NewType t_notSub struct
 subscriberId t_AccountNumber;
EndNewType t_notSub;

Syntype t_AccountNumber=charstring
EndSyntype t_AccountNumber;
```

Figura 3.24: Definição do tipo resultado do procedimento *getServiceReference*

Caso uma exceção seja gerada, atribui-se um valor ao componente relacionado à mesma. Dois casos são possíveis e ambos podem ser bem representados neste procedimento. Caso a exceção necessite de parâmetros que a descrevam mais detalhadamente (como é o caso da exceção *e_notSubscribedService*, que indica que um certo serviço não faz parte dos contratados pelo assinante e, que portanto, necessita do identificador do serviço para que se saiba sobre qual serviço está se referenciando), estes parâmetros são representados por campos de uma estrutura SDL que é utilizada para representar esta exceção (*t_notSub*, neste caso). Caso a exceção não necessite de nenhum parâmetro (como é o caso de *e_applicationError*, que indica a ocorrência de um erro na aplicação, como a indisponibilidade da base de dados, por exemplo), ela é definida como *boolean*. Sendo assim, para indicar a ocorrência da exceção basta fazer $ret!e\_applicationError := true$.

A detecção da ocorrência de uma exceção quando um procedimento é invocado é realizada através do operador *Present*. Este operador indica qual campo de um tipo *choice* está ativado. Na figura 3.23 é mostrada a verificação da ocorrência de uma exceção após a chamada do procedimento *DBexistsServiceContract*. Neste momento é avaliado se o procedimento teve sua execução normal (neste caso o campo *result* é o ativo) ou se ocorreu uma exceção (o caso indicado por *else*). Apesar de ter se generalizado a verificação para o caso da execução normal ou ocorrência de qualquer exceção, a ocorrência de cada exceção pode ser verificada individualmente, bastando verificar se o campo referente à mesma é o ativo.

A interface *i_ServiceContractInfoMgmt*, responsável pela obtenção e modificação dos dados dos contratos de serviço, possui mais de uma instância por objeto SUB. Esta interface é descrita nas recomendações TINA através da linguagem ODL, conforme pode ser visto na figura 3.25.

O **processo SUB_i_ServiceContractInfoMgmt** (figura 3.26), equivalente a esta interface, possui diversos procedimentos exportados. Eles são reponsáveis por lidar com os perfis de serviço definidos para o contrato em questão (*activateServiceProfiles, assignServiceProfile, deactivateServiceProfiles, defineServiceProfiles, deleteServiceProfiles, removeServiceProfiles*) e com a modificação e obtenção dos dados do contrato (*defineServiceContract, getServiceContractInfo*). Existem ainda dois procedimentos que são utilizados apenas pelos citados anteriormente para o cumprimento de sua tarefa, sendo assim não necessitam de ser exportados (*getServiceProfilePosition, validateServiceProfile*).

Além disso, este processo recebe sinais que indicam a alocação de recursos (*pCALL_allocateResources*) sempre que alguém solicita sua inicialização através do procedimento *init* do processo *SUB_i_InitialAccess* e a liberação de recursos (*pCALL_releaseResources*) através do procedimento *terminate* do processo *SUB_i_InitialAccess*. Estas solicitações são contabilizadas de forma que sempre que nenhum usuário necessite mais do processo, ele é destruído e isso é informado ao processo *SUB_Manager* (que gerencia as referências a estes processos) através do sinal *pCALL_interfaceDeleted*.

Também é mostrado na figura 3.26 o fato de que sempre que uma instância de um processo deste tipo é criada são obtidas informações sobre o contrato de serviço na base de dados por intermédio do processo *SUB_DBMSBroker*, através de chamada ao procedimento *DBgetServiceContract*. Assim os dados ficam armazenados localmente e não precisam ser buscados na base de dados sempre que solicitados. Além disso, sempre que algo é alterado por um dos procedimentos, o flag *changed* é ativado. O *timer DBMSupdate*, que é acionado a cada 600 segundos, verifica este flag e, caso algo tenha sido alterado, a base de dados é

```
interface i_ServiceContractMgmt{
  void activateServiceProfiles(in t_ServiceProfileIdList spIdList)
     raises(e_applicationError,
          e_unknownServiceProfile);
  void assignServiceProfile(in t_ServiceProfileId spId,
                       in t_SagIdList sagIdList,
                       in t_entityIdList saeIdList)
     raises(e_applicationError,
          e_unknownSAG,
          e_unknownSAE,
          e_unknownServiceProfile);
  void deactivateServiceProfiles(in t_ServiceProfileIdList spIdList)
     raises(e_applicationError,
          e_unknownServiceProfile);
  void defineServiceContract(in t_ServiceContract serviceContract,
                        out t_ServiceProfileIdList spIdList)
     raises(e_applicationError,
          e_invalidContractInfo,
          e_invalidSubscriptionProfile,
          e_invalidSAGServiceProfile);
  void defineServiceProfiles(in t_SubscriptionProfile subscriptionProfile,
                        in t_SagServiceProfileList sagServiceProfiles,
                        out t_ServiceProfileIdList spIdList)
     raises(e_applicationError,
          e_invalidSubscriptionProfile,
          e_invalidSAGServiceProfile);
  void deleteServiceProfiles(in t_ServiceProfileIdList spIdList)
     raises(e_applicationError,
          e_unknownServiceProfile);
  void getServiceContractInfo(in t_ServiceProfileIdList spIdList,
                         out t_ServiceContract serviceContract)
     raises(e_applicationError,
          e_unknownServiceProfile);
  void getServiceTemplate(out t_ServiceTemplate template)
     raises(e_applicationError);
  void listServiceProfiles(out t_ServiceProfileIdList spIdList)
     raises(e_applicationError);
  void removeServiceProfile(in t_ServiceProfileId spId,
                       in t_SagIdList sagIdList,
                       in t_entityIdList saeIdList)
     raises(e_applicationError,
          e_unknownSAG,
          e_unknownSAE,
          e_unknownServiceProfile);
};
```

Figura 3.25: Descrição em ODL da interface *i_ServiceContractInfoMgmt* (Objeto SUB)

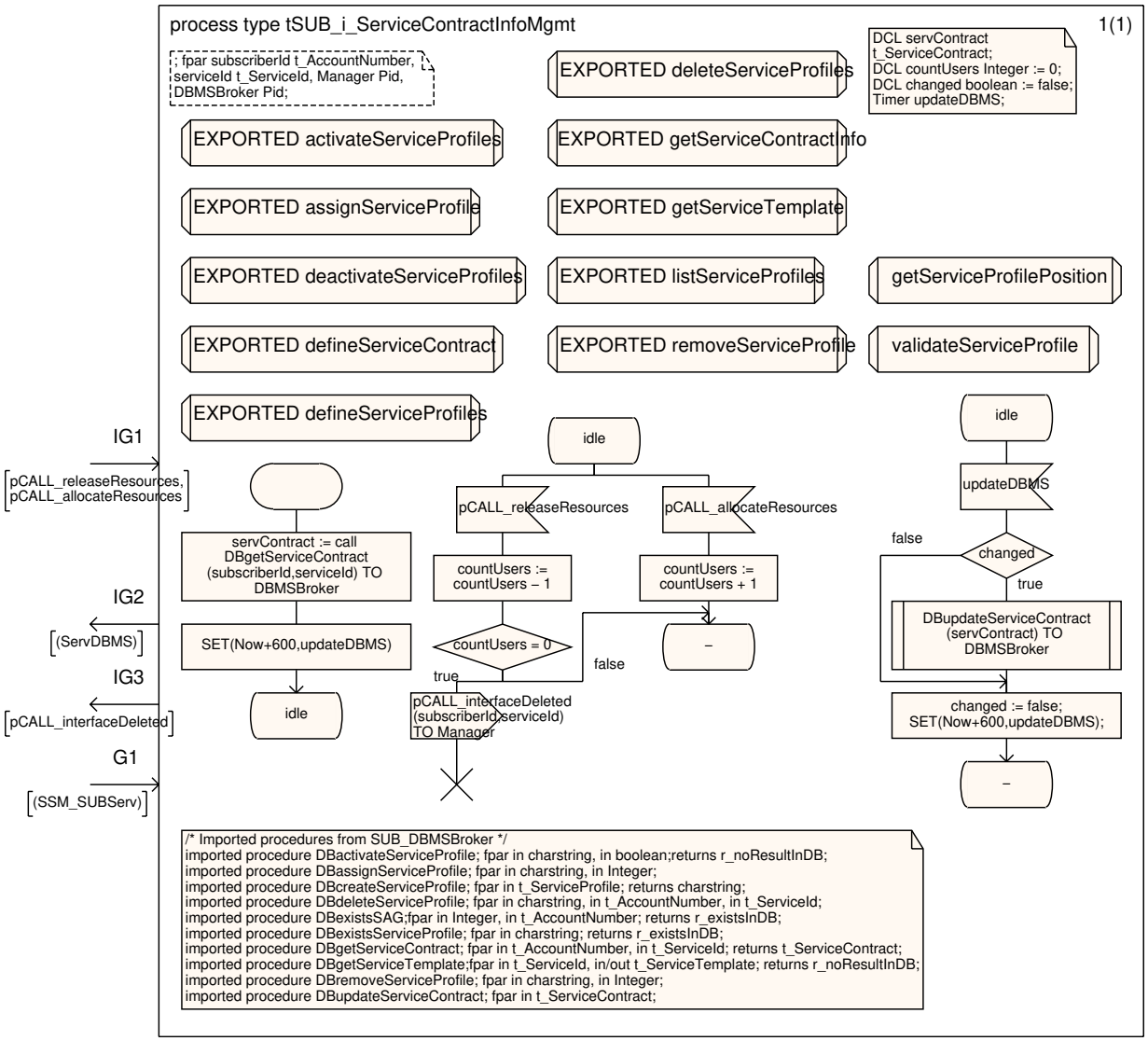


Figura 3.26: Especificação do processo *SUB_i_ServiceContractInfoMgmt* (Bloco SUB)

atualizada através da chamada ao procedimento *DBupdateServiceContract* exportado pelo processo *SUB_DBMSBroker*.

Já as especificações formais em SDL dos processos que representam as duas interfaces diretamente relacionadas à obtenção de informações acerca dos assinantes são exibidas a seguir nas figuras 3.27 e 3.28. Na figura 3.27 também é mostrada a descrição em ODL de sua interface (que serviu de base para a especificação em SDL).

O **processo SUB_i_SubscriberInfoQuery** (figura 3.27), que representa a interface *i_SubscriberInfoQuery*, possui quatro procedimentos exportados que são disponibilizados pa-

ra os agentes de usuário (namedUA e anonUA) por este processo e um que é disponibilizado internamente para a interface *i_Subscribe*. Dois deles estão descritos nas recomendações TINA: *getServiceProfiles* que obtém a lista de perfis de serviço de um dado usuário em relação a um certo serviço e *listServices* que lista os serviços contratados por um dado usuário. Além disso, o procedimento *checkUserInfo*, que é responsável por autenticar o usuário, comparando sua senha com a senha [1] armazenada na base de dados, é disponibilizado para o bloco namedUA. O procedimento *unregisterUA* é disponibilizado para os agentes de usuário (blocos namedUA e anonUA) para que os mesmos informem quando são destruídos. Isto é necessário pois este processo armazena uma tabela com todos os agentes de usuário e suas referências. Desta forma, sempre que um novo serviço é contratado, ou algum serviço é cancelado através do processo *SUB_i_Subscribe*, essa informação pode ser repassada ao agente de usuário interessado. Para isso, outro procedimento, o *notifyUA* é disponibilizado para o processo *SUB_i_Subscribe*.

O **processo SUB_i_SubscriberInfoMgmt** (figura 3.28), que representa a interface *i_SubscriberInfoMgmt*, possui diversos procedimentos exportados que são disponibilizados para o bloco SSMols. Eles são responsáveis por oferecer controle sobre os usuários registrados para um dado assinante (*assignSAEs, createSAEs, deleteSAEs, listSAEs, removeSAEs*), sobre os grupos de atribuição de serviço, ou SAGs, (*createSAGs, deleteSAGs, listSAGs*), sobre os dados do assinante (*getSubscriberInfo, setSubscriberInfo*) e para obtenção da lista de serviços contratados (*listSubscribedServices*).

O **processo SUB_i_InitialAccess** (figura 3.29) é utilizado por todos os blocos clientes para a realização do contato inicial com o bloco SUB. Sempre que necessário, este processo fornece aos clientes referências para os demais processos que compõem o bloco SUB.

Para que seja possível executar sua tarefa, são disponibilizados quatro procedimentos para os blocos clientes e um para o próprio bloco SUB. O **procedimento init** (figura 3.30) é responsável por fornecer referências aos outros processos e, inclusive, aos processos *SUB_i_ServiceContractInfoMgmt* relativos aos serviços contratados pelo assinante, as quais são obtidas junto ao processo *SUB_Manager*. O **procedimento terminate** (figura 3.31) é responsável por informar aos processos de contrato de serviço sobre o fim do uso dos mesmos, permitindo então que seus recursos sejam liberados caso eles não estejam mais sendo utilizados por ninguém. Por último, o **procedimento getSubscriberbyMainUser** (figura 3.32) é responsável por determinar qual é o assinante, dado um identificador de um usuário.

[1] Não foram levadas em consideração questões de segurança.Em uma implementação final, o uso de técnicas como a de criptografia devem ser utilizadas

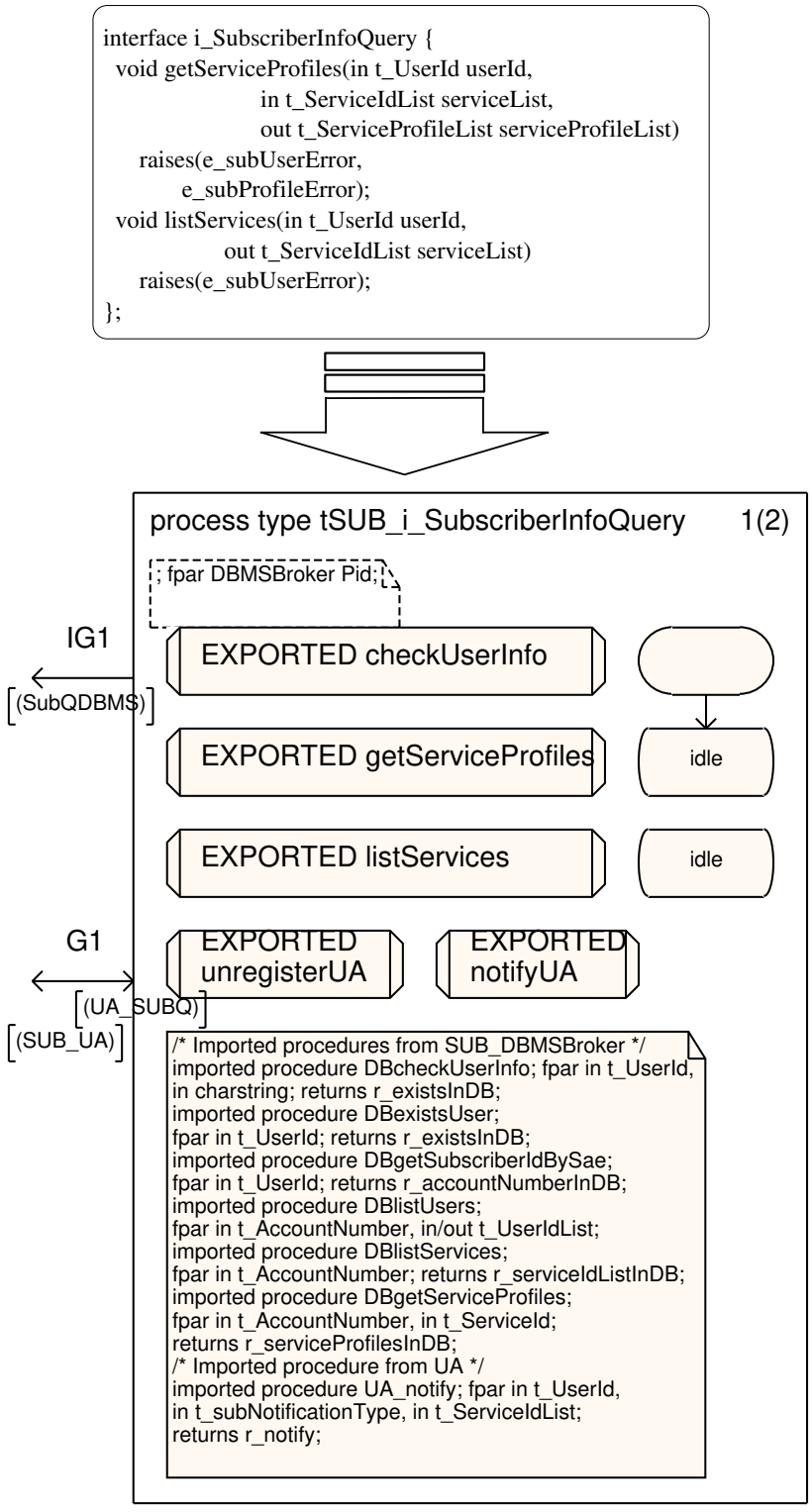


Figura 3.27: Interface *i_SubscriberInfoQuery* em ODL e em SDL (Bloco SUB)

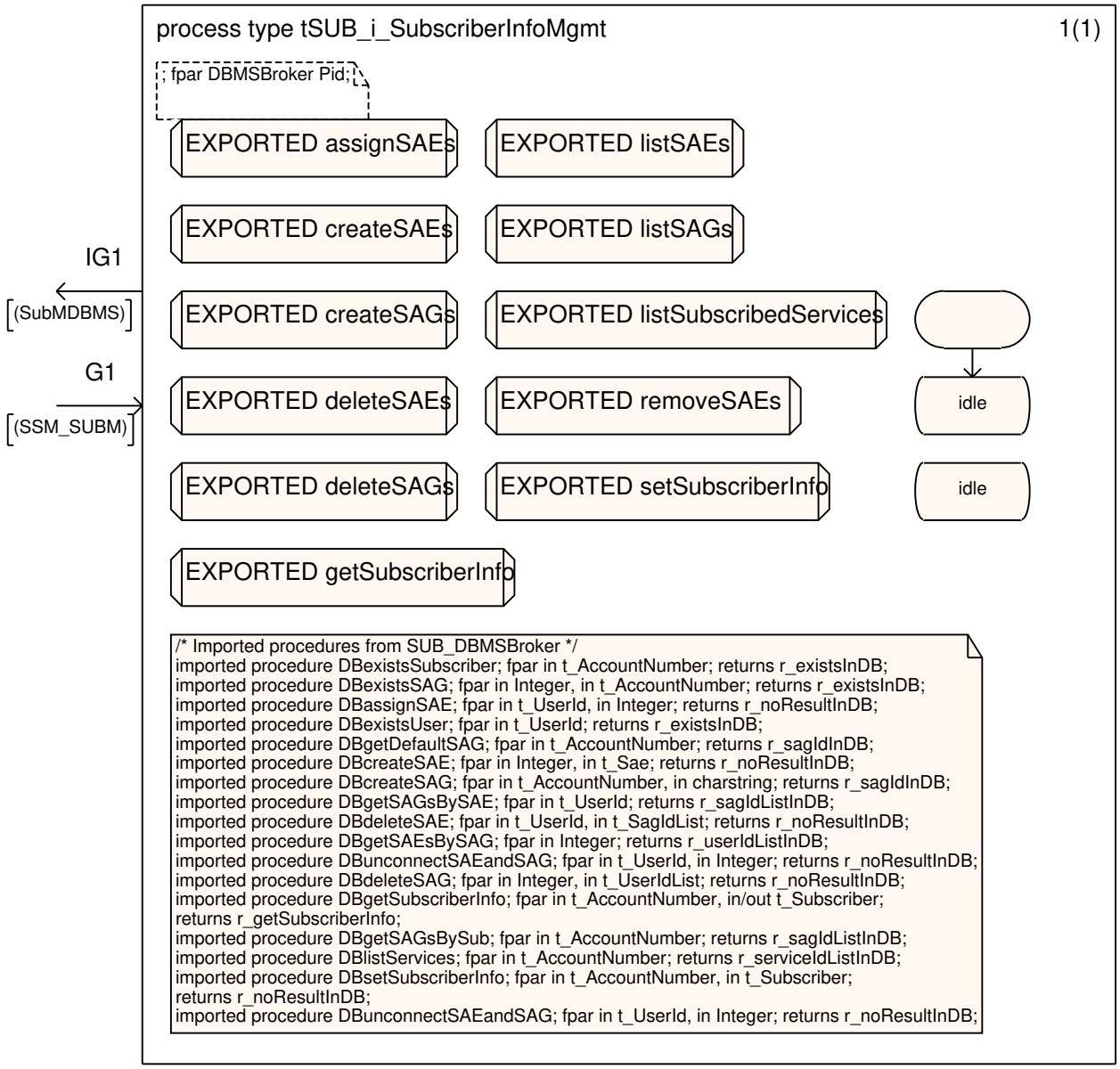


Figura 3.28: Especificação do processo *i_SubscriberInfoMgmt* (Bloco SUB)

Este último procedimento cria um conceito introduzido por este trabalho, que é o de "usuário principal". Conforme indicado pela figura 2.12, um assinante pode possuir diversos usuários, entretanto, apenas um destes usuários é considerado o usuário principal. Somente este usuário pode utilizar o serviço de subscrição, que possibilita a modificação de informações acerca dos serviços assinados por todos os usuários relacionados a um assinante. Uma empresa que contrate serviços de um provedor, por exemplo, não desejaria que qualquer um de seus

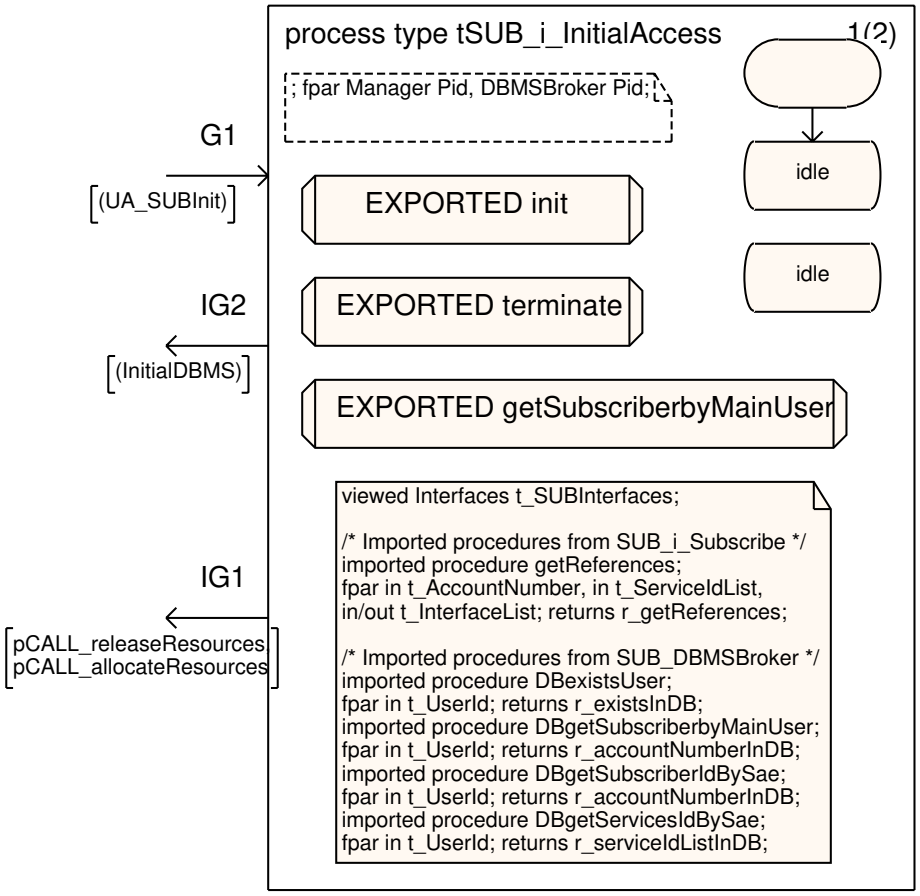


Figura 3.29: Especificação em SDL do processo *SUB_i_InitialAccess* (Bloco SUB)

funcionários pudesse alterar os contratos de serviço firmados. Desta forma, apenas um usuário (o encarregado pelos contratos de serviço) teria acesso a essa possibilidade.

Assim sendo, o procedimento *getSubscriberbyMainUser* retorna o identificador do assinante apenas para um usuário principal. Caso contrário, uma exceção é gerada.

Finalmente, o processo *SUB_i_Subscribe* (figura 3.33), que representa a interface *i_Subscribe*, é composto por uma série de procedimentos exportados. Eles são responsáveis pela inscrição de novos assinantes (*subscribe*), contratação de serviços (*contractServices*), cancelamento de serviços contratados (*unsubscribe*), listagem de assinantes (*listSubscribers*), listagem de serviços (*listAllServices*) e obtenção de referências às interfaces de contratos de serviço (*getReferences*).

Também faz parte do processo *SUB_i_Subscribe* o procedimento exportado *getService*, que tem como objetivo obter a descrição de um serviço dado o seu identificador. Trata-se de um procedimento bastante útil apesar de não ter sido proposto pelas recomendações TINA. Os

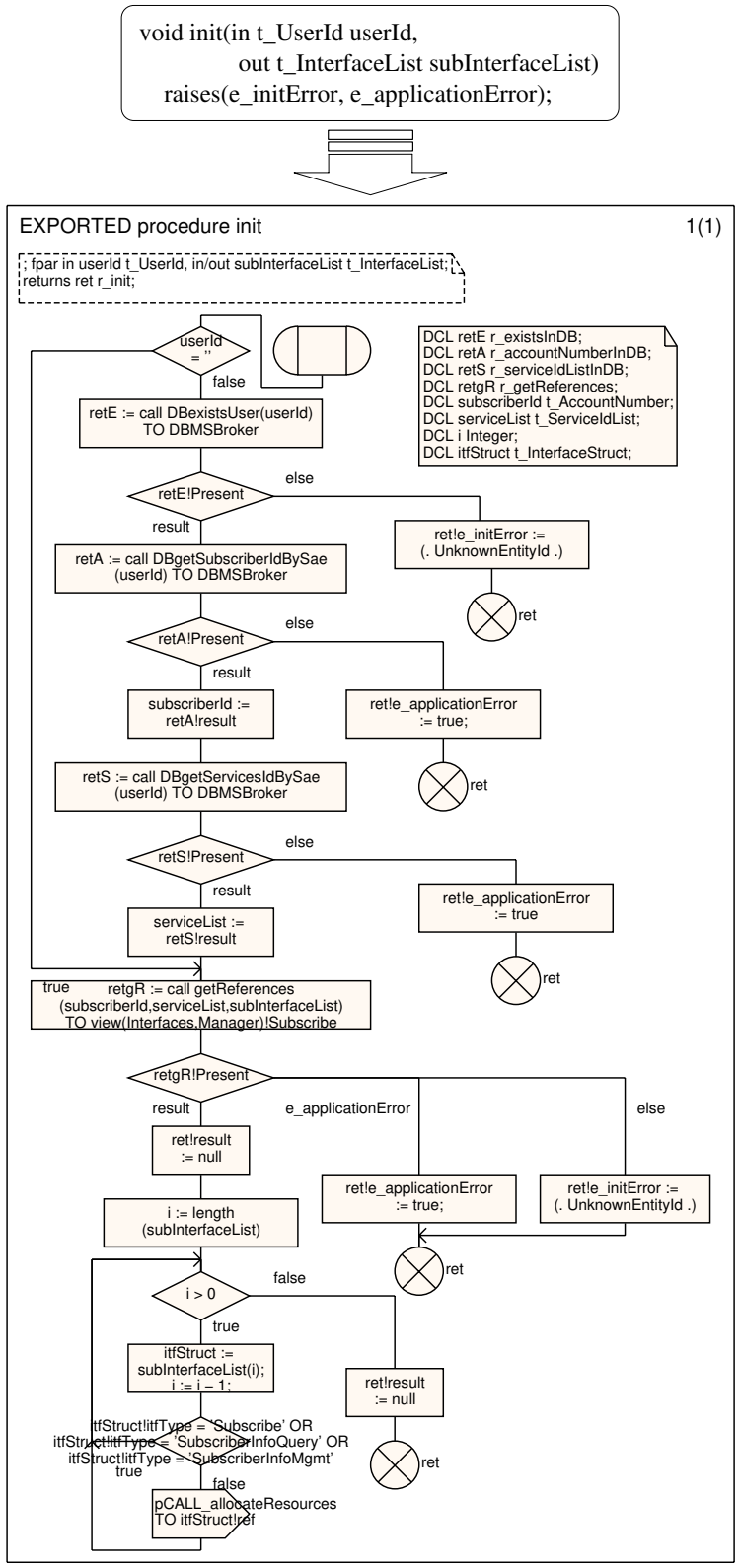


Figura 3.30: Procedimento *init* em ODL e em SDL (Processo *SUB_i_InitialAccess* / Bloco SUB)

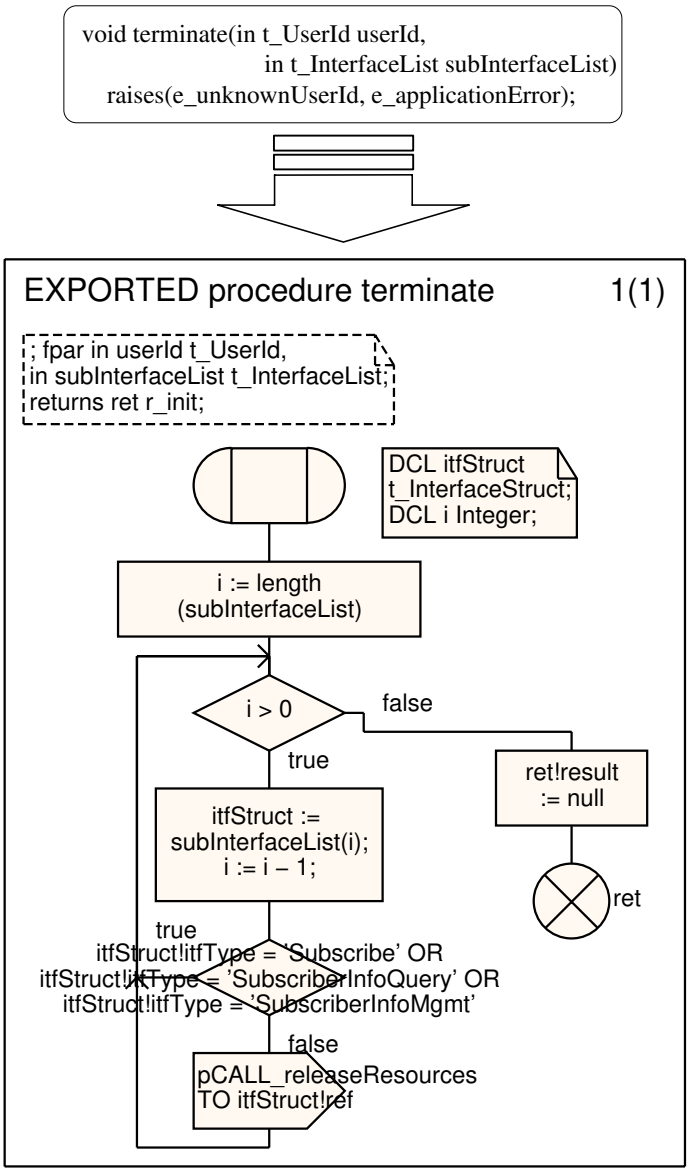


Figura 3.31: Procedimento *terminate* em ODL e em SDL (Processo *SUB_i_InitialAccess* / Bloco SUB)

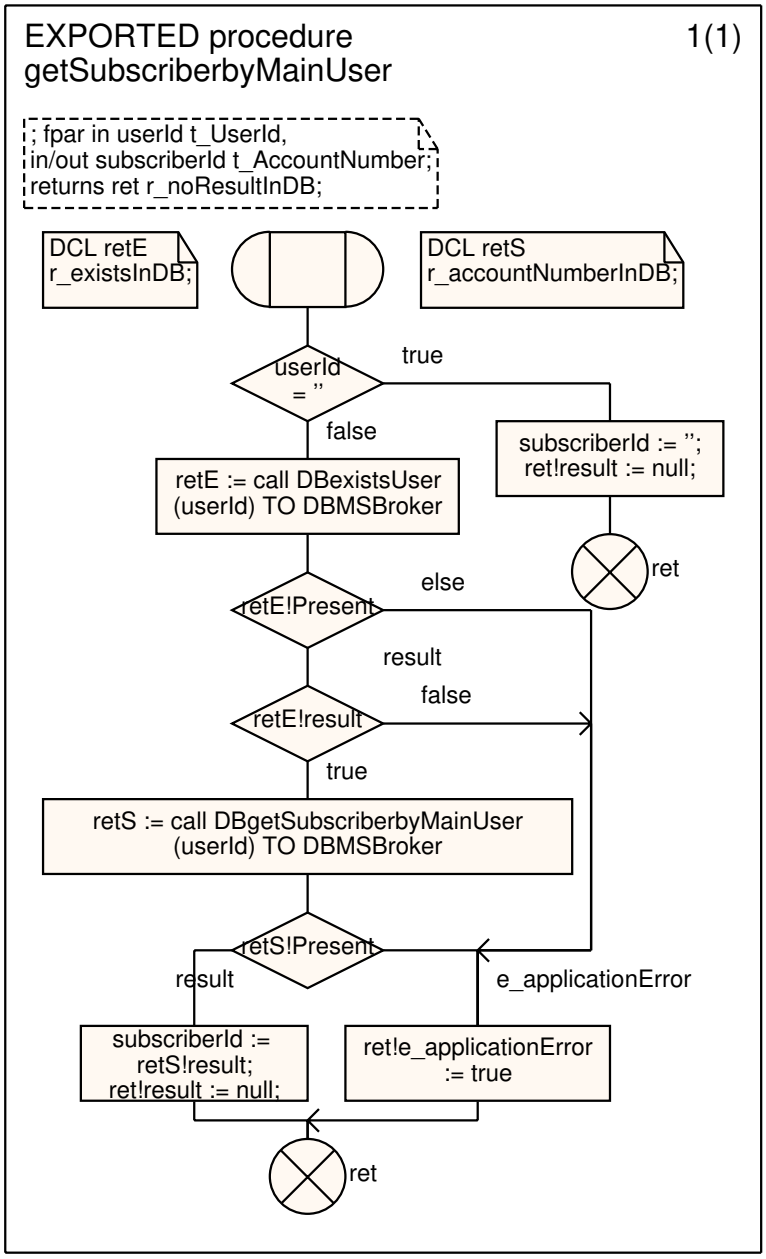


Figura 3.32: Procedimento *getSubscriberbyMainUser* (Processo *SUB_i_InitialAccess* / Bloco SUB)

procedimentos que listam serviços, por exemplo, necessitam deste método para obter uma descrição dos serviços a partir de uma lista de identificadores de serviços. Desta forma, podem apresentar uma lista de serviços mais compreensível para o usuário.

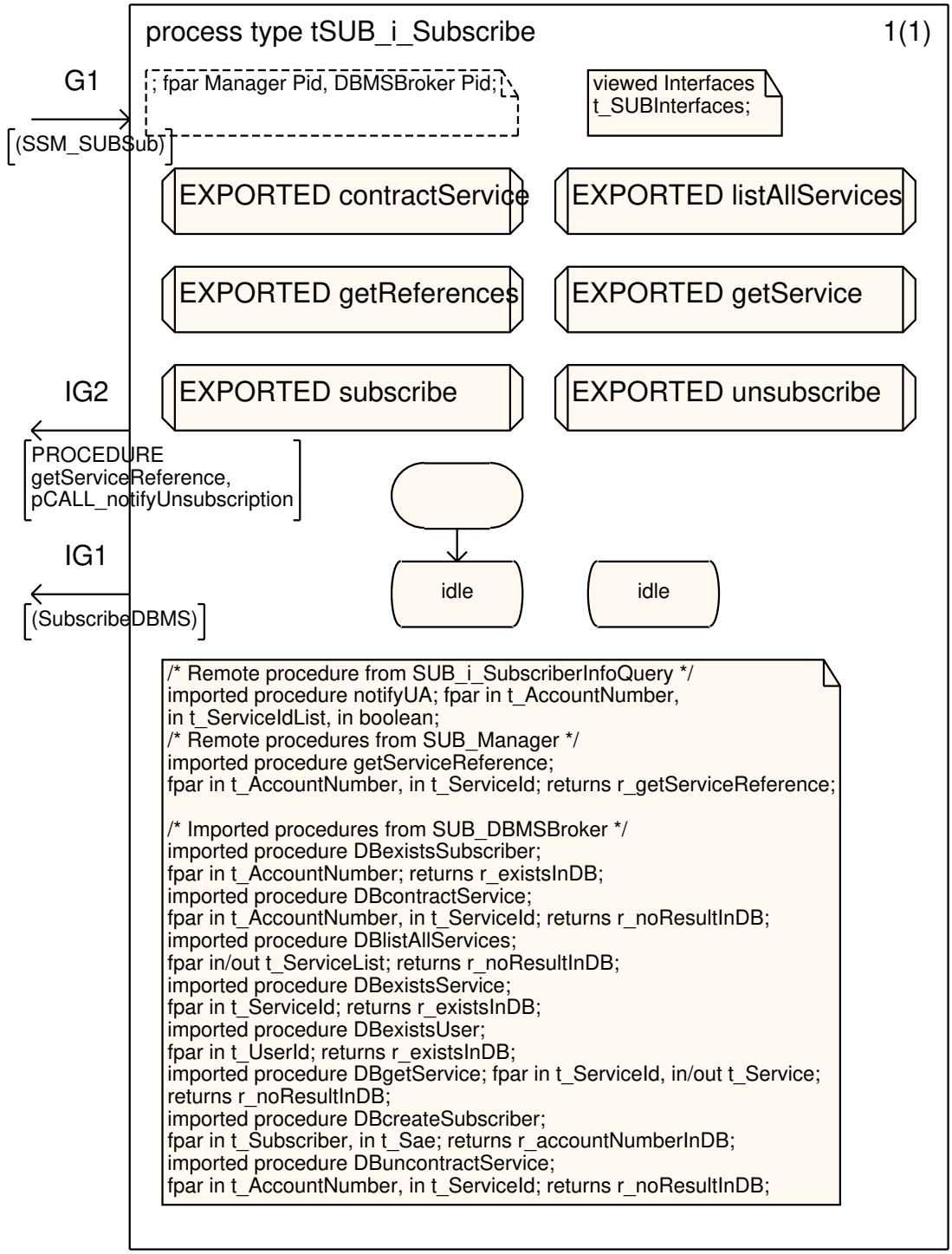


Figura 3.33: Especificação do processo *i_Subscribe* (Bloco SUB)

3.5 O uso de Packages

Todos os blocos integrantes da arquitetura de serviços TINA especificados foram dispostos em dois *packages*, como pode ser visto pela figura 3.34. O *package* é um agrupamento de especificações que podem ser reutilizadas por outros *packages* e por sistemas SDL para a sua composição. O *package BasicClasses* contém o bloco ancestral UA e é utilizado pelo *package AccessSession* onde são definidos os blocos relativos a subscrição e sessão de acesso (inclusive os blocos anonUA e namedUA que derivam de UA).

Esta organização em forma de *packages* é bastante interessante para permitir o reuso das especificações em uma possível expansão do sistema especificado.

Caso se deseje, por exemplo, ampliar as especificicações dos blocos, passando a tratar também as sessões de serviço, basta incluir no projeto os *packages* desenvolvidos e especificar blocos herdeiros destes que tenham mais funcionalidades (processos e procedimentos) capazes de tratar sessões de serviço, além da inclusão de novos blocos que se façam necessários.

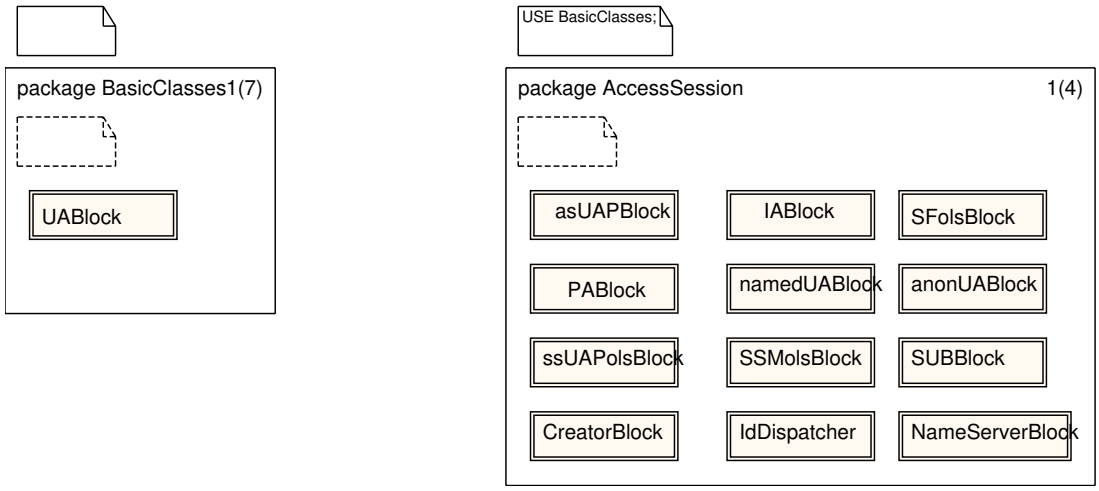


Figura 3.34: *Packages BasicClasses e AccessSession*

Capítulo 4

Resultados da Validação e das Simulações

A validação é um recurso disponibilizado pelo SDT para a facilitar a identificação de erros cometidos durante o processo de especificação do sistema. Através da validação, todos os estados possíveis do sistema especificado são percorridos, verificando em cada estado os sinais que podem ser enviados/recebidos e se algum deles não está sendo tratado adequadamente.

A simulação é utilizada para se verificar a dinâmica do sistema em casos especiais da especificação. É utilizada principalmente para se verificar, de forma mais detalhada, o comportamento do sistema em alguns casos críticos.

A validação e a simulação do sistema especificado neste trabalho foram realizadas através da utilização do *Validator* e do *Simulator* respectivamente. Ambos são módulos integrantes da ferramenta SDT (Apêndice A).

Na seção 4.1 é apresentado o resultado da validação do sistema e é comentado como o processo de validação pode ser utilizado para a identificação de erros de especificação. Na seção 4.2 são apresentadas algumas simulações visando exemplificar o funcionamento do sistema. Os resultados da simulação são mostrados através de gráficos MSC (*Message Sequence Charts*) e comparados com os diagramas de troca de mensagens equivalentes, apresentados nas recomendações TINA [35] como um guia do funcionamento dos objetos.

4.1 Validação

O módulo *Validator* do SDT responsável pela realização da validação das especificações em SDL disponibiliza três formas diferentes de se percorrer os estados de um sistema durante

o processo de validação: *Exaustive Exploration*, *Bit State Exploration* e *Random Walk*.

O *Exaustive Exploration* é o algoritmo utilizado quando se deseja que todos os caminhos possíveis do espaço de estados do sistema sejam percorridos. Trata-se de um algoritmo de exploração utilizado para sistemas pequenos. À medida que o sistema cresce e o número de estados aumenta significativamente, a utilização deste algoritmo se torna impraticável.

O *Bit State Exploration* funciona de forma semelhante, entretanto cada estado é representado através de uma forma mais compacta, utilizando uma estrutura de dados chamada *Hashtable* para gerenciar os estados percorridos. Trata-se de um algoritmo mais eficiente para sistemas de maior complexidade. Entretanto, se o número de estados do sistema for realmente grande, este método pode também ser ineficaz.

O *Random Walk* é o mais indicado para sistemas de maior porte, como é o caso do sistema especificado neste trabalho, onde o número de estados é bastante significativo. Este algoritmo percorre de forma aleatória os ramos da árvore com os estados possíveis do sistema, simplificando bastante o gerenciamento de estados percorridos e não percorridos.

Para a execução da validação utilizando o método *Random Walk*, alguns parâmetros são necessários:

- *Depth*: Indica a profundidade máxima da árvore de estados atingida pelo algoritmo;
- *Repetitions*: Indica o número de vezes que o algoritmo repete a tomada de decisão sobre qual caminho seguir em cada ponto da árvore de estados.

4.1.1 A validação do sistema SDL especificado

Para o sistema SDL desenvolvido (modelo de subscrição e sessão de acesso), os parâmetros utilizados durante a simulação foram: $Depth = 1000$ e $Repetitions = 1000$. Alguns parâmetros de menor valor foram testados, porém à medida que eram aumentados, o número de estados alcançados também aumentava. A partir destes valores ($Depth = 1000$ e $Repetitions = 1000$), mesmo que eles sejam aumentados, o mesmo número de estados continua sendo percorrido. Foram obtidos os resultados apresentados na figura 4.1.

O resultado mostrado na figura 4.1 é composto de diversas informações. *Number of Reports* indica o número de erros encontrados (sinais não tratados, referências a processos inexistentes, etc), que no caso do sistema especificado é zero (após a correção dos erros detectados por validações anteriores), indicando a ausência de erros de especificação. *Generated States* indica o número de estados gerados na árvore de estados explorada pelo algoritmo.

```
** Random Walk **
Number of reports: 0
Generated States: 2609200
Max Depth: 1000
Min Depth: 1000
Symbol coverage: 93.52
```

Figura 4.1: Resultado da Validação (*Random Walk*)

Max Depth e *Min Depth* indicam respectivamente a menor e a maior profundidade atingida na árvore de estados percorrida. O fato dos dois números serem iguais ao parâmetro *Depth* utilizado, indica que todos os ramos da árvore de estados foram completamente percorridos, não havendo nenhum caso em que a exploração parou em uma profundidade menor. *Symbol Coverage* informa a porcentagem de símbolos SDL percorridos pelo processo de validação.

	Total de Símbolos	Símbolos alcançados	Símbolos não alcançados
Bloco SUB	1298	1182 (91,06%)	116 (8,94%)
Bloco UA	127	116 (91,34%)	11 (8,66%)
Bloco SSMols	397	372 (93,70%)	25 (6,30%)
Bloco ssUAPols	354	349 (98,59%)	5 (1,41%)
Bloco PA	112	111 (99,01%)	1 (0,99%)
Bloco namedUA	27	27 (100%)	0 (0%)
Bloco anonUA	23	23 (100%)	0 (0%)
Bloco asUAP	96	96 (100%)	0 (0%)
Bloco IA	67	67 (100%)	0 (0%)
Bloco SFols	28	28 (100%)	0 (0%)
Bloco Creator	10	10 (100%)	0 (0%)
Bloco IdDispatcher	5	5 (100%)	0 (0%)
Bloco NameServer	11	11 (100%)	0 (0%)
Total (Sistema)	2555	2397 (93,52%)	158 (6,48%)

Tabela 4.1: Número de símbolos alcançados pela validação por bloco

A tabela 4.1 mostra que o bloco SUB possui uma grande importância dentro do sistema especificado, pois nele está aproximadamente a metade dos símbolos de todo o sistema. É nele também onde se encontra a maior porcentagem (8,94%) de símbolos não alcançados pelo processo de validação. Isto ocorre devido ao fato de serem tratadas algumas exceções que nunca ocorrem no modelo SDL, mas que podem ocorrer em uma eventual implementação, caso objetos desenvolvidos por terceiros (que não se comportem adequadamente) se comuni-

quem com o SUB. São tratados em todos os procedimentos exportados disponibilizados pelos processos de SUB, os casos em que se deseja realizar alguma operação relativa a um assinante e o identificador do assinante passado como parâmetro é inexistente. Este caso nunca ocorre na especificação, uma vez que todas as interações com o SUB são realizadas através do bloco SSMols e, como existe uma instância deste bloco para cada assinante, nunca um assinante é solicitado ao SUB uma operação para um identificador de assinante inexistente. Outros tipos de verificação desta mesma natureza são realizadas nos blocos SSMols, PA e ssUAPols.

No bloco UA, a herança é responsável pelo fato de alguns de seus símbolos (11) não serem alcançados. Durante a especialização do bloco UA em seus descendentes namedUA e anonUA, alguns símbolos declarados como *VIRTUAL* em UA são redefinidos. Desta forma, estes símbolos nunca são alcançados pelo processo de validação, uma vez que não há nenhuma instância de UA, ele serve apenas como base para as especificações de namedUA e anonUA.

Os símbolos não alcançados podem ser identificados através do *Coverage Viewer*, que exibe uma árvore com a hierarquia de blocos, processos, procedimentos e símbolos do sistema SDL, mostrando o que foi e o que não foi testado pela validação. A figura 4.2 mostra a parte do resultado do *Coverage Viewer* para a validação do sistema especificado neste trabalho relativa ao bloco SUB. Os processos e procedimentos preenchidos com cinza indicam que foram completamente testados, enquanto os parcialmente preenchidos indicam que alguns de seus símbolos não foram alcançados. Este símbolos não alcançados podem ser identificados na especificação em SDL para que se possa verificar se há algo de errado ou se eles tratam casos inatingíveis pela validação (como os dois exemplificados).

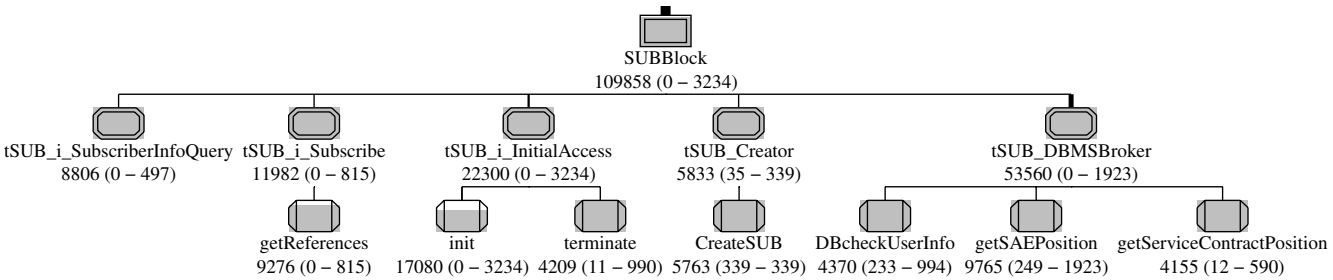


Figura 4.2: *Coverage Viewer*

Para que esta porcentagem de símbolos (93,52%) fosse atingida durante a validação, foi utilizado um recurso de não-determinismo da linguagem SDL para atribuir valores de teste aos parâmetros de entrada dos sinais que o usuário envia ao sistema. Desta forma, diversas possibilidades diferentes foram testadas. A decisão *ANY* (figura 4.3) possui esse caráter não-

determinístico desejado, ou seja, cada caminho da decisão pode ser percorrido com mesma probabilidade (para n caminhos, cada caminho pode ser percorrido com probabilidade $1/n$). Em cada caminho da decisão é atribuído um valor para os parâmetros do sinal enviado, assim, cada vez que a validação passa por essa decisão, um caminho é escolhido e um valor é testado. Sendo o número de passagens pela decisão (indicado pelo parâmetro *Repetitions*) for grande o suficiente, a validação passa por todas as possibilidades, testando o comportamento do sistema para todos os casos. Através do *Coverage Viewer* pode-se verificar se os símbolos relativos à todas as possibilidades foram alcançados.

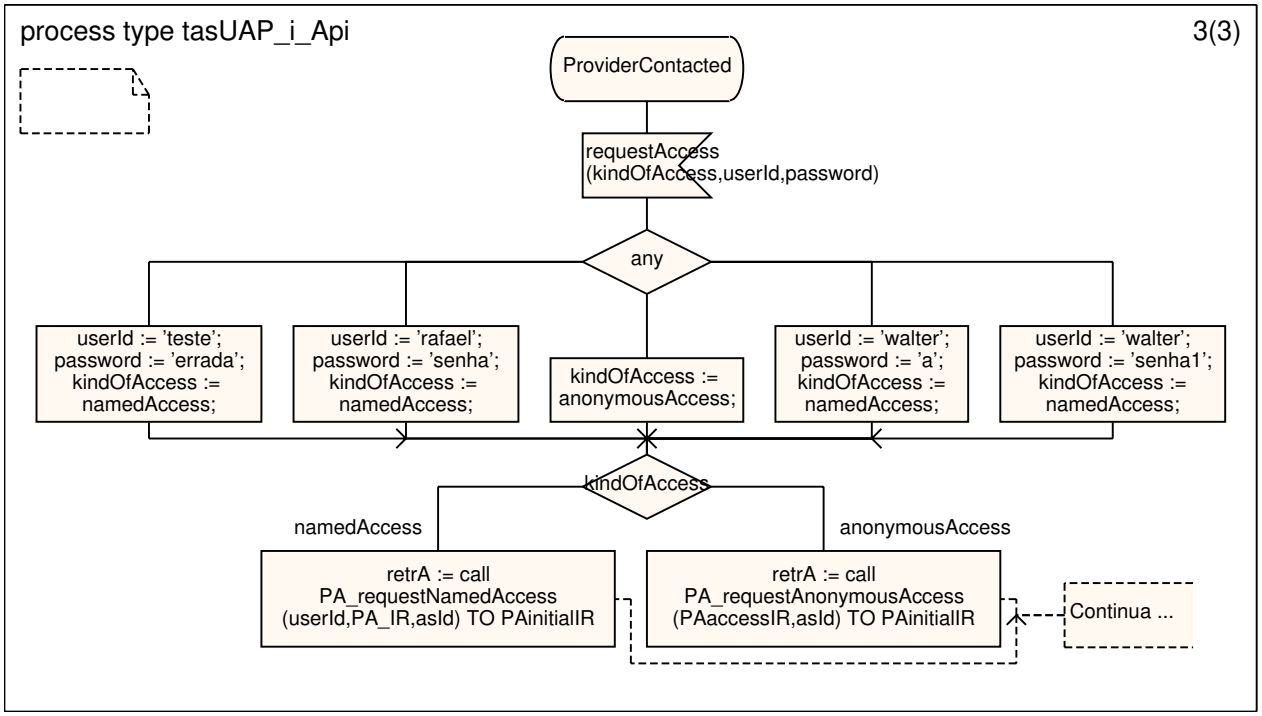


Figura 4.3: Uso de *ANY* para definição de parâmetros para validação

No exemplo mostrado na figura 4.3, todos os casos possíveis de solicitação de acesso ao provedor são testados. No primeiro caso o usuário (teste) não existe na base de dados, no segundo caso o usuário (rafael) existe, é um "usuário principal" e a sua senha está correta, no terceiro caso, é testado o acesso anônimo, no quarto caso, o usuário (walter) existe mas a sua senha está incorreta e no último caso, o usuário (walter) existe, sua senha está correta, porém não é um "usuário principal", ou seja, não tem acesso ao serviço de subscrição.

Este artifício de não-determinismo foi utilizado em todos os sinais provenientes do ambiente (usuário) que podem receber parâmetros apenas durante o processo de validação. Estes sinais estão presentes nos processos *asUAP_i_Api* (bloco asUAP) e *ssUAPols_i_Api* (bloco

ssUAPols), responsáveis por prover interfaces de acesso ao provedor e ao serviço de subscrição respectivamente.

4.1.2 Detecção de erros de especificação através da validação

Através do processo de validação, pode-se detectar erros de especificação que seriam dificilmente encontrados manualmente. Através do uso do *Coverage Viewer*, pode-se identificar os símbolos não alcançados pela validação e, assim, tem-se uma noção mais precisa dos pontos onde podem haver erros.

A figura 4.4 exemplifica a detecção de um erro em uma comparação realizada no procedimento *UA_endAccessSession* do bloco *UA*. O erro pode ser encontrado após a execução do *Validator*, analisando-se o resultado apresentado pelo *Coverage Viewer*. Nenhum dos símbolos abaixo de um dos ramos da comparação foi alcançado, o que sugere que a comparação esteja errada. Após a correção, o sistema foi re-validado e todos os símbolos foram alcançados.

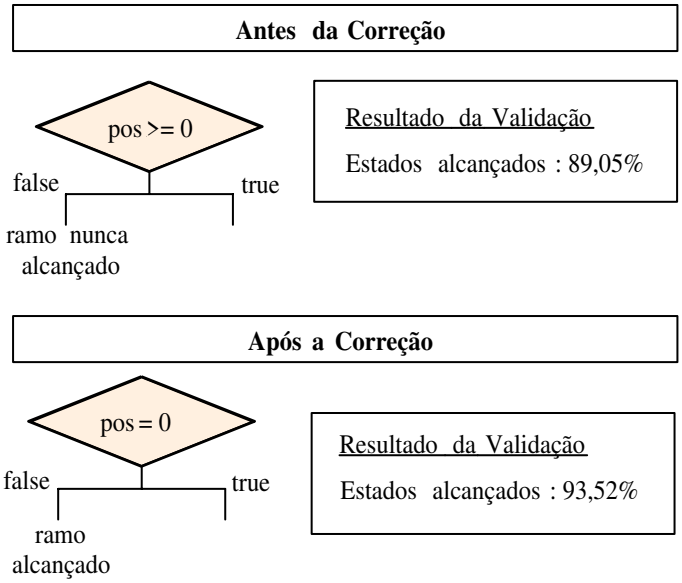


Figura 4.4: Detecção de erro de comparação através da validação

Outros tipos de erros também podem ser detectados durante a validação. Caso seja, por exemplo, enviado um sinal inesperado a um processo, ou seja, que o sinal não seja tratado pelo processo no estado em que ele se encontra, a validação informa a ocorrência de um erro de "consumação implícita de sinal" (*implicit signal consumption*) indicando que a especificação

do processo deve ser revista.

Um outro erro bastante comum, detectado pela validação, é o envio de sinais (ou chamada de procedimentos exportados) a processos que não existam mais. Caso uma referência a um processo que já foi destruído continue sendo armazenada em outro processo, um sinal pode ser enviado para uma referência (Pid) inexistente. Para evitar isso, sempre que um processo é destruído todos os processos que possam possuir referências para ele devem ser notificados. A figura 4.5 mostra o envio de um sinal a um processo já destruído.

Após a correção dos erros detectados pelo processo de validação, o sistema ainda pode ser submetido a testes mais detalhados visando a verificação da existência de erros de lógica, que acarretem o funcionamento inadequado de algum procedimento especificado.

Pode-se então utilizar do recurso de simulação para a realização de testes de eventuais casos críticos e das principais funcionalidades do sistema, analisando-se de forma mais criteriosa cada passo de sua execução. O uso da ferramenta de testes baseada na notação TTCN (*Tree and Tabular Combined Notation*), parte integrante do SDT, também pode ser utilizada no intuito de automatizar o processo de testes, permitindo uma abrangência maior, entretanto esta ferramenta não se encontra disponível no pacote SDT utilizado para o desenvolvimento deste trabalho.

4.2 Simulações

Através das simulações pode-se verificar as diversas funcionalidades do sistema especificado em SDL, na tentativa de encontrar algum erro de lógica que tenha passado despercebido durante o processo de especificação. Através do uso de MSCs (*Message Sequence Charts*) é possível visualizar de forma gráfica o resultado do processo de simulação, permitindo uma visão bastante detalhada do comportamento do sistema.

Com o objetivo de exemplificar o processo das simulações realizadas, são apresentados a seguir alguns dos diagramas MSCs referentes aos resultados das simulações dos casos de contratação de um novo serviço, de criação de um novo usuário e de ocorrência de uma exceção ao se tentar obter um dado junto à base de dados de contratos de serviço. Os dois primeiros diagramas (figuras 4.7 e 4.9) são comparados com diagramas de troca de sinais simplificados (figuras 4.6 e 4.8) disponibilizados pelas recomendações TINA [35]. Estes diagramas são disponbilizados pelo TINA-C como forma de facilitar a compreensão do funcionamento dos objetos e, portanto, devem estar de acordo com os diagramas MSC gerados durante o processo de simulação. O terceiro MSC não possui diagrama equivalente disponibilizado pelo TINA-C.

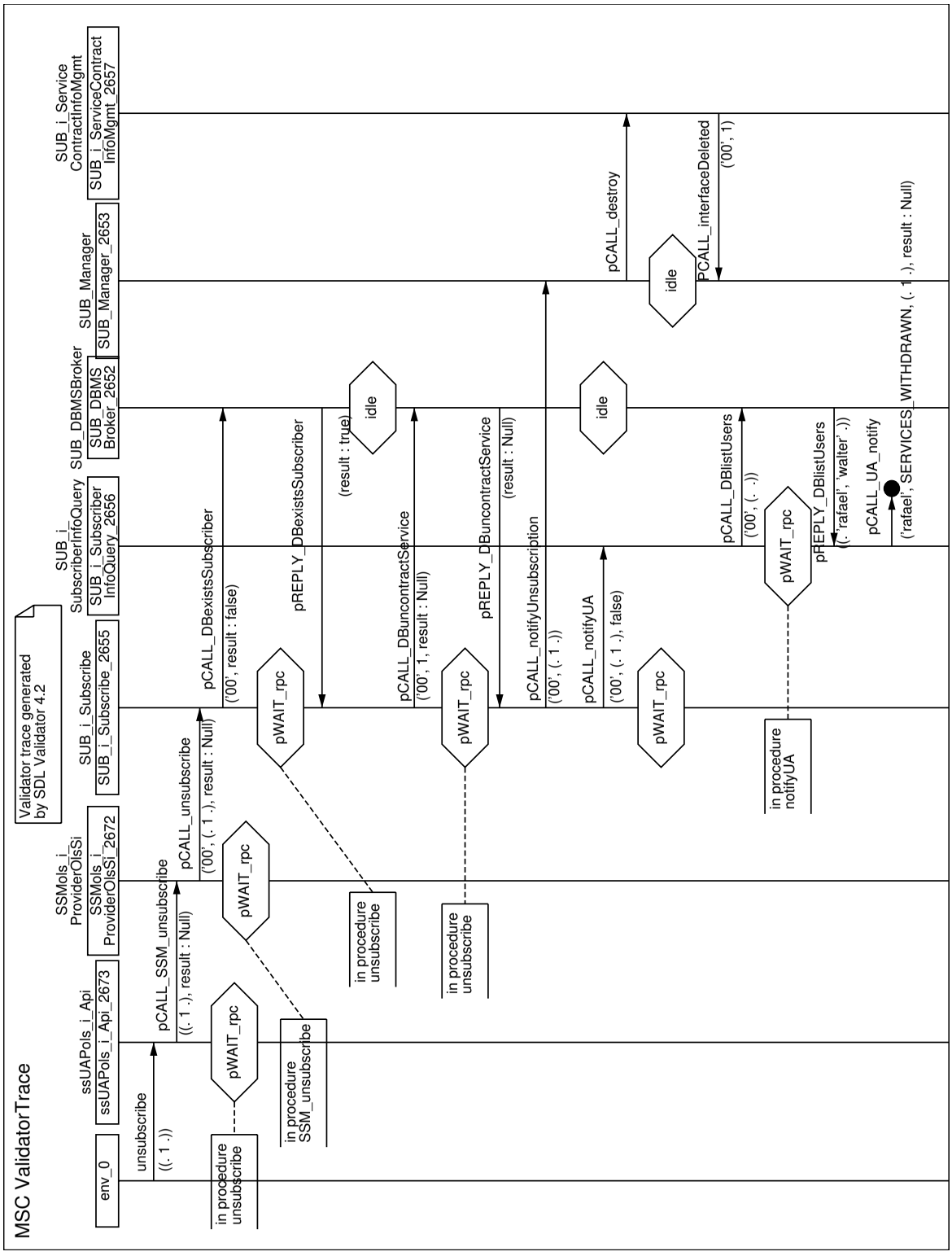


Figura 4.5: Detecção de referência a processo inexistente através da validação

As comunicações entre objetos necessárias para a contratação de um serviço são disponibilizadas pelo TINA-C sob a forma do diagrama de troca de mensagens mostrado na figura 4.6. É exibida de forma simplificada a solicitação de contratação de um novo serviço de ssUAPols para SSMols através de chamada ao método *contractService* (passo 1) e o repasse dessa solicitação para a interface *i_Subscribe* do objeto SUB (passo 2), seguido da solicitação do template de serviço através do método *getServiceTemplate* (passo 3).

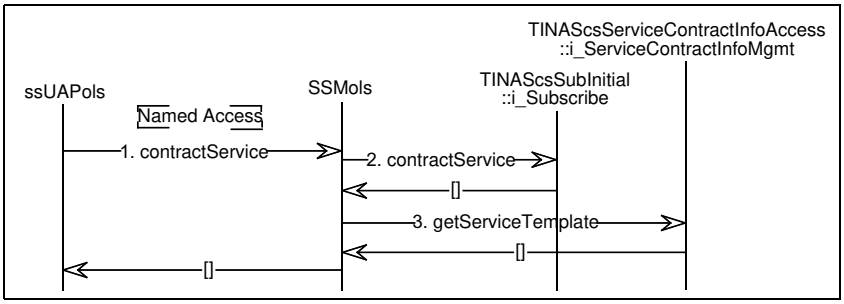


Figura 4.6: Diagrama de contratação de serviço disponibilizado pelo TINA-C [35]

Baseado neste comportamento (troca de sinais) esperado para a contratação de um novo serviço, é mostrada na figura 4.7 o resultado da simulação desta funcionalidade no sistema especificado em SDL. Assume-se aqui que o usuário já se autenticou junto ao provedor e obteve acesso ao serviço de subscrição.

No MSC da figura 4.7, uma chamada a um procedimento exportado é representado pelo envio de um sinal com o mesmo nome do procedimento, precedido por *pCALL_*. A resposta do procedimento é representada por um sinal de mesmo nome do procedimento, precedido por *pREPLY_*. A chamada ao procedimento é síncrona, tendo representada por um par de sinais apenas no MSC gerado pelo SDT para que seja mais fácil a visualização da dinâmica do sistema. Além disso, o usuário é representado por um processo chamado *env_0*, assim todos os sinais trocados com *env_0* são, na realidade, interações do usuário com o sistema especificado.

Os parâmetros dos sinais são indicados sob a forma de uma lista que aparece abaixo do nome do sinal e, no caso de um procedimento exportado, o seu resultado é indicado como o último dos parâmetros. Os tipos de dados dos parâmetros passados pelos sinais são represenados através de algumas notações particulares. Uma lista é representada através do operador (. .), sendo os elementos da lista separados por vírgulas. No MSC apresentado, o parâmetro do sinal *pREPLY_DBlistUsers* é uma lista contendo as strings 'rafael' e 'walter'. Uma estrutura (*struct*) é representada através do mesmo operador da lista, sendo os

componentes desta estrutura também separados por vírgulas. O primeiro parâmetro do sinal *pREPLY_getServiceTemplate* é uma estrutura cujo primeiro componente é o valor 0, o segundo é uma string vazia (''), e o terceiro é uma outra estrutura. Um parâmetro do tipo *choice* é representado pelo nome do componente ativado seguido do valor deste componente (*result:Null*, para indicar o resultado nulo em um procedimento que possa gerar exceções, por exemplo). Por último, referências a processos (Pids) são representadas através do nome do processo seguido do número da instância (*SUB_Manager:1* representa a primeira instância do processo *SUB_Manager*, por exemplo).

A simulação do processo de contratação de um novo serviço se inicia com a solicitação do usuário para a contratação do serviço de identificador 1 através do sinal *contractService* enviado ao processo *ssUAPols_i_Api*. Este processo chama então o procedimento *SSM_contractService* do processo *SSMols_i_ProviderOlsSi* presente no domínio do provedor. Esta solicitação é repassada ao processo *SUB_i_Subscribe* através do procedimento *contractService*. Verifica-se junto ao *SUB_i_DMBSBroker* a existência do assinante (através de *DBexistsSubscriber*) e então solicita-se a contratação através do procedimento *DBcontractService*. É, então, chamado o procedimento *notifyUA* de *SUB_i_SubscriberInfoQuery* que é responsável por notificar os agentes de usuário (namedUA) sobre a contratação do novo serviço. Para isso, é necessário obter-se a lista de usuários do assinante em questão, o que é feito através de uma chamada ao procedimento *DBlistUsers*. O namedUA é então informado através do procedimento *UA_notify* (caso haja mais de um namedUA ativo, ou seja, mais de um usuário do mesmo assinante, todos eles são informados).

Em seguida, o processo *SUB_i_Subscribe* solicita ao *SUB_Manager* referência para o processo responsável pelo contrato de serviço criado, através do procedimento *getReferences*. *SUB_Manager* verifica a existência deste contrato através do procedimento *DBexistsServiceContract* e, como se trata de um contrato recém-criado, cria uma nova instância do processo *SUB_i_ServiceContractInfoMgmt* para gerenciar as informações deste contrato. A referência para este novo processo é então retornada a *SUB_i_Subscribe* e repassada a SSMols_i_ProviderOlsSi através do resultado de *contractService*. O sucesso da contratação é informado ao *ssUAPols_i_Api* e uma mensagem de sucesso é exibida ao usuário.

Paralelamente a isso, o processo *SUB_i_ServiceContractInfoMgmt* recém-criado obtém informações sobre o contrato de serviço que irá gerenciar através dos procedimentos *DBgetServiceContract* e *DBgetServiceTemplate* do processo *SUB_DBMSBroker*. Além disso, o namedUA notificado obtém informações sobre o novo serviço contratado através do procedimento *getService* de *SUB_i_Subscribe*. Este procedimento verifica a existência do serviço

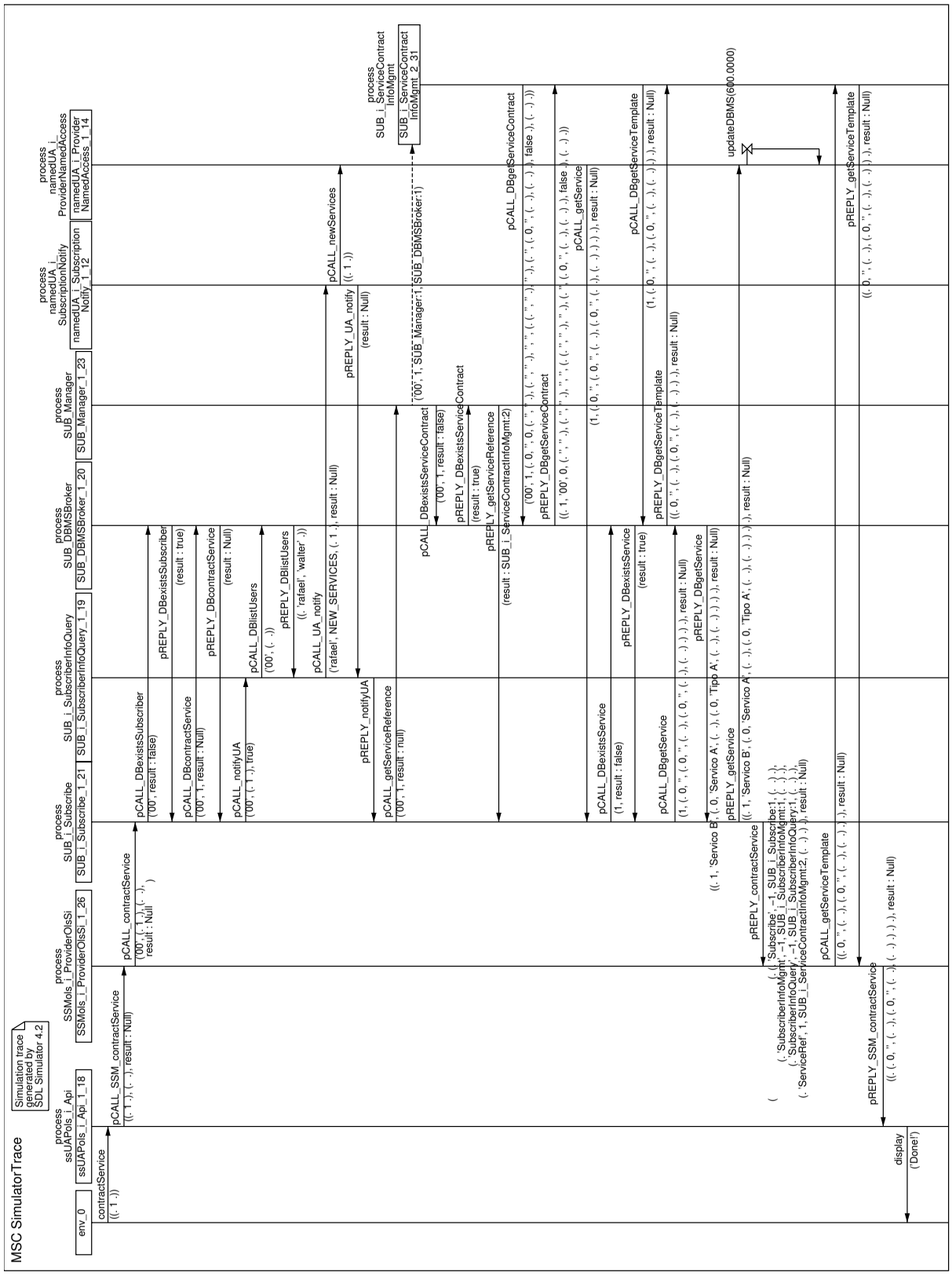


Figura 4.7: Contratação de serviço

através de *DBexistsService* e obtém suas informações através de *DBgetService*.

O processo *SSMols_i_ProviderOlsSi* obtém então o template do serviço contratado junto ao processo *SUB_i_ServiceContractInfoMgmt* criado. O sucesso da contratação e o template do serviço são informados ao *ssUAPols_i_Api* e uma mensagem de sucesso é exibida ao usuário.

Comparando-se o diagrama disponibilizado pelo TINA-C (figura 4.6) e o MSC obtido como resultado da simulação (figura 4.7), é possível notar-se que elas são equivalentes, tendo como diferença principal o fato do MSCs gerado pelo processo de simulação ser mais detalhado, exibindo também as trocas de sinais entre os processos de um mesmo bloco.

A figura 4.8 mostra o diagrama de trocas de mensagens disponibilizado pelo TINA-C para a o processo de criação de um novo usuário para um dado assinante. Primeiramente, durante o processo de inicialização do serviço de subscrição, o USM/SSM_{ols} obtém referências às interfaces do objeto SUB (*getReferences*). É solicitada a criação de um novo usuário através de uma chamada ao método *createSAEs* de USM/SSM_{ols}. Essa solicitação é repassada ao objeto SUB através de sua interface *i_SubscriberInfoMgmt* (passo 2). O usuário é criado e uma instância de um namedUA para este usuário pode ser criada. Em seguida solicita-se a lista dos SAGs disponíveis através do método *listSAGs* (passo 3) e essa solicitação é repassada ao objeto SUB (passo 4). De posse da lista de SAGs, associase-se o usuário recém-criado a um destes SAGs através do método *assignSAEs* (passo 5) e essa solicitação é repassada ao objeto SUB (passo 6). Caso o namedUA tenha sido criado, ele é notificado dessa associação (passo 7).

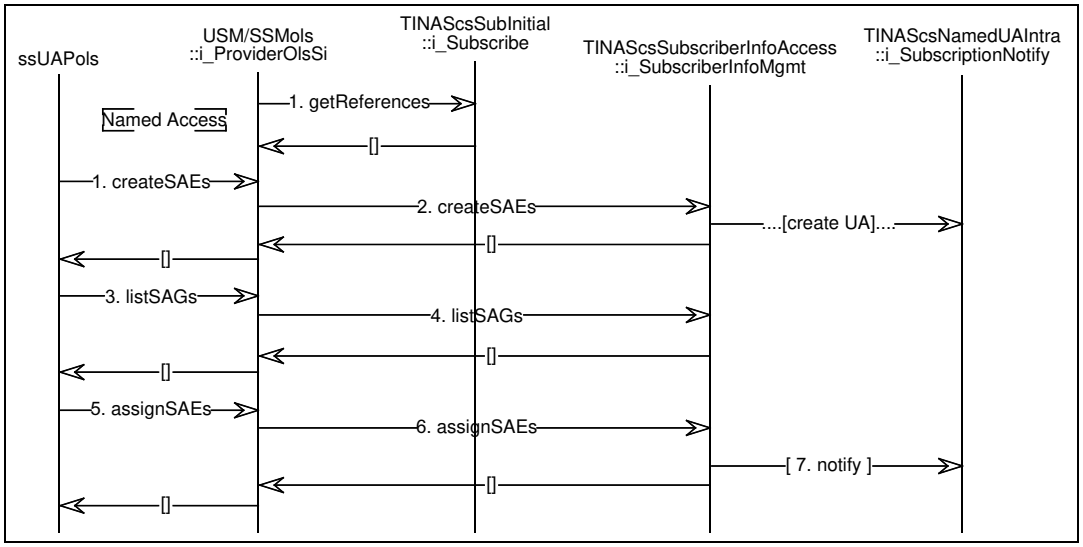


Figura 4.8: Diagrama de criação de um novo usuário disponibilizado pelo TINA-C [35]

A figura 4.9 mostra o resultado da simulação para o mesmo processo de criação de um novo usuário. Inicialmente solicita-se a criação de um novo usuário através do envio do sinal *createSAEs* para o processo *ssUAP_i_Api*. É então chamado o método *SSM_createSAEs* do processo *SSMols_i_ProviderOlsSi* (no domínio do provedor). Este processo repassa o pedido para o bloco SUB através do procedimento *createSAEs* e seu processo *SUB_i_SubscriberInfoMgmt*. É então verificado junto à base de dados, por intermédio do processo *SUB_Manager*, a existência do assinante em questão (*DBexistsSubscriber*), o SAG padrão do assinante (*DBgetDefaultSAG*) e a existência de algum outro usuário com o mesmo nome do que será criado (*DBexistsUser*). Só então é criado o usuário através do procedimento *DBcreateSAE*, associando o mesmo ao SAG padrão do assinante. Uma mensagem de sucesso é exibida ao usuário.

Além disso, pode-se associá-lo a outros SAGs. Para isso, primeiramente é solicitada uma lista dos SAGs do assinante através do sinal listSAGs enviado ao processo *ssUAPols_i_Api*. Esta solicitação é repassada ao *SSM_i_ProviderOlsSi* através do procedimento *SSM_listSAGs* e em seguida ao *SUB_i_SubscriberInfoMgmt* através do procedimento *listSAGs*. É então verificada a existência do assinante junto ao *SUB_DBMSBroker* (*DBexistsSubscriber*)e obtém-se a lista de SAGs através do procedimento *DBgetSAGsBySub*. A lista é então retornada e exibida ao usuário através do sinal *display*.

Escolhe-se então um SAG que deseja-se associar ao usuário criado (no caso, o SAG 1) e solicita-se esta associação através do sinal *assignSAE* enviado a *ssUAPols_i_Api*. Esta solicitação é repassada a *SSMols_i_ProviderOlsSi* (*SSM_assignSAE*) e em seguida a *SUB_i_SubscriberInfoMgmt* (*assignSAE*). É então verificada a existência do assinante (*DBexistsSubscriber*), do SAG (*DBexistsSAG*) e do usuário (*DBexistsUser*) junto ao *SUB_Manager*. Por último o usuário é associado ao SAG através do procedimento *DBassignSAE* e uma mensagem de sucesso é enviada ao usuário.

Este MSC gerado (figura 4.9) é equivalente ao diagrama disponibilizado pelo TINA-C (figura 4.8), entretanto, assim como no caso anterior, são exibidas as trocas de mensagens entre os processos de um mesmo bloco, detalhando cada passo envolvido no processo.

Além da presença das interações entre processos de um mesmo bloco, duas outras diferenças podem ser notadas entre as figuras 4.9 e 4.8. O procedimento *getReferences*, que está ausente no MSC da simulação por já ter sido chamado quando o bloco SSMols foi criado (inicialização do serviço de subscrição) e a criação do namedUA referente ao usuário recém-inscrito. Este último, trata-se de um passo opcional, uma vez que de acordo com a recomendação TINA [35], o objeto namedUA pode ser criado apenas quando seu uso for necessário (o que foi optado na especificação desenvolvida neste trabalho). Como o namedUA

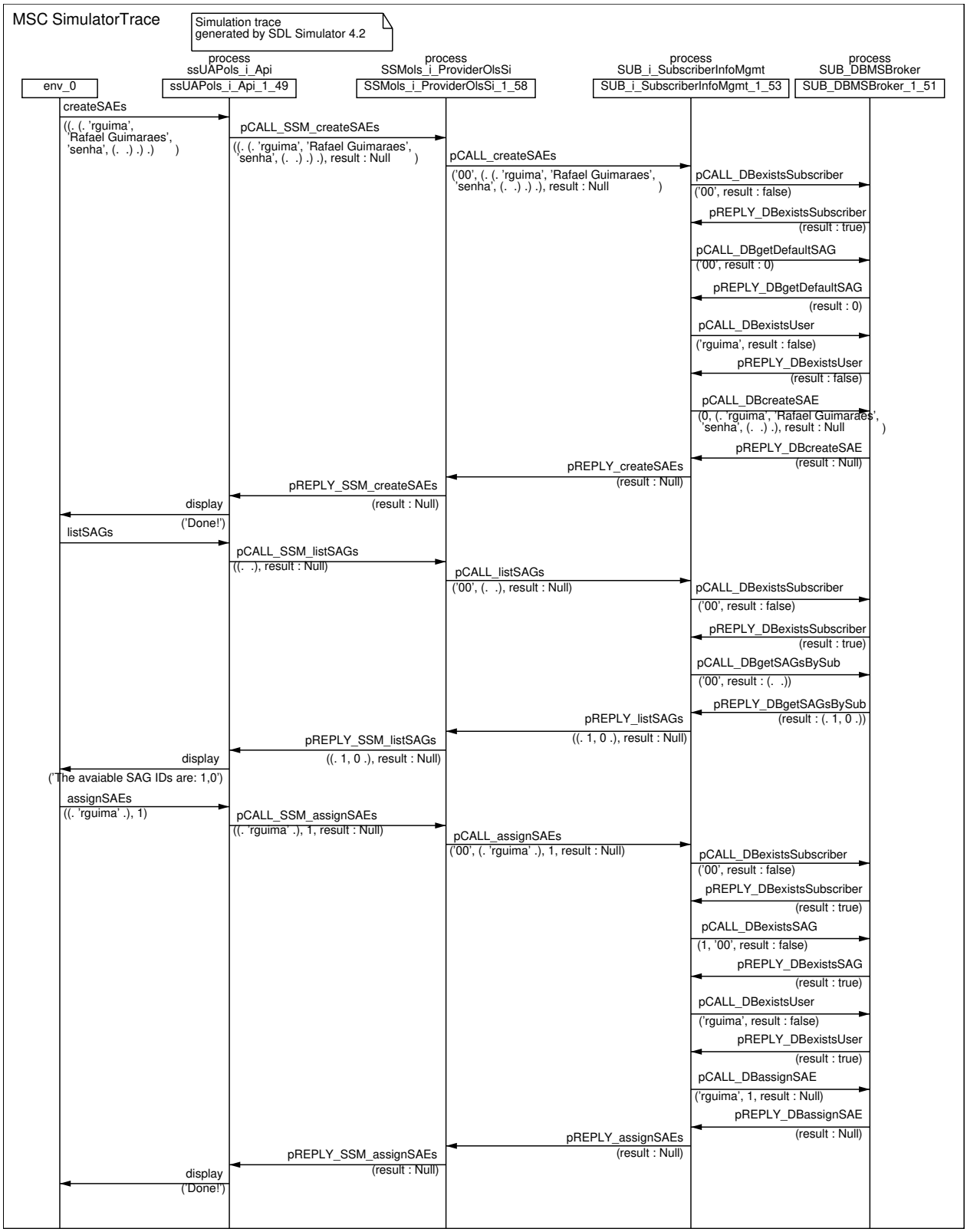


Figura 4.9: Criação de um novo usuário

não é criado neste momento, não é necessário o envio do sinal *notify* representado pelo passo 7 da figura 4.8.

A figura 4.10 mostra a simulação da ocorrência de uma exceção na aplicação. O usuário solicita a lista de todos os serviços disponíveis enviando o sinal *listServices* para o processo *ssUAPols_i_Api*. É chamado então o procedimento exportado *SSM_listServices* do processo *SSM_i_ProviderOlsSi*. Este procedimento por sua vez se comunica com o processo *SUB_i_Subscribe* através do procedimento exportado *listAllServices*. Esta informação deve ser obtida junto à base de dados utilizando o processo *SUB_DBMSBroker* como mediador. Entretanto, neste caso, não é possível obter a informação desejada (a base de dados pode estar fora do ar, por exemplo), é então gerada a exceção *e_applicationError* que é repassada para o processo *SSMols_i_ProviderOlsSi* e em seguida para *ssUAPols_i_Api*. Desta forma, é exibida uma mensagem de erro ao usuário, representada pelo sinal *display*.

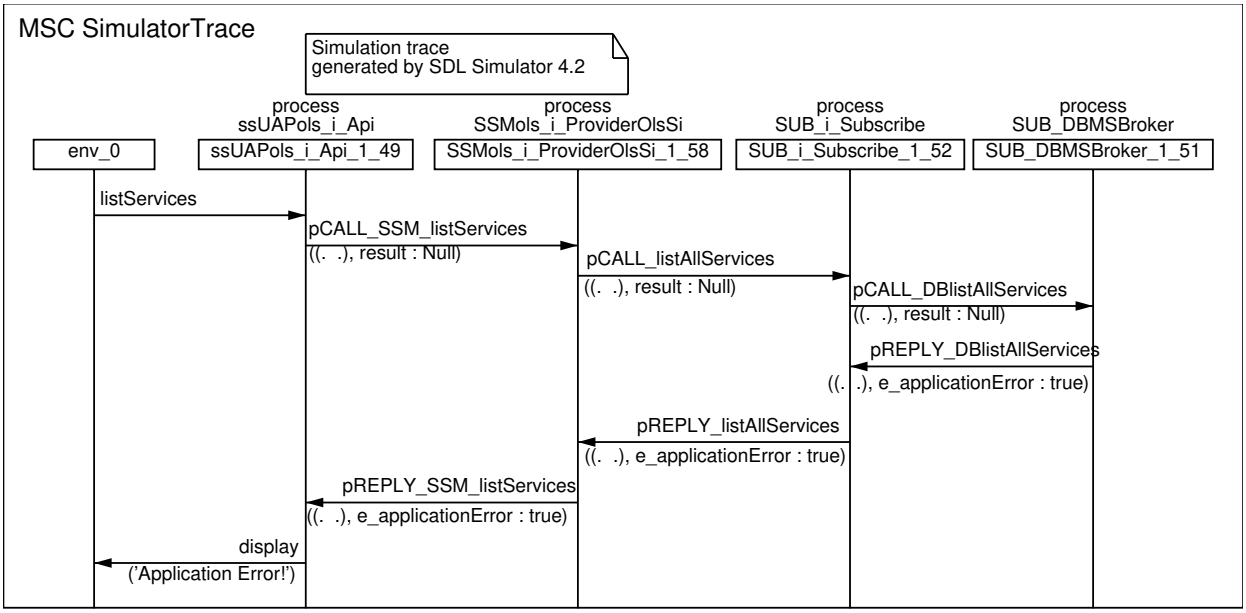


Figura 4.10: Ocorrência de exceção ao tentar listar os serviços disponíveis

Capítulo 5

Geração de código Java para objetos ODL especificados em SDL

Após o processo de validação e simulação das principais funcionalidades e casos críticos do sistema desenvolvido em SDL, pode-se desenvolver uma implementação a partir de sua especificação formal.

Visando automatizar o processo de criação de uma implementação para o sistema especificado, foi desenvolvida neste trabalho uma ferramenta capaz de gerar código distribuído em Java a partir de qualquer especificação em SDL que tenha sido desenvolvida segundo a metodologia proposta no capítulo 2, ou seja, através do uso do mapeamento de ODL para SDL da tabela 2.2. A conversão das especificações SDL para código Java é realizada pela ferramenta segundo as regras de mapeamento de SDL para Java da tabela 2.3 [36]. O código Java gerado faz uso de CORBA como plataforma distribuída de comunicação [37].

A automatização do processo de geração de código, além de simplificar o desenvolvimento do produto final, é bastante útil por evitar ao máximo a incidência de erros, uma vez que é eliminada (sempre que possível) a interferência manual.

Na seção 5.1 é apresentada a estrutura interna da ferramenta geradora de código Java desenvolvida, mostrando o seu funcionamento. A seção 5.2 apresenta o modelo de comunicação utilizado entre os objetos Java gerados pela ferramenta.

Na seção 5.3 são exibidas propostas para as conversões das principais construções SDL que não possuem equivalentes diretos em Java. É mostrado como é simulado o comportamento de variáveis reveladas e exportadas e como são modelados em Java os estados de um processo SDL e os sinais trocados. É apresentada, também, a conversão de procedimentos exportados em métodos síncronos Java, o que resulta em um código final mais adequado do que o

proposto por [27] (pois a característica síncrona da chamada a procedimentos é mantida em sua conversão para chamada a métodos Java). É apresentada também a simulação da construção *timer* através de classes Java com funcionalidade equivalente. Por último, é proposto uma nova conversão da construção *JOIN*, que traz uma ganho significativo quando comparado com [27].

Na seção 5.4 é apresentada a forma utilizada pelo programa Java gerado pela ferramenta a partir das especificações SDL para possibilitar o envio de sinais do usuário para o sistema e do sistema para o usuário através de uma interface gráfica.

5.1 A estrutura da ferramenta geradora de código

A ferramenta geradora de código Java utiliza como entrada um arquivo com as especificações SDL na forma textual (SDL-PR). Internamente o arquivo com as especificações SDL passa por duas fases de reconhecimento (figura 5.1).

Na primeira fase, é reconhecida a hierarquia dos blocos, processos e procedimentos, estas informações são armazenadas sob a forma de uma árvore em classes Java responsáveis pelo armazenamento do sistema SDL de uma forma estruturada. São reconhecidos também os caminhos de comunicação existentes entre os processos de um mesmo bloco (*signal routes*) e entre os blocos do sistema (*channels*), através dos quais pode-se identificar quais processos se comunicam entre si. Nesta fase ainda são reconhecidos os tipos de dados definidos no sistema (estruturas, conjuntos, enumerações, etc).

Após o armazenamento em classes das informações obtidas através da fase 1, a ferramenta processa novamente o arquivo de entrada, reconhecendo desta vez as especificações comportamentais dos processos e procedimentos e armazenando seus equivalentes em Java nas respectivas classes que compõem a árvore de hierarquia construída na Fase 1. Para a conversão das especificações comportamentais em SDL para o código Java equivalente são utilizadas as regras de mapeamento de SDL para Java propostas, além de algumas técnicas descritas neste capítulo. Para esta conversão é de fundamental importância o reconhecimento prévio dos tipos de dados (na fase 1), pois só assim pode-se identificar os tipos das variáveis e, assim, converter adequadamente os operadores da linguagem SDL que podem ter diferentes equivalentes em Java de acordo com os seus operandos. Uma operação que verifique a igualdade de duas variáveis em SDL, $A = B$, por exemplo, pode ser convertida para $A == B$, caso A e B sejam variáveis inteiras ou para $A.equals(B)$, caso A e B sejam um novo tipo de dados definido no sistema SDL.

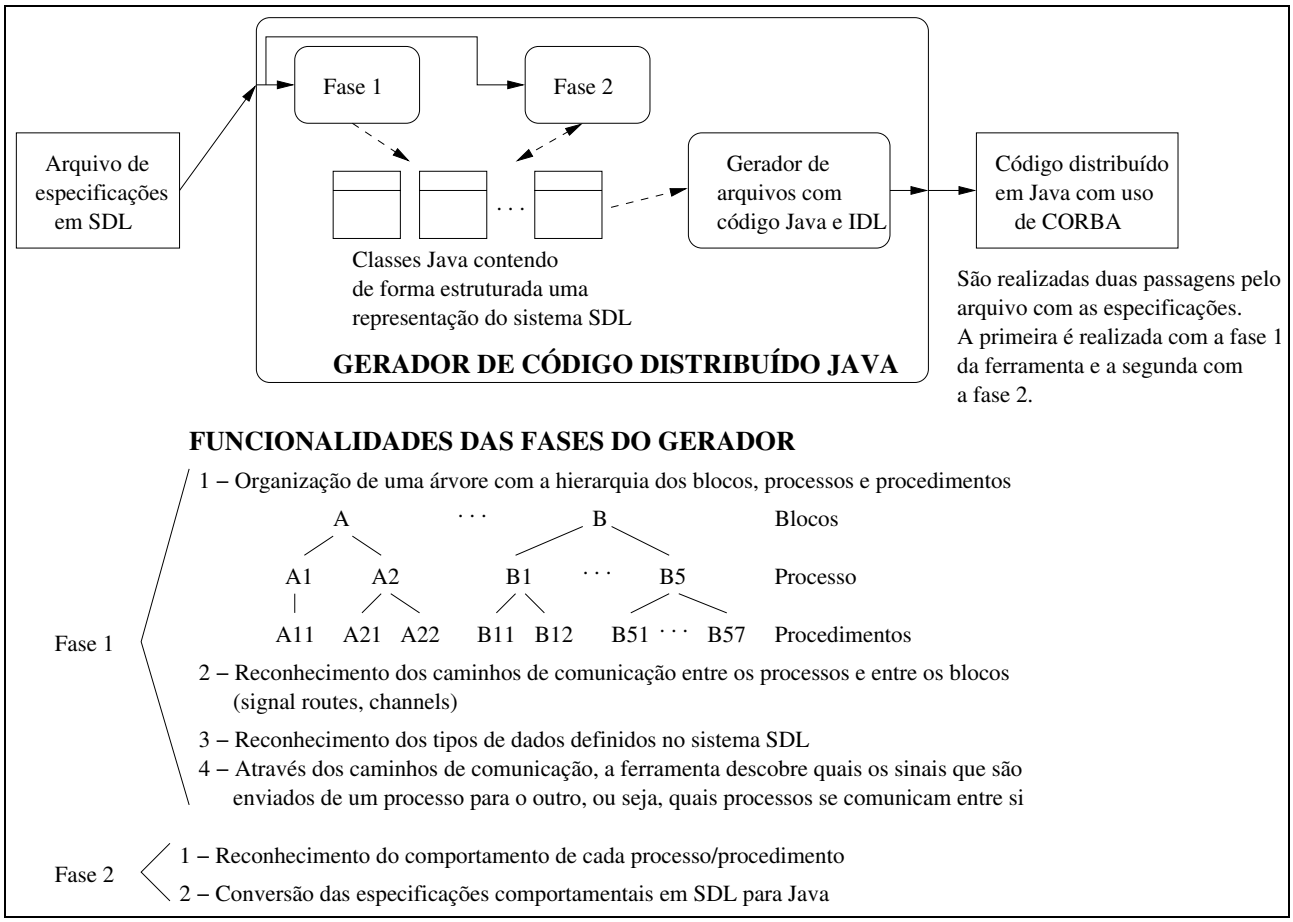


Figura 5.1: Ferramenta geradora de código distribuído Java

Após a finalização da fase 2, entra em ação o módulo responsável pela criação dos arquivos que contém o código Java para os objetos especificados e os arquivos IDL necessários para o uso de CORBA como plataforma de comunicação (DPE). Para a geração deste arquivo, faz-se uso das informações disponibilizadas pelas fases 1 e 2 da ferramenta sob a forma de objetos organizados de forma estruturada com os códigos Java das diferentes partes do sistema.

A plataforma CORBA utilizada pelo código Java gerado é o JavaIDL, que se trata de um ORB (*Object Request Broker*) de domínio público distribuído gratuitamente juntamente com as versões mais recentes do JDK [1] (*Java Development Kit*) disponibilizado pela Sun.

Através dos arquivos com as descrições em IDL das interfaces dos objetos Java gerados, o programa *idlj* (parte integrante do JDK) pode gerar o código Java necessário para o uso da plataforma CORBA.

[1]Neste trabalho foi utilizado o JDK versão 1.3.1, que pode ser livremente obtido em *http://java.sun.com*

Com isso, o processo de geração de uma implementação em Java do sistema especificado em SDL é completamente automatizado, bastando seguir a metodologia proposta.

5.2 O modelo de comunicação entre os objetos Java gerados

Os blocos SDL são compostos por diversos processos que se comunicam entre si e também com processos de outros blocos. Um bloco SDL é a representação de um objeto TINA, sendo assim, trata-se de uma entidade presente completamente em um nó da rede. Desta forma, pode-se afirmar que os objetos Java que representam os processos de um mesmo bloco estarão todos presentes em um mesmo nó da rede e, por isso, podem utilizar chamadas locais a métodos para se comunicar entre si, dispensando assim o uso da comunicação via CORBA.

Entretanto, quando se trata da comunicação de processos de diferentes blocos, os mesmos podem estar distribuídos em diversos nós da rede e, conseqüentemente a comunicação via CORBA entre os objetos Java que representam estes processos se faz necessária.

Os blocos SDL são mapeados em *packages* Java, que são coleções de objetos (representantes dos processos) SDL. Sendo assim, objetos Java de um mesmo *package* utilizam chamadas locais a procedimentos para se comunicar entre si, enquanto objetos Java de diferentes *packages* utilizam o ORB disponibilizado pela arquitetura CORBA (figura 5.2).

5.3 Conversão de elementos da linguagem SDL

Além das estruturas SDL apresentadas na tabela 2.3 proposta no capítulo 2 para o mapeamento de estruturas SDL em estruturas Java e IDL, algumas características da linguagem SDL tem que ser simuladas de alguma forma em Java por não possuírem um equivalente direto [36]. Nesta seção, a conversão realizada pela ferramenta geradora de código para as principais características são apresentadas.

5.3.1 Variáveis exportadas e reveladas

Os processos SDL podem exportar ou revelar variáveis para que outros processos do mesmo bloco tenham acesso aos seus valores. Para isso, basta declarar a variável como *REVEALED* ou *EXPORTED* no processo de origem e como *VIEWED* ou *IMPORTED* no processo que a visualizará. A diferença entre uma variável exportada e um variável revelada

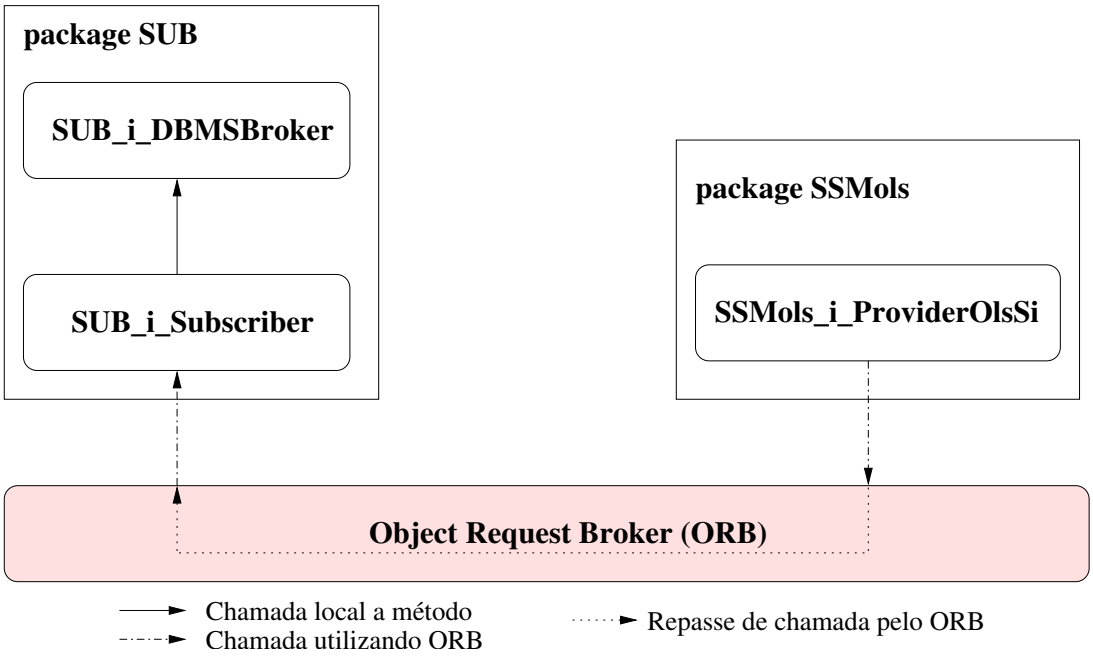


Figura 5.2: Chamadas de métodos: local e através do ORB

é que no primeiro caso a variável deve ser explicitamente exportada no processo de origem e, assim, o processo que a importa visualiza o valor da variável no momento em que ela foi exportada. Já no caso de uma variável revelada, o valor visualizado é o valor da variável naquele instante.

Para reproduzir este comportamento nos objetos Java gerados, é definida uma interface Java chamada *SDLObject* que deve ser implementada por todos os objetos que representam processos SDL. A interface *SDLObject*, mostrada na figura 5.3, possui dois métodos: um para a obtenção de variáveis exportadas (*___getExportedVar*) e outro para obtenção de variáveis reveladas (*___getRevealedVar*). Os métodos desta interface são implementados nos objetos como uma série de *if*s, que retornam o valor da variável exportada/revealada de acordo com o nome passado como parâmetro.

Para cada variável exportada, é definida uma variável auxiliar (*___Exported<NomeVar>*) que armazena o seu valor quando sua exportação é solicitada. Assim, o método responsável por retornar o valor de uma variável exportada, na realidade retorna o valor desta variável (que representa seu estado quando a exportação foi solicitada).

Definição da interface SDLObject

```
public abstract interface SDLObject {
  public abstract Object ___getExportedVar(String ___varName);
  public abstract Object ___getRevealedVar(String ___varName);
}
```

Definição dos métodos no objeto PA

```
// Exported and Revealed Variables
public java.lang.Object ___getExportedVar(String ___varName) {
  if (___varName.equals("TESTIR"))
    return new ObjectHolder(___ExportedTESTIR);
  else return null;
}
public java.lang.Object ___getRevealedVar(String ___varName) {
  if (___varName.equals("IAPROVIDERAUTHENTICATEIR"))
    return new ObjectHolder(IAPROVIDERAUTHENTICATEIR);
  else return null;
}
```

Visualização de variável exportada

```
valor = ((SDLObject) PAPid).___getExportedVar("TESTEIR");
```

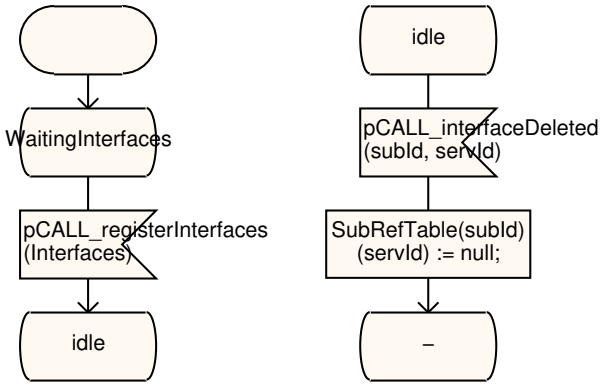
Figura 5.3: Interface SDLObject

5.3.2 Estados e sinais dos processos

Os processos SDL são formados por um conjunto de estados e transições (que são responsáveis pela mudança no estado do processo). Essas transições são iniciadas através da recepção de um sinal assíncrono, o qual é mapeado em um método assíncrono em Java. Ou seja, para cada sinal que um processo pode receber, existe um método no objeto Java equivalente, o qual é definido como *oneway* (assíncrono) em sua respectiva descrição IDL.

O estado de um processo SDL é representado em Java através de um atributo inteiro chamado $\_\_\_STATE$, no objeto equivalente. Para cada estado existente no processo é definida uma constante no objeto Java que é chamada $\_\_\_STATE\_<NomeEstado>$. Como a aceitação de um determinado sinal no processo SDL, bem como a transição que ele desencadeia, é condicionada ao estado do processo, no objeto Java equivalente o método que representa este sinal verifica o estado atual do objeto e executa as operações pertinentes a cada caso. Esta conversão dos estados e sinais SDL para Java é mostrada na figura 5.4

No exemplo da figura 5.4, o método assíncrono que representa a recepção do sinal $pCALL\_$-



CÓDIGO JAVA PARA OS MÉTODOS ASSÍNCRONOS DE SUB_Manager

```
protected int ___STATE;
protected int ___STATE_WAITINGINTERFACES = 0;
protected int ___STATE_IDLE = 1;

...

// Asynchronous Methods
public void PCALL_REGISTERINTERFACES
                        (T_SUBINTERFACES p0___)
   if ((___STATE==___STATE_WAITINGINTERFACES)) {
     INTERFACES = p0___;
     ___STATE = ___STATE_IDLE;
   }
}
public void PCALL_INTERFACEDELETED
                    (String p0___, int p1___)
   if ((___STATE==___STATE_IDLE)) {
     SUBID = p0___;
     SERVID = p1___;
     (SUBREFTABLE.gets(SUBID)).puts(SERVID,null);
   }
}
```

Figura 5.4: Conversão de estados e sinais SDL para código Java

registerInterfaces verifica primeiramente se o estado atual do processo é *WaitingInterfaces* através do teste "*if ((___STATE == __STATE_WAITINGINTERFACES))*", uma vez que o sinal só é tratado neste estado. Logo em seguida, o valor do parâmetro é atribuído à variável *INTERFACES* da classe que representa o processo. Em um último passo, o estado é alterado para idle através da atribuição "*___STATE = ___STATE_IDLE*". O método assíncrono que representa a recepçãod o sinal *pCALL_interfaceDeleted* testa se o estado atual do processo é *idle* através do teste "*if ((___STATE == __STATE_IDLE))*". Em seguida os valores passados como parâmetros são atribuídos às variáveis *SUBID* e *SERVID* e é atribuído o null a um valor armazenado na tabela *SubRefTable* através da operação "*(SUBREFTABLE.gets(SUBID)).puts(SERVID,null)*".

5.3.3 Procedimentos Exportados

Os procedimentos exportados SDL são convertidos para métodos em Java. Tanto os procedimentos exportados quanto os métodos Java possuem o mesmo caráter síncrono, ou seja, quando uma chamada é realizada, o processamento é interrompido até que o método, ou procedimento exportado, finalize sua execução.

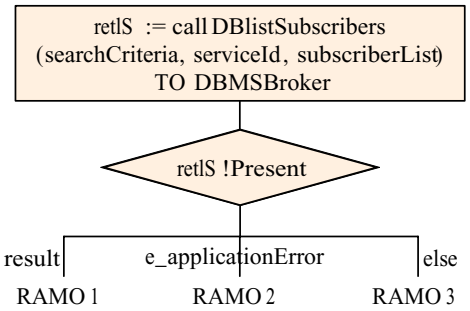
Esta similaridade conceitual e o fato das especificações comportamentais do procedimen-

to exportado estarem isoladas das demais especificações do processo (o que não ocorre com as especificações das transições resultantes da recepção de sinais, as quais estão todas misturadas no escopo do processo) faz com que o processo de geração de código seja bastante simplificado. Sendo assim, diferentemente do processo de geração de código apresentado em [27], os mapeamentos de ODL para SDL e de SDL para Java utilizados permitem que uma chamada a um método síncrono ODL seja mapeado em uma chamada síncrona a um método Java.

O resultado obtido é uma grande contribuição em relação ao código Java gerado em [27], onde uma chamada a um método síncrono ODL é mapeado em um par de sinais assíncronos SDL — um representando a chamada ao método e outro representando a resposta do método (ou a exceção gerada). Estes dois sinais SDL são, conseqüentemente, mapeados em duas chamadas assíncronas a métodos Java, durante o processo de geração de código. Tem-se como resultado, o mapeamento de uma chamada síncrona ODL em duas chamadas assíncronas no código Java correspondente. Sendo assim, o código gerado foge ao padrão esperado, ou seja, que uma chamada síncrona ODL seja mapeada em uma chamada síncrona no código Java gerado. Por este motivo, o protótipo gerado em [27] não é interoperável com outros sistemas TINA, ao contrário do código Java gerado pela ferramenta desenvolvida neste trabalho.

Caso o procedimento exportado possa gerar uma exceção, seu tipo de retorno é especificado em SDL com o uso do construtor *choice*. Neste caso os campos deste tipo indicam as exceções que podem ser geradas e, portanto, são utilizados para a construção do código Java para o método equivalente. A figura 5.5 mostra a conversão de um trecho de especificação SDL responsável pela chamada de um método e detecção da possível ocorrência de uma exceção.

Na figura 5.5, um procedimento exportado SDL é chamado e o seu resultado é avaliado para se decidir qual caminho tomar dependendo de qual componente (do resultado do tipo *choice*) esteja ativado. Se o componente ativado for *result* a execução segue o RAMO 1, se for *e_applicationError* segue o RAMO 2 e se for qualquer outro segue o RAMO 3. No código Java resultante, o método equivalente ao procedimento exportado é chamado dentro de uma construção *try* (responsável por detectar a geração de uma exceção). Caso seja gerada uma exceção, ela é detectada e a execução é redirecionada para o conteúdo da construção *catch* sendo a exceção armazenada na variável *___RETLSException*. Dentro da construção *catch* identifica-se se a exceção é do tipo *E_APPLICATIONERROR*, o que direciona a execução para o código equivalente ao RAMO 1, ou se a exceção é de outro tipo (*else*), o que direciona a execução para o código equivalente ao RAMO 2. Além disso, um flag de controle



CÓDIGO JAVA PARA A DETECÇÃO DA OCORRÊNCIA DE UMA EXCEÇÃO

```
try {
  ___RETLSResult = ((TSUB_DBMSBROKER) DBMSBROKER)).
                    DBLISTSUBSCRIBERS(SEARCHCRITERIA,
                                      SERVICEID,SUBSCRIBERID);
  ___ExceptionThrowed = false;
} catch(Exception ___RETLSException) {
  if (___RETLSException instanceof E_APPLICATIONERROR) {
    [Código para Ramo 1]
  } else {
    [Código para Ramo 2]
  }
  ___ExceptionThrowed = true;
}
if (!___ExceptionThrowed) {
  [Código para Ramo 3]
}
```

Figura 5.5: Conversão da detecção de ocorrência de exceção para código Java

(*___ExceptionThrowed*) indica se alguma exceção foi gerada. Caso não tenha sido, é executado o código para o RAMO 3.

5.3.4 A construção Timer

A linguagem SDL disponibiliza a possibilidade de se utilizar temporizadores (*timers*) com o objetivo de programar, para algum instante futuro, uma transição de estado. O temporizador SDL possui dois operadores, o *SET* e o *RESET*. O operador *SET* é utilizado para ativar o temporizador, indicando o instante de tempo em que ele entrará em ação enquanto o operador *RESET* é utilizado para desativar um temporizador. Uma vez que o tempo indicado tenha se passado, o temporizador envia um sinal (de mesmo nome do temporizador) para o processo que o iniciou. A partir daí este sinal é tratado como outro qualquer. Ou seja, caso um temporizador *TEMP* seja ativado para 10 unidades de tempo, após este intervalo, o sinal TEMP é enviado ao processo e é tratado como um sinal comum.

Para simular este comportamento em Java, é definida uma classe temporizadora (figura 5.6) para cada *timer* existente. Esta classe disponibiliza as operações de *SET* e *RESET* equivalentes e, para simular o envio do sinal de mesmo nome do temporizador, chama o respectivo método do objeto que a criou após decorrido o tempo desejado (a unidade de tempo utilizada é o segundo).

```
public class UPDATEDBMSTimerClass {
  Timer timer;
  class SDLTimerTask extends TimerTask {
    org.omg.CORBA.Object sender;
    public SDLTimerTask() {
      super();
    }
    public void run() {
      ((SUB_I_SERVICECONTRACTINFOMGMT) sender).
                                        UPDATEDBMS(sender);
      timer.cancel();
    }
  }
  public UPDATEDBMSTimerClass(org.omg.CORBA.Object _sender) {
    timer = new Timer();
    sender = _sender;
  }
  public void set(long when) {
    timer.schedule(new SDLTimerTask());
  }
  public void reset() {
    timer.cancel();
  }
}
```

Figura 5.6: Código Java para o temporizador *updateDBMS* definido no processo *SUB_i_ServiceContractInfoMgmt* da figura 3.26

5.3.5 Join

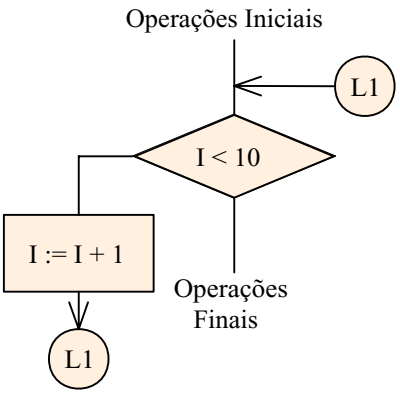
Um dos maiores problemas encontrados durante a geração de código Java é a ausência da construção *GOTO* na linguagem Java. Uma vez que SDL faz um uso extensivo do *JOIN* (que é um tipo de *GOTO*), o código gerado deve de alguma forma simular sua funcionalidade. Uma solução possível é a colocação das operações rotuladas em métodos privados e o mapeamento de uma operação de *JOIN* para este rótulo para uma chamada ao respectivo método. Esta solução, apresentada em [27], resolve o problema, apesar de gerar um código ineficiente, com muita recursão. Em um caso limite, esta solução poderia acarretar um estouro da pilha de execução.

Entretanto, visando evitar a ineficiência do código resultante, o gerador de código Java desenvolvido utiliza uma outra técnica, proposta por este trabalho, para simular o comportamento do *JOIN*. Esta solução consiste em mapear os rótulos existentes em opções de uma decisão do tipo "*if ... else if ... else ...*". São definidas constantes para cada rótulo existente (*___LABEL_<NomeRótulo>*), juntamente com mais duas constantes: uma que representa "nenhum rótulo" (*___NOLABEL*) e uma que representa o rótulo pelo qual o método deve

iniciar sua execução (*___INITIALLABEL*). Além disso, é definida uma variável *___LABEL* que contém o valor do próximo rótulo para o qual o processamento deve se encaminhar.

A decisão de que rótulo seguir é inserida em um laço de repetições (*while*), que se repete até que não haja mais rótulos para os quais encaminhar o processamento, ou seja, até que a variável *___LABEL* seja igual a *___NOLABEL*.

A figura 5.7 mostra um exemplo destacando a parte da especificação referente ao *JOIN*. Em um determinado instante da especificação há uma decisão em que o valor de I é testado. Caso ele seja menor que 10, seu valor é incrementado e a execução se direciona novamente para a decisão através de um *JOIN* para o rótulo *L1*. Caso ele não seja menor que 10, a execução segue o caminho das operações finais.



CÓDIGO JAVA PARA JOIN

```
___LABEL = ___INITIALLABEL;
while (___LABEL != ___NOLABEL) {
  if (___LABEL == ___INITIALLABEL) {
    [Código para Operações Iniciais]
    ___LABEL = ___LABEL_L1;
  } else if (___LABEL == ___LABEL_L1) {
    if ((I<10) == true) {
      I = I + 1;
      ___LABEL = ___LABEL_L1;
    } else if ((I<10) == false) {
      [Código para Operações Finais]
      ___LABEL = ___NOLABEL;
    }
  }
}
```

Figura 5.7: Simulação da funcionalidade de *JOIN* em Java

No código Java equivalente, inicialmente a variável *___LABEL* recebe o valor *___INITIAL-LABEL*. Assim que a execução se inicia, o programa entra no *while*, ___LABEL é avaliada pelo *if* e as operações abaixo de "*if(___LABEL == ___INITIALLABEL)* " são executadas (o bloco de código para as operações iniciais). Em seguida, *___LABEL* recebe o valor *___LABEL_L1*, indicando que o processamento será redirecionado para o rótulo *L1*. Na próxima iteração do *while*, o processamento é direcionado para as operações abaixo de "*else if(___LABEL == ___LABEL_L1)*".

Supondo que I seja menor que 10, o valor de I é incrementado, *___LABEL* recebe o valor *___LABEL_L1* e novamente o processamento se redireciona para o rótulo *L1*. Isso se repete até que I seja igual a 10. Neste caso, o bloco de código das operações finais é executado e

___LABEL recebe o valor *___NOLABEL*. Sendo assim, na próxima iteração, a condição do *while* é avaliada para *false* e o método encerra a sua execução.

Como mostrado por este exemplo, o código gerado possui exatamente a mesma funcionalidade da operação *JOIN* em SDL e faz isso de forma eficiente, sem o uso de recursão, como é feito em [27].

5.4 A entrada e a saída de dados no programa Java gerado

Um sistema SDL se comunica com o usuário através de sinais trocados com o ambiente, os quais são responsáveis pela entrada e saída de dados do sistema. No código gerado em Java para a especificação desenvolvida, estes sinais são mapeados em uma interface gráfica que possui funcionalidade equivalente.

Quando o processo SDL se encontra em um determinado estado, um conjunto de sinais podem ser recebidos do usuário. Em Java, dependendo do estado em que se encontre o objeto, estes sinais podem ser escolhidos a partir de uma lista dos sinais que podem ser recebidos (figura 5.8). Os sinais enviados do sistema para o ambiente, por sua vez, são exibidos através de uma caixa de diálogo que contém o sinal recebido e seus parâmetros (figura 5.9). No caso do envio de sinais que necessitem de parâmetros, estes são solicitados ao usuário através de uma nova janela (figura 5.10). Desta forma, pode-se então estabelecer uma comunicação entre usuário e sistema de forma similar ao especificado em SDL.

Figura 5.8: Escolha do sinal de entrada a ser enviado para o sistema

A ferramenta geradora de código distribuído Java foi criada com o auxílio do JavaCC [17] (*Java Compiler Compiler*), que é uma ferramenta construtora de compiladores. Para isso, foi desenvolvido um programa com cerca de 12.500 linhas de código que possui informações acerca das regras de formação da linguagem SDL. Para cada elemento da linguagem SDL (um bloco, um processo, etc) reconhecido em uma especificação que esteja sendo analisada, a ferramenta

Figura 5.9: Sinal recebido do sistema

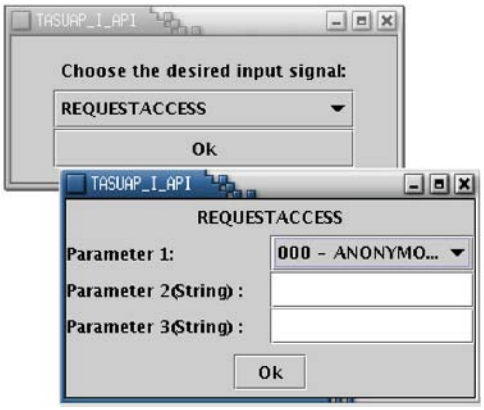


Figura 5.10: Entrada dos parâmetros exigidos pelo sinal

toma decisões sobre o que deve ser executado para converter a especificação em SDL. No apêndice B são apresentados conceitos relacionados com a construção de compiladores e o seu uso para o reconhecimento de uma linguagem (como o SDL, por exemplo). Também é apresentada a ferramenta JavaCC.

A ferramenta geradora de código Java desenvolvida é responsável por reconhecer e interpretar qualquer programa escrito em SDL e convertê-lo para Java utilizando as regras de mapeamento de SDL para Java propostas na subseção 2.6.4 do capítulo 2. Ela utiliza como entrada um programa qualquer em SDL-PR (versão textual das especificações em SDL) que tenha sido criado utilizando a metodologia proposta no capítulo 2.

Capítulo 6

Conclusões

O grande avanço nas tecnologias e o aumento da competitividade do mercado de telecomunicações, principalmente nos últimos anos, tem causado um aumento na demanda por novos serviços, os quais são cada vez mais complexos.

A integração de todos estes serviços em uma só rede de telecomunicações é um dos maiores objetivos atualmente, assim como a criação de um ambiente robusto para o provimento destes serviços. A proposta de TINA vem ao encontro destas necessidades, introduzindo uma arquitetura distribuída que permite o provimento/gerenciamento de todos os tipos de serviços de telecomunicações de forma padronizada.

Sendo a competitividade do mercado de telecomunicações bastante alta, existe uma demanda por novas metodologias para o desenvolvimento destes sistemas baseados em TINA. Estas metodologias devem contribuir trazendo facilidades para o projeto, validação, implementação e manutenção de novos ambientes que utilizem esta arquitetura e que permitam a introdução de novas funcionalidades de forma simples. A redução no tempo dispendido nestes processos de desenvolvimento é de fundamental importância, contribuindo para a agilização na disponibilização de novos serviços e para a redução dos custos envolvidos neste processo.

Neste contexto, a metodologia apresentada no capítulo 2 para o desenvolvimento orientado a objetos com a utilização da linguagem de especificação formal SDL aplicada a sistemas TINA pode reduzir significativamente o tempo total de desenvolvimento dos sistemas. A conversão dos objetos ODL disponibilizados pelo TINA-C em estruturas SDL equivalentes através do mapeamento de ODL para SDL proposto permite a formalização comportamental dos objetos, o que possibilita a realização de simulações e validação do sistema com o objetivo de detectar e corrigir os erros de especificação que possam existir.

Um ganho significativo foi alcançado no mapeamento de ODL para SDL proposto, em

relação a trabalhos anteriores [25], devido à utilização de procedimentos exportados para representar métodos síncronos ODL. Desta forma, pode-se manter a mesma característica síncrona da chamada a um método ODL, não sendo necessária a simulação deste comportamento através da troca de sinais assíncronos, conforme proposto por [25].

Através da utilização desta metodologia foi proposta no capítulo 3 uma especificação formal para os objetos ODL do modelo de subscrição e da sessão de acesso TINA disponibilizados pelo TINA-C, diferentemente de outros trabalhos [25] que se concentraram na especificação da sessão de serviço TINA. Através desta especificação pode-se observar a utilização do recurso de herança disponibilizado pelo SDL na simplificação do processo de desenvolvimento dos objetos.

O uso de uma ferramenta CASE para o desenvolvimento profissional em SDL, como o SDT, traz grandes vantagens para realização das especificações. A utilização de uma representação gráfica da linguagem SDL (SDL-GR) facilita bastante o processo de especificação. No capítulo 4, a realização do processo de validação das especificações desenvolvidas permite a detecção e correção de erros na especificação. Além disso, a análise dos resultados de simulações através de diagramas MSC gerados pelo simulador é de compreensão bastante simples e permite a detecção de erros de lógica no sistema. A comparação realizada entre alguns diagramas MSC resultantes de simulações realizadas e diagramas de trocas de mensagens disponibilizados pelo TINA-C permite a verificação de algumas funcionalidades do sistema, mostrando-se de acordo com o esperado pelas recomendações TINA.

A conversão automática das especificações desenvolvidas em SDL para código distribuído Java através da ferramenta desenvolvida (capítulo 5), permite uma agilização no processo de implementação, simplificando bastante o processo de criação de um protótipo para o sistema desejado. Foram propostas algumas soluções para a conversão de características da linguagem SDL que não possuem equivalente direto em Java.

No caso da construção *JOIN* disponibilizada pelo SDL, foi encontrada uma solução em SDL que é um grande avanço em relação a trabalhos anteriores [27]. A utilização de procedimentos exportados para modelar métodos síncronos ODL foi responsável por um ganho significativo no processo de geração de código Java, pois os procedimentos exportados SDL foram mapeados em métodos síncronos SDL, diferentemente da metodologia utilizada em [27], onde um método síncrono ODL é representado por uma troca de sinais assíncronos em Java.

A ferramenta desenvolvida tem como resultado um código Java equivalente ao sistema especificado em SDL, que faz uso de CORBA, mais especificamente da implementação JavaI-

DL disponibilizada gratuitamente pela Sun, como plataforma distribuída de comunicação. Apesar de neste trabalho a ferramenta ter sido utilizada para a geração de um protótipo para o sistema especificado no capítulo 3, sua utilização é mais extensa, podendo converter para Java qualquer especificação em SDL de objetos TINA, desde que tenha sido aplicada a metodologia proposta.

Pode-se ainda estender a utilização da metodologia de especificação e geração de código proposta para quaisquer sistemas distribuídos, desde que tenha-se a descrição dos objetos em IDL, as quais servirão de base para a construção das especificações SDL. Isto é possível devido a enorme similaridade entre ODL e IDL, o que permite que as mesmas regras de mapeamento de ODL para SDL sejam utilizadas para converter especificações IDL em SDL.

Baseado na metodologia proposta e na ferramenta de geração de código distribuído Java desenvolvida, pode-se propor a realização de alguns trabalhos futuros:

- Migração da metodologia para próximas versões do SDL (2000, 2004, etc);
- Desenvolvimento de uma ferramenta para a conversão automática das especificações SDL a partir de descrições ODL utilizando o mapeamento proposto neste trabalho, criando assim a hierarquia de blocos, processos e procedimentos, deixando apenas a especificação comportamental em aberto;
- Aumento nas possibilidades de "personalização" da ferramenta geradora de código Java, como a possibilidade de se configurá-la para gerar código voltado para outras implementações de CORBA, além do JavaIDL, ou para a utilização de outras plataformas de distribuídas de comunicação, como o DCOM (*Distributed Component Object Model*);
- Introdução na ferramenta geradora de código Java da possibilidade de se gerar código para apenas algumas partes do sistema SDL (apenas um bloco, por exemplo), uma vez que a ferramenta atual permite apenas que todo o sistema SDL seja convertido para Java. Isto pode ser bastante interessante, uma vez que pode-se ter o interesse de gerar código Java apenas para a parte relativa ao cliente (a qual necessita da portabilidade oferecida por Java), deixando a parte do servidor para ser implementada em outra linguagem mais eficiente (C++, por exemplo). Além disso, nem sempre tem-se o interesse de gerar código para todos os objetos especificados em um sistema. Pode-se desejar gerar código para apenas um objeto, que irá se comunicar com outros já implementados;

Referências Bibliográficas

[1] Object Management Group (OMG), Needham, MA - EUA. *The Common Object Request Broker: Architecture and Specification*, Outubro 1999.

[2] A. Chaffee, B. Martin. Introduction to CORBA. Technical report, Instituto Magelang, 1999.

[3] F. Dupuy, G. Nilsson, Y. Inoue. The TINA Consortium: Towards Networking Telecommunications Information Services. *IEEE Communications Magazine*, 33(11):78–83, Novembro 1995.

[4] Y. Inoue, D. Guha, H. Berndt. The TINA Consortium. *IEEE Communications Magazine*, 36(9):130–136, Setembro 1998.

[5] M. Chapman, S. Montesi. Overall Concepts and Principles of TINA. Technical report, Consórcio TINA, Fevereiro 1995.

[6] M. Lapierre et al. *The TINA Book: A co-operative solution for a competitive world*. Prentice Hall Europe, Hertfordshire, Inglaterra, 1st. edition, 1999.

[7] H. Berndt, T. Hamada, P. Graubmann. TINA: Its achievements and its future directions. *IEEE Communications Surveys & Tutorials*, 3(1):2–16, Primeiro Quadrimestre 2000.

[8] R.P. Guimarães. TINA - Telecommunications Information Networking Architecture. *Projeto de Graduação - UFES*, Janeiro 2000.

[9] D. Muldowney, C. Liccardi. What is TINA and is it useful for the TelCos? Assessment of TINA and the Internet. Technical report, EURESCOM - Projeto P847-GI, 1999.

[10] A. Parhar. TINA Object Definition Language Manual - Version 2.3. Technical report, Consórcio TINA, Julho 1996.

[11] ITU-T, Genebra, Suiça. *Specification and Description Language*, 1993. (CCITT Recommendation Z.100).

[12] K. Kimbler. Specification of the TOSCA Process Architecture for Service Creation. Technical report, Projeto TOSCA AC237-D9, Dezembro 1998.

[13] S. Efremidis et al. Re-Usable Components and Services from Experiments on Targeting Architecture. Technical report, Projeto SCREEN A-42-D48, Agosto 1999.

[14] C. Horstmann, G. Cornell. *CoreJava 2 - Volume I - Fundamentos*. MAKRON Books, São Paulo, 2001.

[15] Telelogic AB, Suécia. *Telelogic Tau 4.2 SDL Suite Getting Started*, Setembro 2001.

[16] Telelogic AB, Suécia. *Telelogic Tau 4.2 SDL Suite Methodology Guidelines*, Setembro 2001.

[17] Metamata Inc. *Java Compiler Compiler (JavaCC) - The Java Parser Generator*, 2000.

[18] C. Abarca et al. Service Architecture - Version 5.0. Technical report, Consórcio TINA, Junho 1997.

[19] ITU-T, Genebra, Suiça. *Message Sequence Chart (MSC)*, 1993. (CCITT Recommendation Z.120).

[20] D. Brown, S. Montesi. Requirements upon TINA-C Architecture. Technical report, Consórcio TINA, 1995.

[21] P. Farley et al. Ret Reference Point Specifications - Version 1.1. Technical report, Consórcio TINA, Abril 1999.

[22] N. Hoa. Distributed Object Computing with TINA and CORBA. In *WDS'97 - Charles University*, Charles University, Praga, República Tcheca, Junho 1997.

[23] G. Pavlou et al. Issues in Realising the TINA Network Resource Architecture. *Interoperable Communication Networks Journal*, 2(1):133–146, Março 1999.

[24] J. Pavón et al. The vital network resource architecture. In *TINA '97 Conference on Global Convergence of Telecommunications and Distributed Object Computing*, pages 130–138, Santiago, Chile, 1997.

[25] M. Kolberg, R. Sinnott, E. Magill. Experiences modelling and using formal object-oriented telecommunication service frameworks. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31:2577–2592, Dezembro 1999.

[26] M. Kolberg, R. Sinnott. Business-Oriented Development of Telecommunication Services. In *Workshop on Behavioural Semantics of OO Business and System Specification (OOPSLA'98)*, Vancouver, Canadá, Outubro 1998.

[27] E. Sherratt, C. Loftus. Designing distributed services with SDL. *IEEE Concurrency*, 08(01):59–66, Janeiro-Março 2000.

[28] C. Abarca et al. Service Architecture Annex - Version 5.0. Technical report, Consórcio TINA, Junho 1997.

[29] N.A. Nassif, W.C. Borelli. A combinação da Técnica de Orientação a Objetos OOA com a Linguagem Formal de Especificação SDL para o desenvolvimento de um Sistema de Banco de Dados. In *Terceiro Congresso Argentino de Ciência da Computação (CACIC'97)*, La Plata, Argentina, 1997.

[30] C.G. Macário, M. Pedroso, W.C. Borelli. 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*, pages 351–365, Paris, França, 1997.

[31] A. Olsen et al. The Pros and Cons of using SDL for creation of Distributed Services. In *Sixth International Conference on Intelligence in Services and Networks (IS&N 99)*, Barcelona, Espanha, Abril 1999.

[32] M. Bjorkander. Mapping IDL to SDL. Technical report, Telelogic AB, 1997.

[33] Object Management Group (OMG), Needham, MA - EUA. *OMG IDL to Java Language Mapping*, Novembro 2000.

[34] R. Guimarães, W. Borelli. Uma proposta de especificação formal em SDL de uma Sessão de Acesso TINA. In *19o Simpósio Brasileiro de Telecomunicações (SBrT'2001)*, Fortaleza, CE, Setembro 2001.

[35] C. Abarca et al. Service Component Specification - Version 1.0b Final. Technical report, Consórcio TINA, Janeiro 1998.

[36] R. Guimarães, W. Borelli. Object-Oriented Development of TINA systems using SDL and Java. In *X Congreso Internacional de Computación*, Cidade do México, DF, México, Novembro 2001.

[37] R. Orfali, D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, Inc., Nova Yorque, NY, EUA, 2nd. edition, 1998.

[38] ITU-T, Genebra, Suiça. *Specification and description language (SDL) combined with abstract notation one (ASN.1)*, 1995. (CCITT Recommendation Z.105).

Apêndice A

A linguagem SDL e o SDT

O SDL (*Specification and Description Language*) [11] é uma liguagem padronizada pelo ITU-T (*International Telecommunications Union – Telecommunication*) para a especificação e descrição de protocolos e sistemas de telecomunicações. O SDL se concentra basicamente na especificação dos aspectos comportamentais do sistema (envolvendo dados, quando necessário), sendo assim quaisquer outros aspectos se encontram fora do escopo do SDL.

A linguagem SDL é uma linguagem ao mesmo tempo textual e gráfica, ou seja, os sistemas especificados em SDL podem fazer uso de componentes gráficos da linguagem, os quais possuem equivalentes textuais. Por questões de praticidade, as ferramentas de desenvolvimento em SDL fazem uso da representação gráfica da linguagem, sendo assim, é a esta representação que é dado enfoque neste apêndice.

O desenvolvimento do SDL se iniciou em 1972, tendo recomendações lançadas em 1976, 1980, 1984, 1988, 1992, 1996 e 2000, inicialmente pelo CCITT e depois pelo ITU-T. Esta última recomendação, entretanto, por ainda se tratar de uma versão muito recente, não possui suporte completo das ferramentas de desenvolvimento disponíveis no mercado (apenas algumas poucas funcionalidades estão disponíveis em algumas ferramentas). Sendo assim, a seguir serão apresentadas brevemente algumas características da versão utilizada nesta tese, o SDL-96.

A.1 Estruturação de um sistema

Primeiramente, deve-se diferenciar especificação, tipo e instância. Uma especificação define um sistema que é composto por diversas entidades que se comunicam entre si. Esta entidades possuem um comportamento definido que determina qual é o seu tipo. Cada

um destes tipos pode possuir inúmeras instâncias, todas possuem o mesmo comportamento (mesmas caracterísicas), mas existem de forma independente umas das outras. Ou seja, o tipo pode ser visto como uma descrição das instâncias que serão criadas para que, ao se relacionar, executem as tarefas do sistema.

Para a especificação dos tipos, o SDL oferece algumas estruturas básicas. Dentre elas podemos destacar:

System: É a entidade mais externa, que representa o sistema como um todo. É composto por diversos blocos que se comunicam entre si e com o ambiente através de canais de sinais. Os sinais e tipos de dados utilizados pelo sistema e que devem ser conhecidos por diversos blocos devem ser declarados neste nível.

Block: É entidade que agrupa um ou mais processos que se comunicam entre si e com o os canais de comunicação do sistema (podendo se comunicar com outros blocos ou com o ambiente). Os sinais e tipos de dados utilizados pelo bloco e que ainda não foram declarados no nível do sistema devem ser declarados neste nível.

Process: É a estrutura responsável por um determinado processo a ser executado. É neste nível que é entra a especificação comportamental do sistema, através da descrição de uma máquina de estados finita com dados, que é independente das demais. O conjunto de processos que são executados em paralelo é que definem o comportamento do sistema. Entretanto, as funcionalidades dos processos pode ser agrupadas em procedimentos.

Procedure: Da mesma forma que nas linguagens de programação, agrupam funcionalidades que se repetem em diversas partes do sistema ou então agrupam funcionalidades de forma a simplificar a análise sistema. Um procedimento pode ou não receber parâmetros para o seu processamento e, da mesma forma, pode ou não retornar um resultado. Além disso, todas as variáveis declaradas no processo são visíveis pelos procedimentos contidos no mesmo. Os procedimentos podem ser declarados como *Exported Procedures*, possibilitando que outros processos e outros blocos possam fazer uso do mesmo.

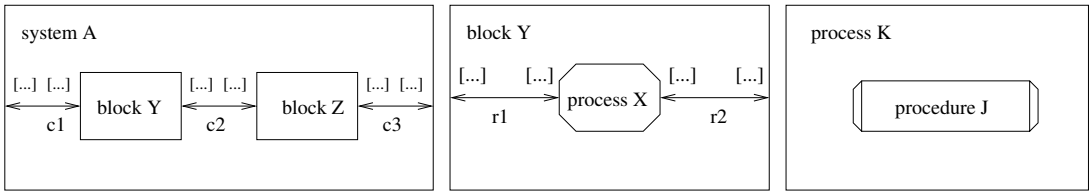


Figura A.1: Estruturas básicas do SDL

A.2 Comunicação em sistemas SDL

Toda a comunicação entre blocos e entre processos é realizada através de trocas de sinais e de chamadas a procedimentos remotos.

A.2.1 Troca de sinais

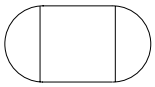
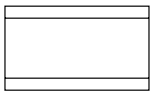
Os sinais são transportados de um bloco para outro ou para o ambiente através de canais de comunicação (vide *c1*, *c2* e *c3* na figura A.1). Cada um destes canais pode ser uni ou bidirecional e possui listas de sinais que indicam quais sinais são trocados em cada direção do canal. Para a troca de sinais entre processos, faz-se uso de rotas de sinais (vide *r1* e *r2* na figura A.1), os quais possuem uma estrutura similar aos canais de comunicação. Os processos também fazem uso de rotas de sinais para se comunicar com o nível mais externo, conectando uma das extremidades da rota de sinais à um canal de comunicação do nível de sistema. Neste caso, os sinais transportados pelas rotas de sinais devem ser os mesmos transportados pelo canal de comunicação.

A.2.2 Chamada a Procedimentos Remotos

As chamadas a procedimentos remotos (RPC - *Remote Procedure Call*) são a outra forma de comunicação entre processos.Quando se deseja que um dado procedimento seja visível por um outro processo (processo cliente), o mesmo deve ser declarado como *Exported Procedure* no processo servidor. Além disso, o processo cliente deve declará-lo como *Imported Procedure*, avisando que a definição deste processo encontra-se em outro escopo. Finalmente, o procedimento deve ser declarado como *Remote Procedure* em algum nível visível aos processos cliente e servidor.

A.3 Principais elementos para a descrição de processos e procedimentos

A seguir são apresentados alguns dos principais elementos utilizados na especificação dos processos e dos procedimentos que compõem uma especificação em SDL.

Representação	**Descrição**
	*Start*: Utilizado apenas uma vez no processo indicando o início de sua execução.
	*Stop*: Utilizado para indicar o término da sua execução de um processo.
	*Start Procedure*: Utilizado apenas uma vez no procedimento indicando o início de sua execução.
	*Return*: Utilizado para indicar o término da sua execução de um procedimento. Indica um valor de retorno do procedimento, quando for o caso.
	*Task*: Utilizado para definir uma tarefa ou um conjunto de tarefas (separados por ';'). As tarefas podem ser substituídas por uma string com a descrição textual das mesmas.
	*Create*: Utilizado para criar uma instância de um processo, desde que esteja no mesmo bloco do processo criador.
	*State*: utilizado para indicar o estado em que se encontra a máquina finita que representa o processo/procedimento. Uma mudança de estado só ocorre quando se recebe um sinal. O estado pode ter um nome, ou pode ser declarado o estado '$*$' que se refere a todos os estados existentes, indicando que as ações a serem tomadas são comuns a todos eles. Além disso, após a recepção de um sinal (e execução das ações relacionadas), a execução pode encontrar o estado '-' que indica que o estado não se alterou devido às ações.

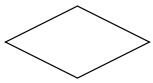
Representação	**Descrição**
	*Procedure Call*: Utilizado para indicar uma chamada a um procedimento.
	*Text*: Utilizado para a declaração de variáveis, sinal e tipos de dados.
	*Comment*: Utilizado para a inserção de comentários na especificação.
	*Decision*: Utilizado para uma tomada de decisão. Consiste de uma questão e diversas possíveis repostas e, dependendo da resposta, a execução toma uma caminho distinto.
	*Input*: Utilizado para indicar um sinal de entrada. Deve vir sempre após um estado indicando uma transição devido ao recebimento deste sinal.
	*Output*: Utilizado para indicar um sinal de saída. Através desta estrutura envia-se sinais e dados para outros processos. Para cada *output* deve haver um *input* associado no processo de destino. Para indicar o processo destino pode-se utilizar o endereçamento implícito – omitindo-se o endereço do destino – ou explícito – indicando-se o endereço destino através da opção *TO <dest>* ou indicando a rota de sinal a seguir através da opção *VIA <rota>*).

Tabela A.1: Descrição e representação de algumas estruturas SDL

A.4 Herança e Especialização de Tipos

No processo de desenvolvimento de um sistema é muito comum a especialização de um tipo pré-existente, incluindo novas funcionalidades ou redefinindo antigas. Para isso, o SDL

fornece a facilidade do reuso de especificações através do conceito de herança de tipos, onde todas as propriedades do supertipo são herdadas pelo subtipo, devendo ser especificadas apenas as diferenças existentes entre os mesmos.

A herança de tipos é feita através da construção *INHERITS*. Caso a herança seja apenas para adicionar novas funciolidades (sem a redefinição de antigas), deve-se utilizar a construção *INHERITS <supertipo> ADDING*. Entretanto, se além da inclusão de novas propriedades, outras existentes forem modificadas, utiliza-se a construção *INHERITS <supertipo>* e define-se a propriedade a ser modificada como *VIRTUAL* no supertipo, redefinindo-a com a construção *REDEFINED* no subtipo.

Para que se possa utilizar este conceito de herança, alguns tipos devem ser utilizados:

System Type: Define um sistema a ser reusado por outro através do mecanismo de *Package*.

Block Type: Define um conjunto de blocos com as mesmas características (ou seja, um tipo de bloco). Cada *Block Type* pode possuir inúmeras instâncias.

Process Type: Define um conjunto de processos com as mesmas características (ou seja, um tipo de processo). Cada *Process Type* pode possuir inúmeras instâncias e cada instância se comporta como uma máquina finita de estados que trabalha de forma independente e concorrente com as demais.

Sendo assim, podem ser especializados apenas os tipos *System Type*, *Block Type*, *Process Type* e *Procedure*, sendo que o tipo *System Type* apenas pode ser reutilizado por outro sistema, não podendo ser redefinido. Para a especialização de processos, novas funcionalidades podem ser adicionadas através da inclusão de novos estados e novas transições em estados pré-existentes. Além disso, pode-se redefinir transições de *Input* e de *Start*, declarando-as como *VIRTUAL* no supertipo e *REDEFINED* no subtipo. Este subtipo pode ainda dar origem a um outro tipo que herde todas as suas características (e por conseqüências todas as características de seu ancestral). Neste caso, para este novo tipo, todas as propriedades *REDEFINED* podem ser novamente redefinidas. Para que isso seja evitado, ao invés de definir as propriedades como *REDEFINED* pode-se utilizar *FINALIZED*, proibindo os tipos herdeiros deste subtipo de redefinir estas propriedades.

A.4.1 Package

Um conceito bastante importante em SDL é o *Package*. Ele permite a criação de uma biblioteca que pode ser utilizada em um sistema. No *package* podem ser definidos tipos de dados, listas de sinais, *System Types*, *Block Types*, *Process Types*, etc, podendo ser utilizado integralmente ou parcialmente pelo sistema que o inclua através da construção *USE*.

Além de poder ser incluído em um sistema, um *package* pode ser incluído em outro *package*, criando assim um hierarquia de *packages*.

A.5 Tipos Abstratos de Dados

Os tipos abstratos de dados em SDL são denominados *SORTS*. Existem diversos tipos pré-definidos de *sorts* que permitem a manipulação de dados nas especificações dos processos e dos procedimentos. Estes *sorts* são utilizados através da declaração de variáveis (com a construção *DCL*).

Os *sorts* definidos na recomendação Z.100 [11] são:

- **Boolean:** Pode assumir os valores *true* e *false*.
- **Character:** Representa um caracter ASCII.
- **Duration e Time:** Utilizados para o controle do tempo. *Time* especifica um valor de tempo absoluto, enquanto *Duration* especifica um intervalo de tempo. Um conceito importante associado a estes tipos é o *Timer*. Um *timer* consiste em um objeto que gera um sinal após um certo período de tempo. Para que seja utilizado, seve ser declarado junto com as demais variáveis do processo ou procedimento. Para ativá-lo, utiliza-se a função *set*, para desativá-lo, *reset* e para verificar se um timer está ativo, *active*. Existe ainda uma variável do tipo *Time* pré-definida chamada *Now*, que contém o tempo absoluto do momento.
- **Integer e Natural:** O tipo *integer* especifica valores matemáticos inteiros. Já o tipo *natural* especifica os inteiros não-negativos.
- **Real:** Utilizado para especificar os valores matemáticos reais.
- **Pid:** É um identificador de processo, ou seja, um endereço que identifica um processo dentro de um dado sistema. Cada processo apresenta quatro *Pids* que são automaticamente definidos:

self — identificador do próprio processo;
parent — identificador do processo que o criou;
sender — identificador do processo que enviou o último sinal recebido;
offspring — identificador do último processo criado por ele;

Alguns *sorts* também são definidos na recomendação Z.105 [38]:

- **Bit:** Pode assumir os valores 0 e 1.
- **Bit_String:** Representa uma seqüência de *Bits*.
- **IA5String, NumericString, PrintableString, VisibleString:** São strings com algumas restrições. *IA5String* é igual ao *Charstring*. *NumericString* só aceita os caracteres '0' a '9' e espaço. *PrintableString* e *VisibleString* só aceitam alguns dos caracteres visíveis.
- **Null:** Só pode assumir um valor: nulo.
- **Object_Identifier:** Trata-se de uma seqüência de valores do tipo *Natural* utilizados para identificar algum objeto do sistema. Caso, por exemplo, em um determinado instante deva ser enviada uma mensagem que pergunta a uma entidade se ela suporta um determinado protocolo X, este protocolo X pode ser representado por um *Object_Identifier*.
- **Octet:** Pode assumir um valor de oito bits, isto é, de 0 a 255.
- **Octet_String:** Representa uma seqüência de *Octets*.

Além destes tipos pré-definidos, novos tipos de dados podem ser definidos pelo usuário através do uso dos construtores *Syntype* e *Newtype*.

O *Syntype* possibilita a criação de novos tipos a partir de outros pré-existentes, renomeando-os apenas ou impondo maiores restrições. Pode-se por exemplo definir um tipo *smallint* que pode assumir os valores de 0 a 10 da seguinte forma:

```
syntype smallint = natural
   constants 0:10
endsyntype;
```

Diversos tipos de dados podem ser criados com o uso do construtor *Newtype*, dentre eles:

- **Enumeration:** Trata-se de um tipo que pode assumir diferentes valores de literais pré-definidos.

```
newtype Tempo literals
   ensolarado, chuvoso, nublado
endnewtype;
```

- **Struct:** Possibilita a agregação de dados que sejam relacionados.

```
newtype Pessoa struct
   Nome     charstring;
   Endereco charstring;
   Idade    natural;
endnewtype;
```

- **Bit Fields:** Possibita a definição do tamanho em bits de um campo da *Struct*.

```
newtype Exemplo struct
   a Integer : 4;
   b Integer : 2;
   c Natural : 3;
endnewtype;
```

- **Choice:** Semelhante ao *Union* da linguagem C. Possibilita a criação de um tipo composto de diversos campos, mas que pode assumir o valor de apenas um dos campos por vez.

```
newtype Escolha struct
   a Integer;
   b Charstring;
   c Boolean;
endnewtype;
```

Caso se declare uma variável *var* com o tipo *Escolha*, pode-se fazer $var!a := 1$, $var!b :=' teste'$ ou ainda $var!c := true$. Para se testar qual campo está ativo, basta avaliar a expressão $var!Present$.

- **Array:** Possibilita a criação de um *array* que pode ser indexado por qualquer tipo de dados.

```
newtype A1 Array(Character,Integer)
endnewtype;
```

  Caso se declare uma variável *var* com o tipo *A1*, pode-se fazer $var('a') := 1$.

- **String:** Possibilita a criação de uma sequência de valores de qualquer tipo.

```
newtype S1 String(Integer,Empty)
endnewtype;
```

  Neste caso, trata-se de uma seqüência de valores do tipo *Integer* e a seqüência vazia é representada pelo literal *Empty*.

- **Powerset:** Possibilita a criação de um conjunto de valores de qualquer tipo. Um conjunto vazio é sempre representado pelo literal *Empty*.

```
newtype P1 Powerset(Charstring)
endnewtype;
```

- **Bag:** Possibilita a criação de um conjunto de valores de qualquer tipo, entretanto difere de *Powerset* por permitir a existência do mesmo valor mais de uma vez no conjunto.

```
newtype B1 Bag(Integer)
endnewtype;
```

A.6 SDT - SDL Design Tool

O SDT (*SDL Design Tool*) [15, 16] é uma ferramenta CASE (*Computer Aided Design Software Engineering*), distribuída pela Telelogic, que utiliza a linguagem SDL para a especificação, validação e simulação de sistemas. Dentre os componentes da versão utilizada, o SDT 4.2, pode-se destacar os seguintes:

Organizador: utilizado para a organização geral das especificações. Fornece uma visão de todos os diagramas e documentos que compõem o sistema, os quais podem ser livremente agrupados em capítulos e módulos de forma a facilitar a organização;

Editor SDL: possibilita a edição comportamental do sistema através do uso da representação gráfica da linguagem SDL;

Visualizador de Tipos: permite a visualização do impacto dos mecanismos de herança e especialização no sistema;

Analisador: realiza a análise sintática e semântica da especificação desenvolvida verificando a existência de erros e gerando código para os processos de simulação e validação;

Simulador: permite a simulação das características do sistema, tendo como saída um diagrama de troca de mensagens (MSC - *Message Sequence Chart*) que mostra o comportamento do sistema ao longo do tempo;

Validador: permite a validação do sistema, ou seja, percorre todos os possíveis estados em que o sistema pode se encontrar auxiliando assim na detecção de erros, como por exemplo o não tratamento de algum sinal em determinados estados;

Editor MSC: utilizado pelo simulador para a geração dos diagramas de troca de mensagens;

Visualizador de área coberta: utilizado pelo simulador e pelo validador para exibir uma árvore com todos os estados do sistema destacando aqueles alcançados pelo pelo processo em questão (simulação ou validação);

Geradores de código: possibilitam a geração de código C, Cmicro e CHILL a partir das especificações em SDL.

A presença dos componentes Simulador e Validador são de fundamental importância para a realização de testes da especificação, garantindo assim a corretude do sistema. Através do uso destes componentes pode-se encontrar erros na especificação através da simulação de casos críticos, bem como através da análise dos estados alcançados pelo Validador com o uso do visualizador de área coberta (*Coverage Viewer*).

Além disso, o SDT permite a integração do desenvolvimento em SDL com outras linguagens como o ASN.1 (*Abstract Syntax Notation One*) e UML (*Unifing Modelling Language*), garantindo assim uma maior flexibilidade no processo de desenvolvimento. Existe também a possibilidade da realização de testes de casos de uso através do uso da linguagem TTCN (*Tree and Tabular Combined Notation*). A relação entre estas linguagens é apresentada na

figura A.2. As linguagens presentes em elipses escuras não estão disponibilizadas no pacote utilizado para a realização deste trabalho.

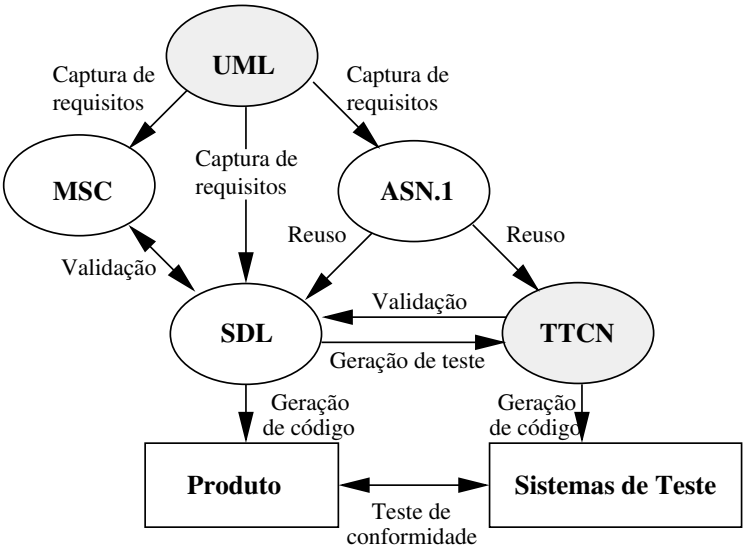


Figura A.2: Relação entre as linguagens que fazem interface com o SDT

Uma visão geral dos módulos componentes da ferramenta e suas relações pode ser visto na figura A.3. As características que apresentadas nas caixas escuras não estão presentes no pacote utilizado para a realização deste trabalho.

O Organizador é responsável por gerenciar quase todos os componentes do pacote SDT. Através dele tem-se acesso à todas as funcionalidades disponíveis. Tanto para a realização da simulação quanto da validação, as especificações passam primeiramente por uma análise (através do módulo Analisador) que garante a corretude sintática das especificações. Após esta análise, é gerado código C, o qual é utilizado pelos módulos responsáveis pela realização das simulações e da validação. Ainda é disponibilizada uma biblioteca de aplicações que, quando integrada ao gerador de código C, permite a geração de uma implementação em C do sistema especificado. execução do sistema especificado.

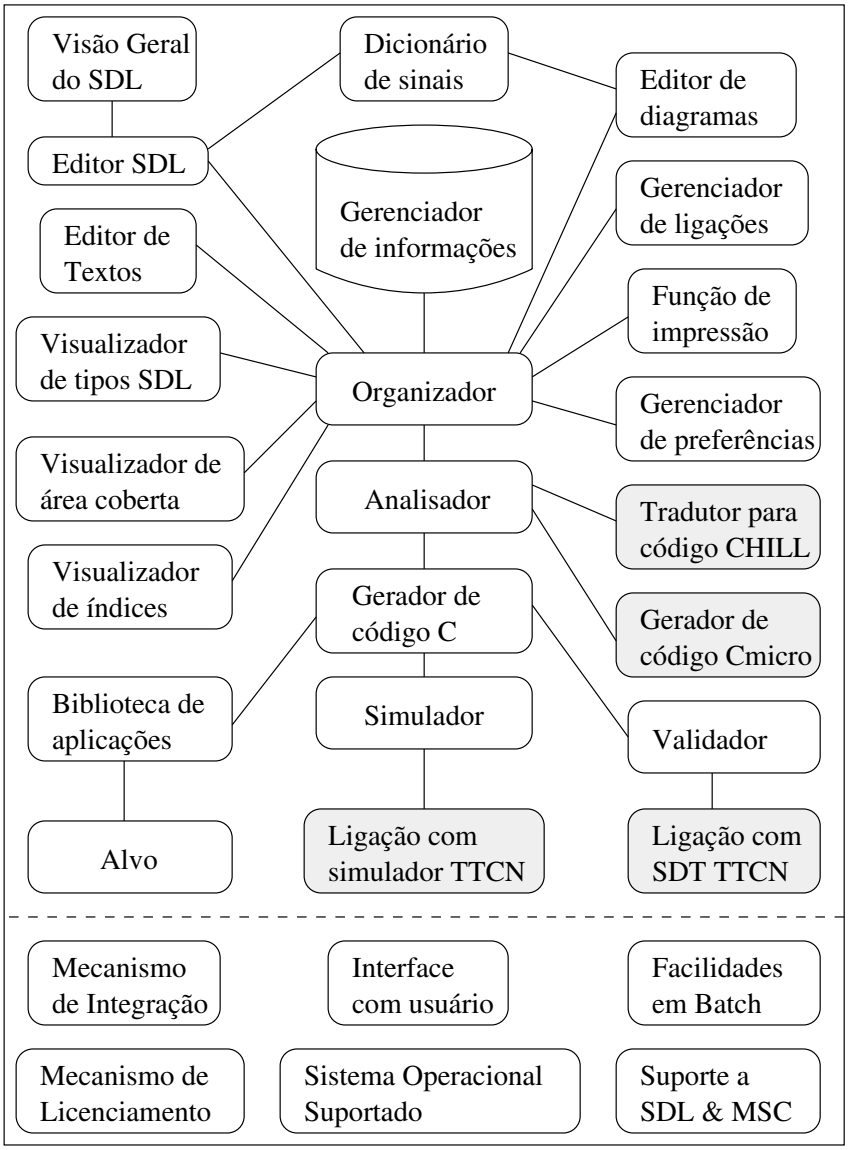


Figura A.3: Visão geral dos módulos que compõem o SDT

Apêndice B

Geração de código e JavaCC

Uma linguagem de programação (como o SDL) pode ser definida através da descrição de como seus programas se parecem (a sintaxe da linguagem) e o que eles significam (a semântica da linguagem).

Para a especificação da sintaxe, a notação mais utilizada é a chamada gramática de livre contexto ou BNF (*Back-Naur Form*).

Já a especificação da semântica de uma linguagem é mais complexa e, por isso, não possui uma notação adequada, devendo ser descrita informalmente, como por exemplo através de linguagem natural.

Além de especificar a sintaxe de uma linguagem, a notação BNF pode ser utilizada para auxiliar a tradução de programas de uma linguagem para a outra, ou seja, para organizar a estrutura de um compilador que realize esta tarefa.

B.1 O gerador de código Java

O gerador de código desenvolvido pode ser visto como um compilador que tem como saída um programa em Java. O processo de compilação possui diversas fases, sendo que saída de uma fase é a entrada de outra.

Dentre as fases mais importantes do processo de compilação, podemos citar em ordem de ocorrência: análise léxica, análise sintática, análise semântica e geração de código.

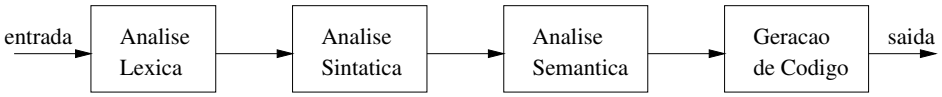


Figura B.1: Fases de um compilador

B.1.1 Análise Léxica

A análise léxica é a primeira fase de um compilador. Sua principal tarefa é a leitura dos caracteres de entrada e a geração de uma seqüência de *tokens* [1] que são utilizados na próxima fase — a análise sintática.

O analisador léxico também pode ser responsável pela retirada de comentários presentes no programa de entrada (que são inúteis para no processo de compilação), bem como de caracteres de espaço, *tab* e *newline*.

Os caracteres da atribuição abaixo:

tamanho := incremento * 10

seriam agrupados nos seguintes tokens:

1. O identificador *tamanho*
2. O símbolo de atribuição *:=*
3. O identificador *incremento*
4. O sinal de multiplicação
5. O número 10

Os espaços em branco são ignorados.

B.1.2 Análise Sintática

Após o processo de análise léxica, é necessário verificar se a seqüência de *tokens* reconhecidos respeita a sintaxe esperada. Para isso, necessita-se representar essa sintaxe através do uso da notação BNF.

Uma chamada a um procedimento em SDL tem a seguinte forma:

call *identificador* (*lista de parâmetros*) to *expressão*

[1] *Token*: Seqüência de caracteres que possuem algum significado

Isto significa que a chamada a um procedimento é a concatenação da palavra reservada *call*, de um abre-parênteses, de uma lista de parâmetros, de um fecha-parênteses, da palavra reservada *to* e de uma expressão. Utilizando as variáveis *id* para representar o identificador, *param* para representar a lista de parâmetros, *expr* para representar a expressão e *prcdcall* para representar a chamada ao procedimento em si, a estrutura acima pode ser representada por:

$$prcdcall \rightarrow \text{call } id \text{ (} param \text{) to } expr$$

onde a seta pode ser lida como "pode ser representada por". Uma regra como esta é chamada produção e elementos como *id*, *param* e *expr* são chamados não-terminais, pois se referem a alguma outra produção. Além dos terminais, a notação BNF possui mais três tipos de elementos: os terminais (*call*, *to*, abre e fecha parênteses), as produções (um não terminal, seguido de uma seta, seguido de uma sequência de terminais e não-terminais) e uma produção inicial de onde deve-se iniciar o processo de reconhecimento da linguagem.

Uma expressão matemática que envolva as operações de soma e subtração, por exemplo, pode ser reconhecida pela seguite representação BNF (sendo *expr* a produção inicial):

1: $expr \rightarrow expr + numero$

2: $expr \rightarrow expr - numero$

3: $expr \rightarrow numero$

No caso acima, assume-se que o token *numero* já é reconhecido pelo analisador léxico como uma seqüência de dígitos. Pode-se assim, deduzir que 9+32-4 é uma *expr*, uma vez que (vide figura B.1.2):

1: 9 é uma *expr*, de acordo com a produção 3;

2: $9 + 32$ é uma *expr*, de acordo com a produção 1;

3: $9 + 32 - 4$ é uma *expr*, de acordo com a produção 2.

Durante o processo de reconhecimento do programa de entrada (em SDL) através da análise sintática, as estruturas do programa são armazenadas em memória em uma hierarquia lógica de forma a facilitar a próxima fase — a geração de código.

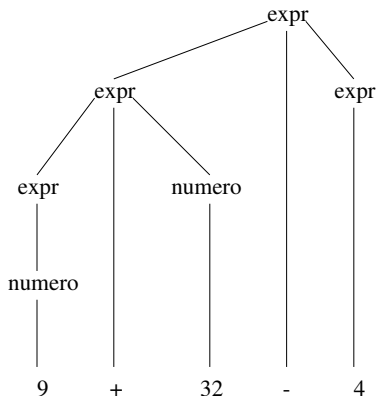


Figura B.2: Reconhecimento de uma expressão

B.1.3 Geração de código

A fase final do compilador é a geração de código, isto é, a transformação do programa reconhecido em alguma coisa executável (ou em um programa escrito na linguagem definida como destino). No caso do gerador de código desenvolvido no trabalho, esta etapa é reponsável pela geração do código distribuído Java que representa o programa em SDL reconhecido pela ferramenta.

B.2 O JavaCC

Neste contexto, o JavaCC (*Java Compiler Compiler*) se insere como uma ferramenta que auxilia na construção de um compilador. Através de uma sintaxe simples, o JavaCC constrói um compilador, dada a representação em notação BNF da linguagem a ser reconhecida.

Além disso, uma série de comandos em Java podem ser inseridos após o reconhecimento de cada um terminal/não-terminal para que se possa tomar ações de acordo com o que for reconhecido. Desta forma, pode-se armazenar uma representação do programa reconhecido para que em uma etapa posterior se gere código Java a partir destas representações.

No JavaCC, algumas extensões à notação BNF podem ser utilizadas, são elas:

1. (expr)+ – Indica que o não-terminal expr pode ocorrer uma ou mais vezes;
2. (expr)* – Indica que o não-terminal expr pode ocorrer zero ou mais vezes;
3. (expr)? – Indica que o não-terminal pode ou não ocorrer
4. (expr1 | expr2) – Indica que pode ocorrer o não-terminal expr1 ou expr2;

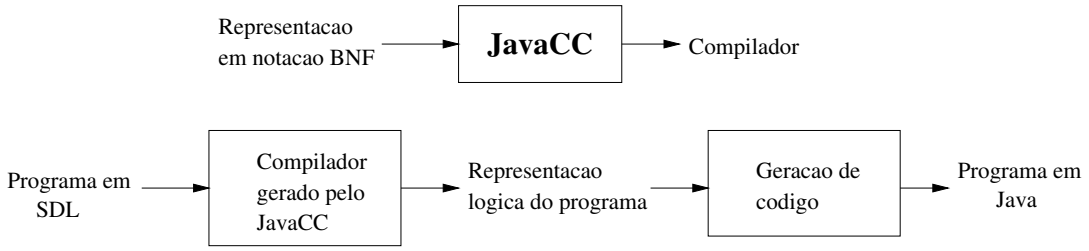


Figura B.3: Construção de um compilador com o uso de JavaCC

Para o reconhecimento dos programas em SDL foi convertida para o padrão JavaCC a gramática livre de contexto descrita no documento Z.100 [11] do ITU-T que padroniza a linguagem SDL.

Apêndice C

Artigos Publicados

C.1 Anais do 19o Simpósio Brasileiro de Telecomunicações (SBRT) - Fortaleza/CE - Set/2001

UMA PROPOSTA DE ESPECIFICAÇÃO FORMAL EM SDL DE UMA SESSÃO DE ACESSO TINA

Rafael P. Guimarães, Walter C. Borelli
{rguima, borelli} @dt.fee.unicamp.br

DT – FEEC - Unicamp
Caixa Postal 6101
Campinas, SP, Brasil – Cep 13083-970

RESUMO

O trabalho apresentado neste artigo consiste na especificação formal orientada a objetos da Sessão de Acesso em uma arquitetura de serviços TINA (*Telecommunications Information Networking Architecture*) utilizando SDL (*Specification and Description Language*) com o objetivo de conferir um caráter comportamental ausente nas especificações em ODL (*Object Description Language*) descrita nas recomendações TINA. Sendo assim, é proposto neste artigo um novo mapeamento de ODL/IDL para SDL, com o qual é possível modelar os objetos componentes da arquitetura TINA de uma forma mais adequada.

Após a especificação proposta para a Sessão de Acesso TINA, são apresentados exemplos de simulações e sua completa validação.

Palavras-Chave: TINA, SDL, Arquitetura de Serviços

ABSTRACT

The work presented in this paper consists on the object oriented formal specification of the Access Session on a TINA Service Architecture using SDL, with the intention to provide a behaviour description which is not presented in the ODL specifications described on TINA recommendations. Then, it is proposed in this paper a new mapping from ODL/IDL to SDL, so that it is possible to model the TINA objects in a better way.

After the proposed specification for the TINA Access Session, it is presented examples of simulations and its full validation.

Keywords: TINA, SDL, Service Architecture

1. INTRODUÇÃO

O consórcio TINA (*Telecommunications Information Networking Architecture*) define uma arquitetura distribuída aberta para o provimento e gerenciamento de serviços de telecomunicações. A proposta TINA [1] é de disponibilizar um ambiente distribuído bem estruturado onde possa se oferecer desde os serviços mais simples até os mais complexos existentes.

Para simplificar a análise do problema, a arquitetura TINA é separada logicamente em 3 arquiteturas principais (que agrupam diferentes conceitos): a arquitetura de serviços, a arquitetura de rede e a arquitetura de computação.

A arquitetura de serviços [2], que é o foco principal deste artigo, define uma série de conceitos e princípios para a criação, implementação e gerência de serviços de telecomunicações. A arquitetura de rede define uma série de conceitos e princípios para a criação, implementação e gerência de redes de transporte. Finalmente, a arquitetura de computação define uma série de conceitos e princípios para a criação de sistemas distribuídos e de um ambiente de suporte aos sistemas.

Desta forma, uma série de objetos computacionais são especificados nas recomendações TINA afim de fornecer o suporte necessário para a realização desta arquitetura. Estes objetos são descritos através da linguagem ODL [3] (*Object Description Language*), que especifica as interfaces deste objetos, deixando suas características comportamentais em aberto.

Afim de fornecer uma especificação mais completa para estes objetos, conferindo a estas especificações um caráter comportamental e orientado a objetos, este artigo trata da aplicação da linguagem SDL [4] (*Specification and Description Language*) na especificação da Sessão de Acesso de uma Arquitetura de Serviços TINA. A escolha do SDL deve-se principalmente ao fato da mesma se tratar de uma linguagem para a especificação formal de sistemas e protocolos proposta pela ITU já consolidada e bastante utilizada no meio das empresas de telecomunicações, que contam com o suporte de ferramentas profissionais para o desenvolvimento, como o SDT[1] [5] por exemplo, utilizado no trabalho apresentado neste artigo.

Para a elaboração destas especificações em SDL, foi necessário o desenvolvimento de um mapeamento entre as estruturas ODL e as estruturas SDL, baseado em outros discutidos nos artigos [6] e [7], porém com algumas contribuições adicionais.

Este artigo discute os pontos principais da especificação em SDL da Sessão de Acesso TINA, finalizando com a apresentação de sua validação e de alguns exemplos de simulações de suas funcionalidades.

2. A ARQUITETURA DE SERVIÇOS TINA

A arquitetura de serviços define um conjunto de conceitos e princípios que possibilitam a concepção, a implementação, o uso e a operação de serviços de telecomunicações. Para isto, esta arquitetura define componentes reusáveis com os quais se constrói os serviços de telecomunicações.

A arquitetura de serviços TINA faz uso do conceito de sessão, o qual pode ser definido como uma relação temporária entre um grupo de recursos que irão conjuntamente efetuar alguma tarefa

[1] *SDT (SDL Design Tool) v.4.1: adquirido pelo DT/FEEC/UNICAMP através de um projeto temático FAPESP (Proc. 91/3660-0)*

por um período de tempo. São três os tipos principais de sessão: Sessão de Acesso, Sessão de Serviço e Sessão de Comunicação.

A Sessão de Acesso, que é especificada neste artigo, promove o acesso do usuário aos serviços disponibilizados pelo provedor. Para isso a sessão de acesso engloba a autenticação do usuário, troca de informações acerca das capacidades do usuário junto ao provedor, para que finalmente o usuário possa acessar serviços e se unir a sessões de serviço já existentes, quando convidado.

A Sessão de Serviço é responsável por prover um serviço dentre os disponíveis. É composta por uma instanciação do serviço a ser acessado e possui informações necessárias para o uso do mesmo.

Finalmente, a Sessão de Comunicação provê os recursos de comunicação necessários para o estabelecimento de conexões de fluxo de dados (*stream*) entre os participantes de uma sessão de serviço.

Na Figura 1 são apresentados os objetos participantes da Sessão de Acesso TINA e seus relacionamentos.

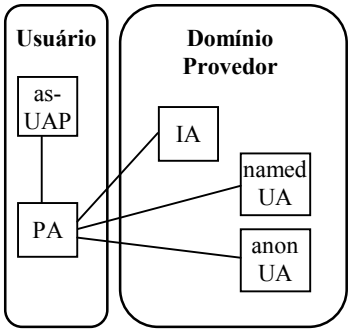


Figura 1. Interação entre componentes da Sessão de Acesso da Arquitetura de Serviços TINA

Os componentes relacionados com a sessão de acesso são independentes do serviço oferecido. O as-UAP (*access session User Application*) é responsável pela interface homem-máquina, provendo ao usuário mecanismos para se comunicar com o provedor. O PA (*Provider Agent*), por sua vez, é o representante do provedor no domínio do usuário, sendo responsável pela intermediação entre o as-UAP e o domínio do provedor. No domínio do provedor, o IA (*Initial Agent*) é o ponto de contanto inicial do usuário com o provedor, para que seja estabelecida uma sessão de acesso. O UA (*User Agent*), por sua vez, é o representante do usuário no domínio do provedor e suporta mecanismo de acesso aos diversos serviços disponibilizados – pode ser do tipo anonUA quando o usuário requisita um acesso anônimo, ou namedUA quando o usuário se identifica fornecendo, por exemplo, um *username* e uma senha. Apesar de um usuário poder ter mais de uma sessão de acesso com o provedor, existe um único UA por usuário no provedor.

3. ESPECIFICAÇÃO EM SDL DA SESSÃO DE ACESSO TINA

Nas recomendações TINA, os objetos apresentados na sessão anterior são descritos utilizando-se a linguagem ODL, que é uma linguagem de descrição das interfaces do objeto, praticamente igual à linguagem IDL (*Interface Description Language*).

Sendo assim, para que este sistema possa ser descrito em SDL, algumas regras de mapeamento da linguagem ODL (ou IDL) para a linguagem SDL foram estabelecidas (Tabela 1).

Uma das maiores vantagens do mapeamento proposto neste artigo sobre o apresentado em [6] é o fato dele permitir a existência de exceções no modelo SDL, o que é de fundamental importância em um sistema distribuído (como é o caso da arquitetura de serviços TINA). Além disso, trata-se de um modelo mais próximo do real quando comparado ao apresentado em [7], por utilizar procedimentos exportados para representar métodos dos objetos.

Estruturas ODL/IDL	Mapeamento em SDL
Group Type	Block Type
Object Type	Block Type
Interface Type	Process Type
Referência de objeto	Pid[2]
Operação *oneway* (assíncrona)	Sinal precedido por pCALL_
Operação (síncrona)	Procedimento exportado, onde o valore retornado é do tipo *choice*, ou seja, pode assumir valores de tipos diferentes, um para o resultado, outro para ocorrência de exceção
Enum	NewType com os literais correspondentes
Typedef	Syntype
Struct	NewType com a estrutura correspondente
Constant	Synonym

Tabela 1. Mapeamento de ODL para SDL

Sendo assim, seguindo este mapeamento, os objetos as-UAP, PA, IA, namedUA e anonUA são modelados como *Block Types* (Figura 2) e trocam sinais entre si da forma mostrada na figura 1.

Adicionalmente, no modelo SDL foram implementados os objetos NameServer, IdDispatcher e Creator. Estes objetos não estão dentro das especificações TINA, porém auxiliam a implementação da sessão de acesso.

O bloco NameServer é um Servidor de Nomes, cuja única funcionalidade é fornecer ao usuário uma referência ao objeto IA do provedor desejado quando solicitado, para que se possa estabelecer uma comunicação entre o usuário e o provedor. Para isto o objeto PA faz uso de um método do Servidor de Nomes chamado *getReference*. Para que o Servidor de Nomes possua referências para os objetos IA, sempre que um novo IA é instanciado, este se registra junto ao Servidor de Nomes através do método *registerReference*.

[2] *Pid: Process Identification*, corresponde a uma referência à localização do processo (identificador do processo)

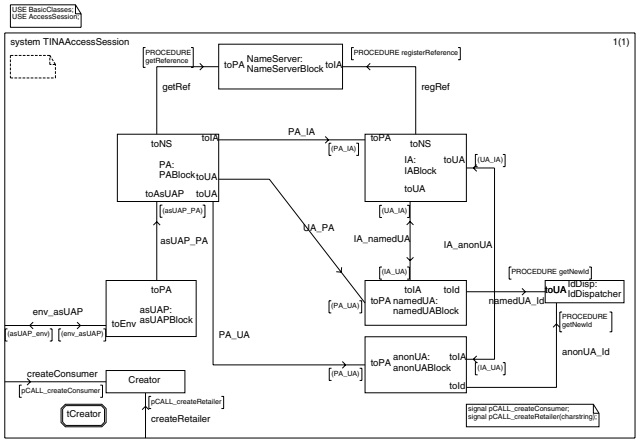


Figura 2. Modelo SDL da Sessão de Acesso TINA

O bloco IdDispacher, por sua vez, é responsável apenas por gerar números seqüenciais (através do método *getNewId*) que serão utilizados pelo namedUA e anonUA como identificadores de sessões de acesso. Finalmente, tem-se o objeto criador – *Creator* – (Figura 3), cujo único objetivo é tornar mais prática a instanciação de novos consumidores e provedores durante a simulação do modelo, através apenas de dois sinais: *createConsumer* e *createRetailer*.

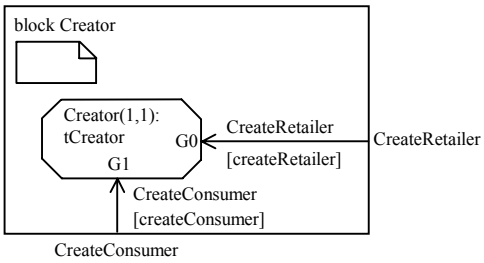


Figura 3. Funcionalidade do bloco (objeto) Creator

Quanto aos objetos pertencentes à arquitetura de serviços TINA, todos eles foram organizados da mesma forma. Foram mapeados em blocos SDL e suas interfaces em processos SDL. Entretanto, como um objeto especificado em ODL pode ser criado dinamicamente, enquanto um bloco SDL não pode ser instanciado (apenas processos SDL podem), para cada bloco SDL, tem-se além dos processos que representam suas interfaces, um processo pré-existente que é responsável pela "instanciação do bloco" – um processo criador (Figura 4). Este processo instancia cada um dos processos do bloco e após a criação dos mesmos envia a cada um deles as referências de todos os outros processos componentes deste objeto instanciado, de forma que os processos possam se comunicar (uma vez que conhecem os *Pid* uns dos outros), e assim atuar como uma unidade, um objeto.

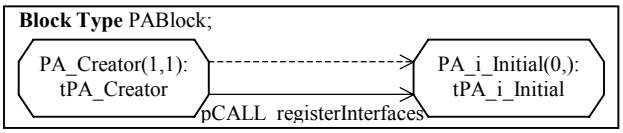


Figura 4. Solução para a instanciação de blocos SDL

A Figura 5 mostra a especificação em SDL bloco PA, com a presença do processo criador. Este processo contém apenas um procedimento responsável pela instanciação do objeto. Procedimento este que instancia as interfaces do objeto e envia a cada uma delas a referência das outras (através do sinal *pCALL_registerInterfaces*, que representa uma operação assíncrona) para que as interfaces possam interagir umas com as outras, se comportando como um único objeto. Além disso, o processo criador ainda pode fornecer outras informações relevantes às interfaces no momento de sua criação.

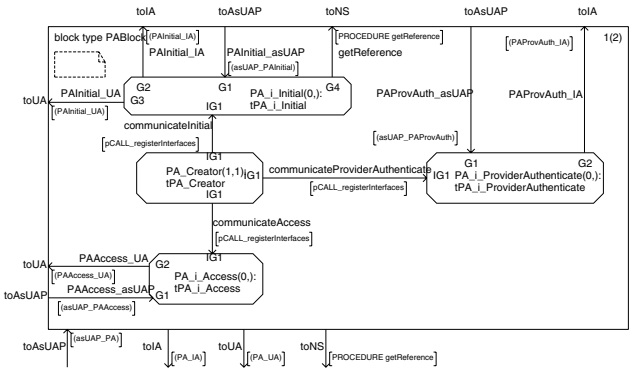


Figura 5. Bloco PA

A Figura 6 mostra o processo *PA_i_Initial*, que, assim como todos os demais processos, possui seu conjunto de procedimentos exportados (representando os métodos) que podem ser acessados por quaisquer outros blocos. Da mesma forma, todas as interfaces possuem um conjunto de procedimentos importados de outras interfaces (do mesmo objeto ou de outro), que representam os métodos que as mesmas podem acessar.

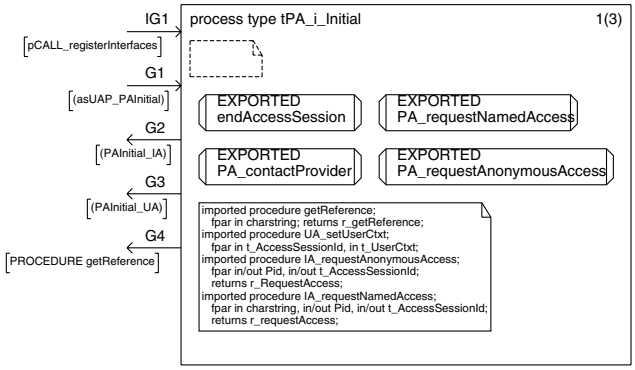


Figura 6. Interface i_Initial do objeto PA

A Figura 7 apresenta o procedimento exportado *PA_contactProvider*, no qual é definida a especificação comportamental do método por ele representado. Esta especificação não se encontra no modelo ODL, uma vez que o ODL se limita à especificar os objetos e suas interfaces, sem se preocupar com a dinâmica dos mesmos.

Este procedimento exportado é responsável por contactar o provedor cujo nome é fornecido através do parâmetro *desiredProviderParam*. Para isto, é chamado o procedimento

remoto *getReference* do bloco *NameServer* (o qual foi previamente importado pelo processo em questão, como pode ser visto na Figura 6). O resultado deste procedimento é retornado, informando o endereço do IA do provedor, ou então, retornando uma exceção (quando for o caso).

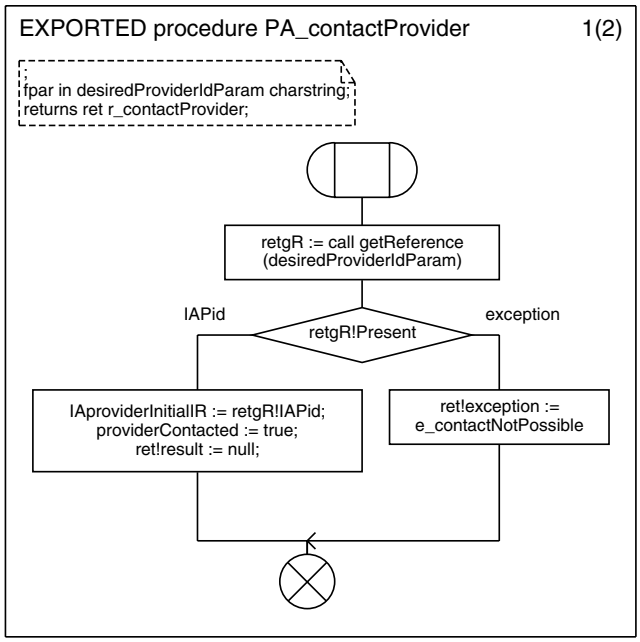


Figura 7. Procedimento exportado PA_contactProvider

O mapeamento proposto neste artigo se distingue de outras propostas ([6],[7]) ao estabelecer que para cada método (procedimento exportado) que possa gerar uma exceção, é definido um tipo de dados para o resultado do método. Este tipo é criado com o gerador *choice*, que permite que a uma mesma variável possam ser atribuídos valores de tipos distintos. Na Figura 8 tem-se um exemplo da definição do tipo do resultado do método (ou processo exportado) *PA_contactProvider*. O resultado pode assumir valores do tipo *null* ou do tipo *e_ContactProvider*. Neste exemplo, o método não retorna nenhum valor, assim, caso tudo ocorra corretamente, o resultado do método terá o seu componente *result* definido para *null*. Caso ocorra uma exceção, o resultado terá o seu componente *exception* definido para a exceção ocorrida (do tipo *e_ContactProvider*).

```
NewType r_contactProvider choice
   result null;
   exception e_ContactProvider;
EndNewType r_contactProvider;
```

Figura 8. Definição do tipo do resultado do método *PA_contactProvider*

O objeto que invocou este método, pode facilmente verificar a ocorrência da exceção, verificando qual componente do resultado está definido (Figura 9) e seguindo o fluxo normal ou executando os procedimentos pertinentes à ocorrência de uma exceção (a exceção pode ser identificada através do componente *exception*).

Todos os outros blocos (as-UAP, IA, anonUA e namedUA) foram organizados da mesma forma.

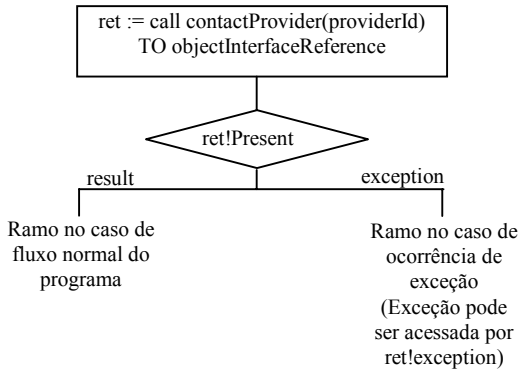


Figura 9. Detecção de ocorrência de exceção

Além disso, pode-se destacar o fato dos blocos namedUA e anonUA possuírem funcionalidades bastante similares, pois ambos representam os usuários (anônimo ou não) no domínio do provedor. Esta similaridade entre blocos, possibilitou a criação de um bloco chamado UABlock, o qual nunca é instanciado, e que possui as características (processos e procedimentos) comuns entre os blocos anonUA e namedUA (Figura 10). Desta forma, os blocos anonUA e namedUA são definidos como herdeiros do bloco UABlock, o que lhes permite partilhar de suas características comuns e adicionar às mesmas particularidades de seu funcionamento, através de redefinição de alguns dos procedimentos herdados.

Dentre os procedimentos comuns aos blocos namedUA e anonUA, que puderam ser definidos em UABlock, pode-se destacar os procedimentos responsáveis por listar os serviços disponíveis para o usuário e finalizar uma sessão de acesso. Dentre os procedimentos específicos dos blocos, pode-se citar o reponsável pelo estabelecimento de uma nova sessão de acesso, uma vez que no bloco namedUA é necessário realizar a autenticação do usuário, enquanto no bloco anonUA não, já que ele trata apenas das sessões de acesso anônimas.

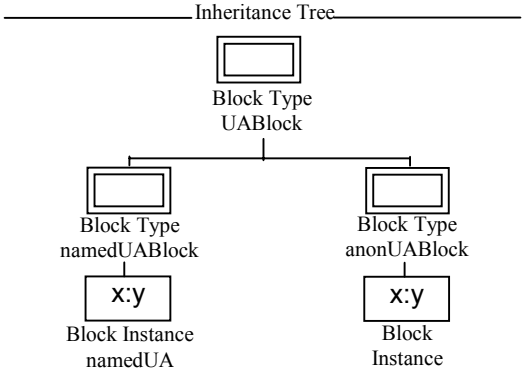


Figura 10. Herança dos blocos namedUA e anonUA

4. VALIDAÇÃO E SIMULAÇÕES DA SESSÃO DE ACESSO TINA

No modelo de sessão de acesso TINA desenvolvido, todas as funcionalidades esperadas foram cobertas e totalmente testadas:

- Contato inicial com o Provedor ou *Retailer*, através do IA, cuja referência é obtida junto ao Servidor de Nomes;
- Estabelecimento de sessões de acesso identificadas (namedUA), incluindo autenticação do usuário;
- Estabelecimento de sessões de acesso anônimas (anonUA);
- Obtenção de lista de serviços disponibilizados para o usuário que foi autenticado (ou usuário anônimo);
- Finalização das sessões estabelecidas (processo de *logout*).

O sistema foi completamente validado, utilizando-se a ferramenta *Validator* da suíte SDT da Telelogic para desenvolvimento em SDL, sendo possível a detecção de pequenos erros na especificação, como por exemplo, uma comparação errada que era realizada no processo *UA_endAccessSession* do bloco UA.

Tal erro pode facilmente ser descoberto após a execução do *Validator*, notando-se que nenhum símbolo abaixo de um determinado ramo da comparação havia sido alcançado (Figura 11). Após a correção desta comparação, o sistema foi novamente validado e todos os símbolos esperados foram alcançados.

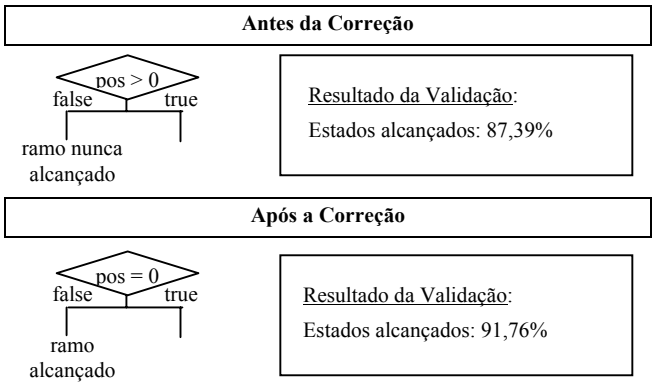


Figura 11. Detecção de erros através da validação

Nem todos os símbolos (aproximadamente 8%) foram alcançados durante a validação, pois alguns deles tratam situações de erro que poderiam ocorrer no sistema real, mas que não ocorrem em uma situação ideal. Um exemplo é a possibilidade de um PA se comunicar com o namedUA errado. Dada a especificação do PA, isto é impossível porém, há no namedUA um tratamento especial para este tipo de situação. Além disso, alguns símbolos do bloco UA (definidos como VIRTUAL) foram redefinidos em anonUA e namedUA, sendo assim, nunca são alcançados.

Após a completa validação do sistema, foram simuladas as principais funcionalidades para a garantia da validade do sistema. Para tanto foi utilizada a ferramenta *Simulator*, que também é parte integrante do SDT. Esta ferramenta permite o acompanhamento total da simulação gerando diagramas de trocas de mensagens (MSC – *Message Sequence Charts*), que permitem visualizar a interação entre os diversos componentes do sistema.

Na Figura 12, pode-se visualizar um destes diagramas gerados.

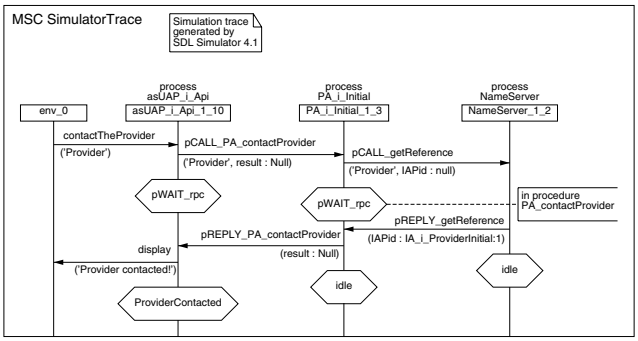


Figura 12. Contato inicial com o provedor

Durante o processo inicial, o usuário informa o desejo de contactar o provedor na inteface disponibilizada pelo as-UAP, aqui representado pelo envio do sinal *contactTheProvider* enviado à interface *i_Api* do objeto as-UAP. O objeto repassa esse pedido para o PA através de sua interface *i_Initial* e este contacta o Servidor de Nomes (através do procedimento *getReference*) afim de obter uma referência ao IA do provedor desejado. O Servidor de Nomes retorna a referência ao PA, que fica sabendo como contactar o provedor. Uma mensagem de sucesso é exibida ao usuário, a qual é aqui representada pelo sinal *display*.

Em um segundo passo, o usuário deve solicitar uma sessão de acesso ao provedor (que já pode ser contactado, uma vez que o PA possui uma referência para IA). Esta sessão pode ser anônima ou identificada. No caso de uma sessão anônima (Figura 13), o usuário solicita a sessão através de alguma interface gráfica – aqui representada pelo sinal *requestAccess* da interface *i_Api*. Essa solicitação é repassada para o PA através do método *PA_requestAnonymousAccess* da interface *i_Initia*l, que a repassa para o objeto IA através do método *IA_requestAnonymousAccess* da interface *i_ProviderInitia*l.

O método *IA_requestNamedAccess* solicita então ao processo anonUA_Creator a instanciação do objeto anonUA, que passará a ser o representante do usuário no domínio do provedor. Após a instanciação, o anonUA_Creator passa algumas informações relevantes às interfaces que compõem o anonUA (como por exemplo os endereços das outras interfaces).

O IA ainda solicita ao anonUA recém-criado o estabelecimento de uma sessão de acesso através de uma chamada ao método *UA_setupAccessSession* da interface *i_Initia*l. Este método solicita então ao objeto IdDispatcher um identificador para a sessão a ser estabelecida através do método *getReferenc*e.

Uma vez aceita a nova sessão, o PA repassa ao anonUA informações de contexto do usuário (capacidades do terminal usado, algumas interfaces importantes, etc) através do método *UA_setUserCtxt* da interface *i_ProviderNamedAcces*s. Só aí o as-UAP recebe uma confirmação do estabelecimento da sessão de acesso (através do resultado do método *PA_requestAcces*s) juntamente com um identificador da mesma, o *asI*d. Finalmente o as-UAP pode exibir ao usuário uma mensagem de sucesso do estabelecimento da sessão de acesso, representada aqui pelo sinal *displayAsI*d.

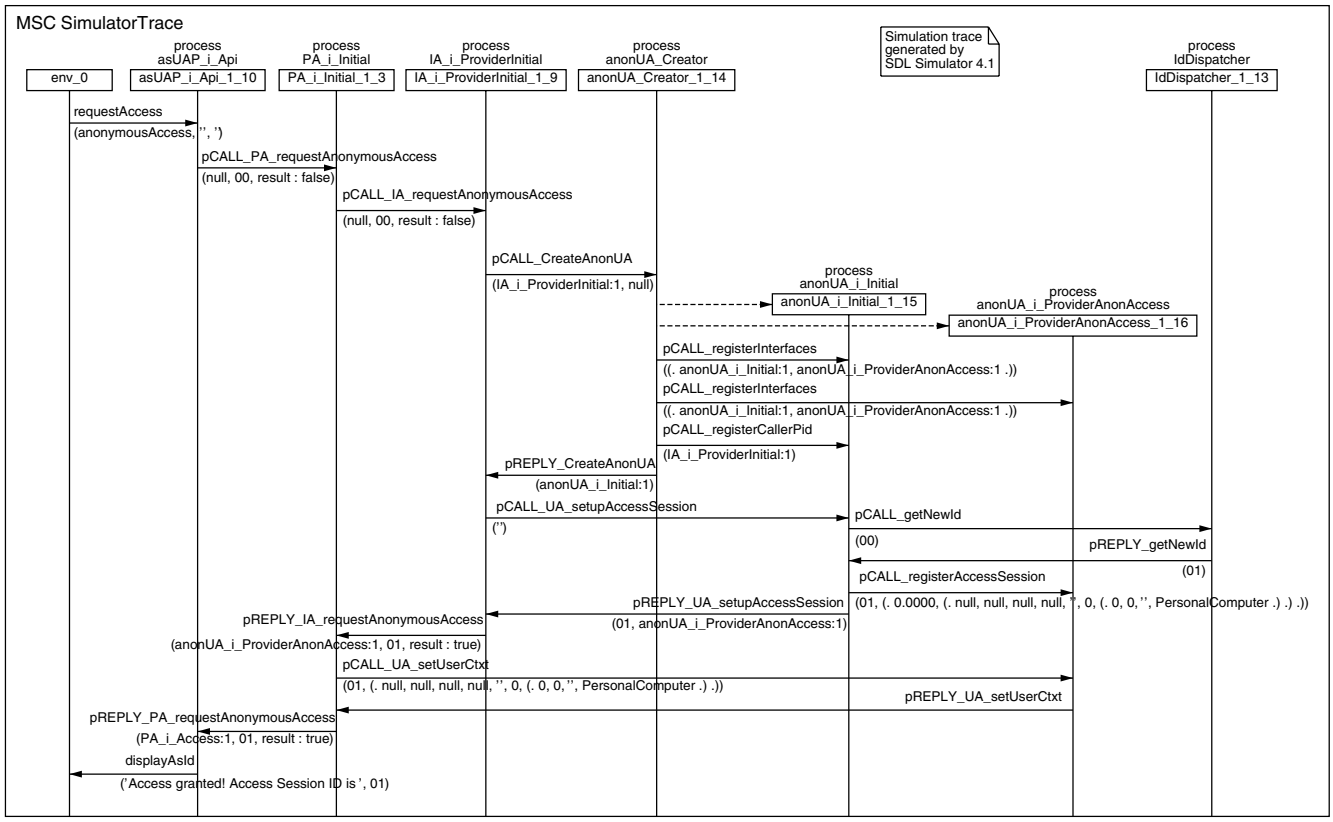


Figura 13. Estabelecimento de uma sessão de acesso anônima

5. CONCLUSÃO

O trabalho apresentado neste artigo propõe uma especificação formal da Sessão de Acesso em uma Arquitetura de Serviços TINA. Para tanto, é introduzido um mapeamento de ODL para SDL mais adequado, pois se aproxima mais do conceito de um objeto real com múltiplas interfaces, através do mapeamento de seus métodos para procedimentos exportados, com a possibilidade do tratamento das exceções ocorridas.

Utilizando este mapeamento, foi construído um modelo orientado a objetos em SDL da Sessão de Acesso TINA, conferindo à mesma uma formalidade também em seus aspectos comportamentais, frente à formalidade conferida pelo ODL (que se resume à especificação das interfaces dos objetos envolvidos). Sendo assim, a especificação em SDL permite a observação de toda a dinâmica envolvida no processos da sessão de acesso, possibilitando a simulação de diferentes funcionalidades do modelo e sua completa validação.

Para tanto, fez-se uso do conjunto de ferramentas SDT, utilizando-se o *Simulator* para a simulação dos casos de uso mais importantes da sessão de acesso e o *Validator* para a validação completa do sistema. Utilizando-se de tais ferramentas, todas as possíveis situações puderam ser testadas, o que garante a validade do modelo proposto. Pode-se desta forma, facilitar o processo de uma eventual implementação, uma vez que a lógica do sistema já está verificada e não possui erros.

6. REFERÊNCIAS

[1] Chapman, M., Montesi, S. *Overall Concepts and Principles of TINA*. TINA Consortium, February 1995.

[2] Abarca, C. et al. *Service Architecture – Version 5.0*. TINA Consortium, June 1997.

[3] Parhar, A. *TINA Object Definition Language Manual – Version 2.3*. TINA Consortium, July 1996.

[4] ITU-T (CCITT Recommendation Z.100). *Specification and Description Language*. Geneve, Switzerland, 1993.

[5] Telelogic AB. *Telelogic Tau 4.1 SDL Suite Getting Started*. Sweden, September 2000.

[6] Björkander, M. *Mapping IDL to SDL*. Telelogic AB, 1997.

[7] Kolberg, M., Sinnott, R., Magill, E. *Experiences modelling and using formal object-oriented telecommunication service frameworks*. Computer Networks: The International Journal of Computer and Telecommunications Networking, vol. 31, pp. 2577-2592, December 1999.

[8] Olsen A. et al. *The Pros and Cons of using SDL for creation of Distributed Services*. In: Proceedings of the Sixth International Conference on Intelligence in Services and Networks (IS&N'99), Barcelona, Spain, 1999.

[9] Sherratt, E., Loftus, C. *Designing distributed services with SDL*. IEEE Concurrency, vol. 08, n. 01, pp. 59-66, January-March 2000.

C.2 Anais do X Congreso Internacional de Computación (CIC'2001) - Cidade do México/DF - México - Nov/2001

Object Oriented Development of TINA systems using SDL and Java

Rafael Paoliello Guimarães and Walter da Cunha Borelli

DT – FEEC – Unicamp
Caixa Postal 6101 – Campinas – SP – Brasil
E-mail: {rguima, borelli} @dt.fee.unicamp.br

Abstract. The work presented in this paper consists on a methodology to develop TINA systems by the use of SDL and Java. In order to example the method it will be presented the object oriented SDL formal specification of the Access Session on a TINA Service Architecture. Through this specification we provide a behavioral description to the involved TINA objects, what is not presented in the ODL specifications, described on TINA recommendations. In order to create this SDL model, it is proposed a new mapping from ODL/IDL to SDL, which will help us modeling TINA objects in a better way. After the presentation of simulation examples and the system's full validation, we briefly describe the tool developed to generate Java code from the SDL specifications using CORBA as the target DPE.

Keywords: TINA, SDL, Service Architecture, Java, CORBA.

1 Introduction

Over the past years, the TINA (Telecommunications Information Networking Architecture) Consortium designed an open distributed architecture with the objective to provide and manage telecommunication services. The TINA proposal [1] is to provide a well-structured distributed environment on which it may be possible to offer all kinds of services.

In order to simplify the problem analysis, TINA architecture is logically divided into 3 main architectures (each of them grouping different concepts): the service architeture, the network resources architecture and the computing architecture.

The service architeture [2], which is the main focus of this paper, defines concepts and principles for the creation, implementation and management of telecommunication services. The network resources architecture defines concepts and principles for the creation, implementation and management of transport networks. Finally, the computing architecture defines concepts and principles for the creation of distributed systems and a supporting environment for those systems.

In order to provide the necessary support for the creation of this architecture, some computational objects are specified in TINA recommendations. These objects are specified using ODL [3] (Object Description Language), which fully defines the object interfaces, leaving its behavioral characteristics unspecified.

This paper presents a methodology for the development of TINA objects, providing them with an object oriented behavioral specification, making it easier to eventually create an implementation. For the purpose of exemplifying this methodology, this paper covers the use of SDL [4] (Specification and Description Language) for the formal specification of the Access Session in a TINA Service Architecture.
The choice of SDL may be explained by the fact that it is being largely used for the specification of systems and protocols by telecommunication companies worldwide. The development process is made easier due to the support provided by professional development tools, such as SDT[1] [5], which is the one used in this work.
For the creation of the SDL specifications, it was necessary the development of some mapping rules from the ODL structures into SDL structures. This mapping was based on previous ones discussed in [6] and [7], but a few contributions were added.
This paper discusses the SDL specification of a TINA Access Session, presents its validation and simulation examples and finally discusses some issues on the Java code generation based on the SDL specifications, using CORBA (Common Object Request Broker) as the target DPE (Distributed Process Environment).

2 TINA Service Architecture

The TINA service architecture defines concepts and principles that makes it possible to create, implement, use and operate telecomunication services. This architecture defines reusable components from which telecommunication services are built.
The TINA service architecture is based on the session concept, which may be defined as the temporary relation among a group of resources that together will perform some task over a period of time. There are 3 kinds of sessions: Access Session, Service Session and Communication Session.
The Access Session (specified in this paper) promotes user access to the avaiable services. In order to accomplish this, the Access Session deals with user authentication and exchange of information about user capacities with the provider, so that the user may access services or join pre-existent service sessions. The Service Session is responsible for providing a service among the avaiable ones. The Communication Session provides communication resources so that stream connections may be established. Figure 1 shows objects needed for the creation of a TINA Access Session.

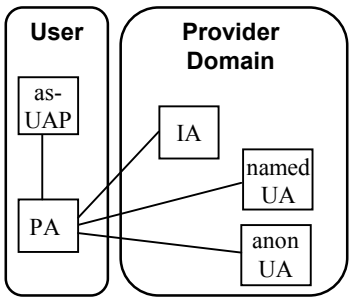


Figure 1. Interaction among the TINA Access Session components.

[1] *SDT (SDL Design Tool) v.4.1: obtained by DT/FEEC/UN CAMP through a FAPESP thematic project Proc. 91/3660-0*

All access session related components are independent of the provided services.. The as-UAP (access session User Application) is responsible for the human-machine interface, providing ways for the user to communicate to the provider. The PA (Provider Agent) represents the provider in the user's domain, being responsible for the communication between the as-UAP and the provider's domain. In the provider's domain, the IA (Initial Agent) provides the user's initial contact to the provider. Finally, the UA (User Agent) represents the user in the provider's domain and supports access mechanisms for all available services. It can be of two different kinds: anonUA, when the user desires an anonymous access, or namedUA, when the user identifies itself, providing (for example) its username and password.

3 SDL Specification of the TINA Access Session

In TINA recommendations, the objects presented above are described using ODL, which is an interface description language very similar to IDL.
So, in order to achieve the objective to specify the system using SDL, some ODL (or IDL) to SDL mapping rules must be defined (Table 1).

ODL/IDL Structures	SDL Mapping
Group Type	Block Type
Object Type	Block Type
Interface Type	Process Type
Object Reference	Pid[2]
Oneway method (asynchronous)	Signal prefixed with pCALL_
Method (synchronous)	Exported procedure, where the returned value is a choice type, i.e., may assume different type of values, one for the result and another one in the case of an exception occurrence
Enum	Newtype with the corresponding literals
Typedef	Syntype
Struct	Newtype with the corresponding structure
Constant	Synonym

Table 1. ODL to SDL mapping

One of the major advantages of this mapping over the one presented at [6] is the fact that it allows the existence of exceptions in the SDL model, that play an important role in distributed systems (which is the case of the TINA Service Architecture). Moreover, the resulting specification is closer to reality when compared to the one presented at [7], since it uses exported procedures to represent object methods.
So, by using this mapping, the as-UAP, PA, IA, named and anonUA objects are modelled as Block Types (Figure 2).
Additionally, three other objects were created in the SDL model. NameServer, IdDispatcher and Creator are not part of the TINA specifications, however they give support for the implementation of the access session.

[2] *Pid: Process Identification*, represents a reference to the process localization (process identifier)

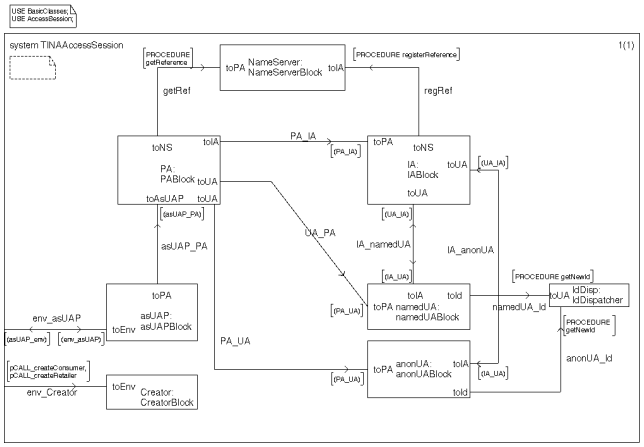


Figure 2. TINA Access Session SDL modelling

The NameServer block has one only functionality of providing the user with a reference to an IA object once the desired provider name is given, so that the user may contact the provider. Since the NameServer must have references to all IA objects, whenever a new IA is created, it may register itself in the NameServer through the *registerReference* method.

The IdDispacher block is responsible for generating sequential numbers (through *getNewId* method) that will be used by as access sessions identifiers.

Finally, the Creator object has a single purpose: make it easier to instantiate new consumers and providers during the system simulation (what is done by sending two signals – *createConsumer* and *createRetailer*).

Every object that composes the TINA service architecture were organized in a similar way. They were mapped into SDL blocks and their interfaces were mapped into SDL processes. However, as an ODL object may be dinamically created but an SDL block can not be instanciated, for each SDL block there may be a pre-existent process (besides the ones that represents its interfaces) that acts as the creator of the block (Figure 3), allowing the block to be "instantiated". This process instantiates all processes of that block and, after their creation, sends them references to all the other processes that composes the block, so that they may communicate to each other (once they know each other *Pid*s) and then act as a single object (an unit).

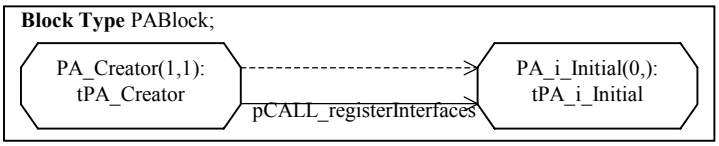


Figure 3. Solution for lack of block instatiation

Figure 4 shows the SDL specification of PA block, with the presence of the creator process. This process contains just one procedure, which is responsible for the instantiation of the object. This procedure instantiates the interfaces of the object and

sends them the references to the others (through the *pCALL_registerInterfaces* signal, an asynchronous operation). Moreover, the creator process may also provide the interfaces with other relevant informations by the time of their creation.

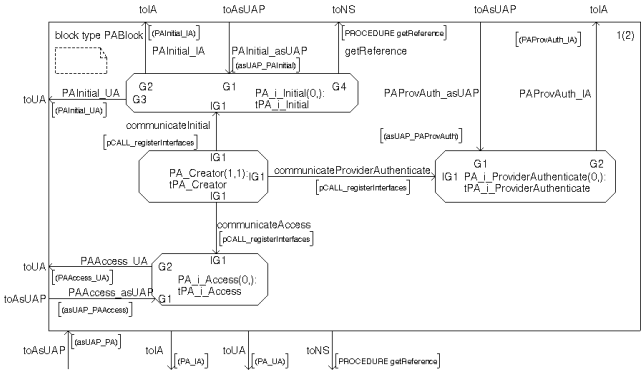


Figure 4. PA Block

Figure 5 shows the *PA_i_Initial* process that, as all the other processes, has its set of exported procedures (representing the methods) that may be accessed by any other blocks. In the other hand, all interfaces must have their set of imported procedures, which are imported from other interfaces (from the same or from another object), that represents the method that they are allowed to access.

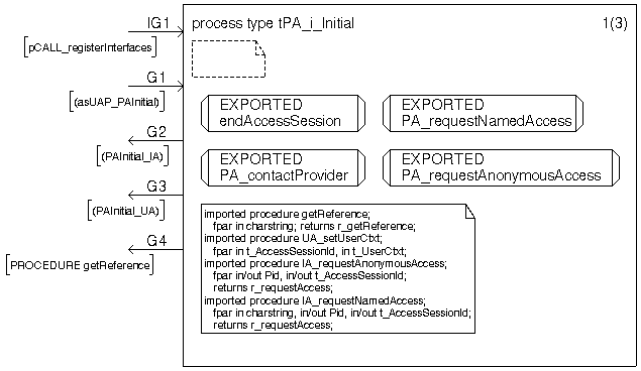


Figure 5. *i_Initial* interface from PA object

Figure 6 presents the *PA_contactProvider* exported procedure. This procedure defines the behavioral specification of the method it represents. Since ODL is limited to the specification of object interfaces, not dealing with the object dynamics, this behavioral specification can not be found on TINA ODL specifications.

This exported procedure is responsible for contacting the provider whose name is provided by the *desiredProviderParam* parameter. So, the *getReference* remote procedure from the NameServer block is called (this procedure was previously imported by the process, as shown in Figure 5). So, if no exception is thrown, the result of the procedure gives the provider's IA address.

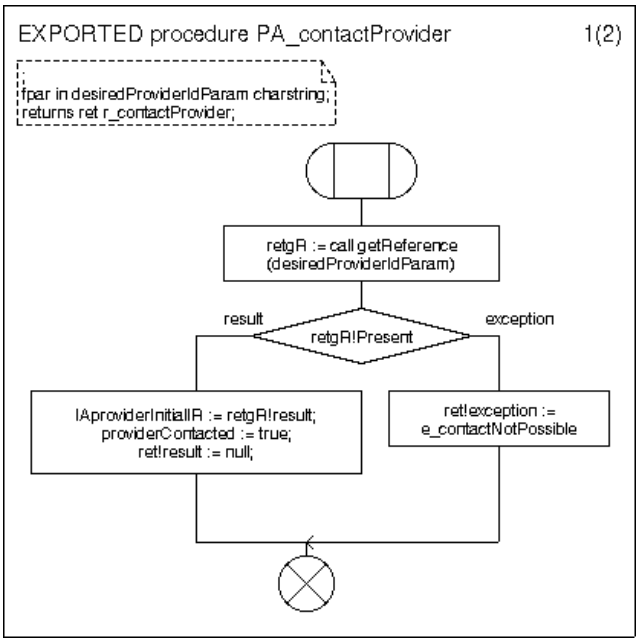


Figure 6. *PA_contactProvider* exported procedure

A fact that deserves some attention is the high similarity between the anonUA and namedUA blocks. They both represent (anonymous or identified) users in the provider's domain. This similarity between them allowed the creation of a block called UABlock, which is an abstract block (i.e. never instantiated) that groups the common characteristics of anonUA and namedUA, causing them both to be defined as specializations of UABlock. They inherit all UABlock characteristics and are free to add their own particularities by adding new methods or redefining existing ones.

Among the common procedures shared by namedUA and anonUA block (defined in UABlock), there are the procedures responsible for listing of the avaiable services for a particular user and for finalizing an access session. Among the procedures that are specific of each block, there is the one responsible for establishing a new access session – since that the namedUA block must authenticate the user, while the anonUA block must not, as it deals only with anonymous access sessions.

Finally, an important characteristic of the proposed ODL to SDL mapping that distinguishes itself from other proposals ([6],[7]) is that is allows the mapping of ODL methods into exported procedures with the possibility of throwing/catching exceptions. In order to achieve this, it is defined a data type for the method result that is created using the Choice generator. This allows variables of this data type to assume different data type values depending on the context. Using this, the method result variable may carry the result value (if the method execution had no problems) or a value that represents an exception (if an exception was thrown). In Figure 7 we may see an example with the definition of the result data type for the *PA_contactProvider* method (or exported procedure). The result may assume either a null value (the real

method result type, represented by the *result* component) or a value of *e_contactProvider* type (an enumeration type where each literal represents an exception that may be thrown, represented by the *exception* component).

```
NewType r_contactProvider choice
   result null;
   exception e_ContactProvider;
EndNewType r_contactProvider;
```

Figure 7. Definition of the *PA_contactProvider* method return value type

The object that invoked this method, may easily verify the exception ocurrence by checking which return value component is defined (Figure 8) and then following the normal program flow or executing the branch that deals with the exception ocurrence (the exception may be identified by evaluating the *exception* component).
All other blocks (as-UAP, IA, anonUA, namedUA) were organized in the same way.

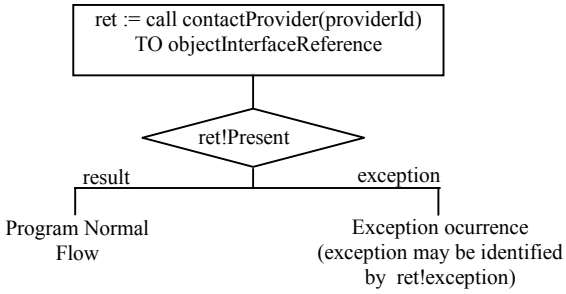


Figure 8. Detection of an exception ocurrence

4 TINA Access Session Simulation and Validation

After specifying the whole Access Session system, the desired functionalities were completely tested. The system was fully validated, by the use of the Validator tool from Telelogic's SDT suite for the SDL development. In this process it was possible to detect some specification errors as, for example, a wrong compasion that was done on *UA_endAccessSession* process from UA block. That error was easily found after the execution of Validator, by noting that no symbol below one of the comparison branches was ever reached (Figure 9). After the correction of this comparison, the system was re-validated and all symbols were now reached.

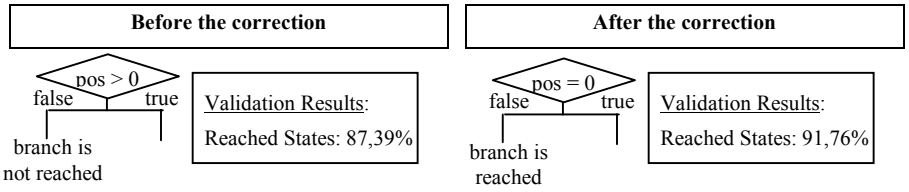


Figure 9. Error detection through validation

Not every symbol (around 8%) was reached during the validation process. Some of them deal with error situations that could happen in the real system, but does not happen in an ideal situation. An example is the case where the PA communicates to the wrong namedUA. Given the PA specification, it is impossible to happen, however, namedUA treats this situation. Moreover, some UABlock symbols (defined as VIRTUAL) were redefined in anonUA and namedUA, so they are never reached.
After the complete validation of the system, its main functionalities were simulated. The Simulator tool, which is also part of SDT, was used for this process. This tool allows the total visualization of the simulation through the generation of Message Sequence Charts (MSC) that shows the interactions among all the system components. Figure 10 shows one of these generated diagrams.

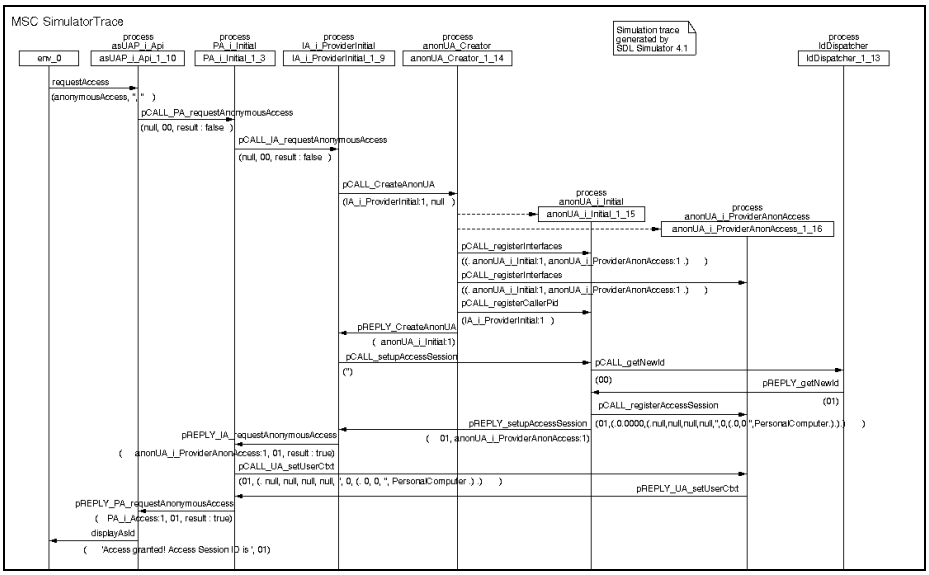


Figure 10. Anonymous access session establishment

After the initial contact to the provider, the user already owns a reference to the IA of the desired provider. Now, the user is able to perform lots of action, as for example asking the provider for an access session.
This session may be anonymous or identified. If it is an anonymous session the user asks for the session through some graphic interface – here represented by the *requestAccess* signal from *i_Api* interface. This requisition is sent to PA through *PA_requestAnonymousAccess* method from the *i_Initial* interface. Then, PA sends the signal to IA through *IA_requestAnonymousAccess* method from *I_ProviderInitial* interface.
The *IA_requestAnonymousAccess* method then asks the *anonUA_Creator* method the instantiation of an anonUA object, that will represent the user in the provider's domain. The IA object still asks the new anonUA the establishment of a new access session through the call of the *UA_setupAccessSession* method from the *i_Initial*

interface. This method asks IdDispatcher an identifier for the new session that will be created through the *getNewId* method.
Once the new session is acepted, PA passes to anonUA some user context informations (terminal capacities, some important interfaces, etc) through the *UA_setUserCtxt* method from the *i_ProviderAnonAccess* interface. After all this, as-UAP receives a confirmation of the establishment of the access session (through the result of the *PA_requestAnonymousAccess* method) with the session identifier, *asId*.
Finally, as-UAP may display a success message to the user confirming the access session establishment. The message is represented here by the *displayAsId* signal.

5 Code Generation

Once the SDL model is created and fully tested (simulated and validated), it is time to produce code to execute the system. In our case, the system is intented to be executed in a distributed environment, such as CORBA.
In order to facilitate the production of code based on the SDL specifications and also to reduce the errors that might appear in the process of converting the SDL specifications into a fully functional program, this process was made automatic by the development of a code generation tool with the help of JavaCC [8]. This tool is responsible for the generation of Java code, using JavaIDL ORB (Object Request Broker) as the target DPE, based on the SDL specifications.
Java was chosen for several reasons, but the most important one is the platform independence. This characteristic bring much freedom for the generated code, since it is only necessary to target the application to one platform (the Java Virtual Machine) and it will run on any operating system. Moreover, CORBA was chosen as the target DPE since it is becoming a standard distributed environment and, more specifically, JavaIDL ORB was chosen since it's free and comes with the recent versions of JDK.
The Java code generation process must follow some rules that may be expressed as a set of mappings from SDL to Java (Table 2). This mappings are based on the ODL to SDL mapping presented at Table 1.

SDL Structures	Java Mapping
Block Type	Package
Process Type	Interface
Signal	Oneway public method
Exported Procedure	Public Method
Procedure (non-exported)	Private Method
Pid	Org.omg.CORBA.Object
Newtype	Class representing corresponding data type
Timer	Class with timer functionalities

Table 2. SDL to Java mapping

All data types defined outside the blocks (SDL system or SDL package scope) are placed in a special package named *DataTypes*. The block types are mapped into a java packages containing all data types defined inside the block. The process types are mapped into interfaces placed inside its corresponding package. Finally procedures

are mapped into methods that may be public (in case of an exported procedure) or private.
All SDL data types are mapped into their equivalent Java data types. The timer variables are mapped into timer objects that presents the same functionaly of the SDL timer sort, i.e. it may be set or reset, its state may be checked (activated or deactivated) and after the time it was set to is passed, it calls the desired method (that is the equivalent of sending the desired SDL signal) as a thread (asynchronous call).
The Newtype constructor always generate a class that presents a representation of the desired data type. The classes generated following the OMG (Object Managament Group) recommendation [9] for the use of Java with CORBA.
The only exception is the case of Newtype defined with the Choice generator. When it represents some function result, it does not generate any class, however it is used to indicate what is the method result type (*result* component) and the exceptions that may be throwed by the method (*exception* component). Figure 11 shows how an exception catching is mapped into Java code (evaluating the *ret* variable, that is of Choice type).

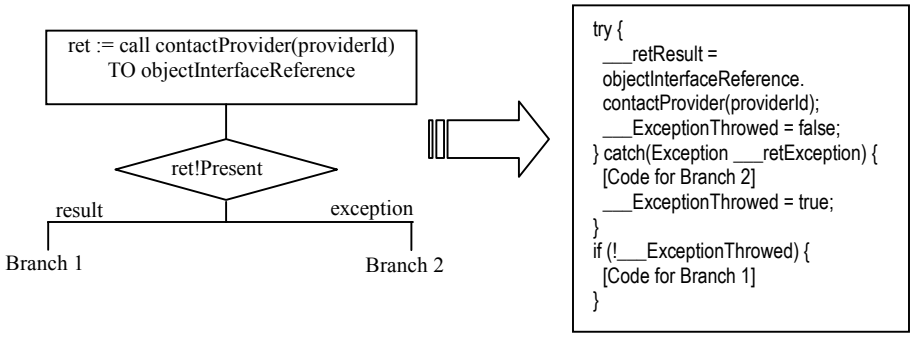
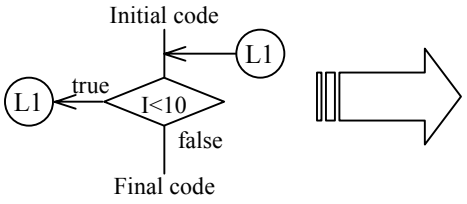


Figure 11. Java code generation of exception catching

The resulting Java code is fully functional and is composed by the TINA objects and a new IDL for each defined object (since some minor changes may occur during the process of mapping from the original IDL/ODL to SDL and then to Java). The Java code generated may be freely modified so that some enhancements may be added to the generated application, such as graphic interface, database access, convertion to applet so that it may be access over the Internet by any browser, etc. However, this changes must be made very carefully, so that it does not insert errors in previously tested functions (what would compromise the whole simulation/validation process).
One major problem encountered during the Java code generation is the lack of *GOTO* in the Java language. Since SDL makes an extensive use of the *JOIN*, which is a kind of *GOTO*, the generated code should somehow simulate the *JOIN* functionality. One possible solution is to put the labeled statements into private methods and then map a *JOIN* statement into a method call. This solution, presented at [10], solves the problem, although it leads to an ineficcient code, with plenty of recursion.
However, in order to avoid the inefficiency of the resulting code, the Java code generator developed uses another technique for simulating the *JOIN* behavior. This solution consists on mapping the existing labels into options of a big *switch* statement. We define constants for each existing label (*___LABEL_<labelname>*), along with

two more constants, one that represents no label (___*NOLABEL*) and one representing from where the method execution should start (___INITIALLABEL). This big *switch* statement is then placed inside a *while* loop, that keeps looping until the code has no label to go to. So, a ___*LABEL* variable is defined in order to control the flow of the program. An example is shown at Figure 12.




```
___LABEL = ___INITIALLABEL;
while(___LABEL != ___NOLABEL) {
 switch(___LABEL) {
  case ___INITIALLABEL:
   [Initial code block]
  case ___LABEL_L1;
   if ((I<10) == true) {
    ___LABEL = ___LABEL_L1;
    break;
   } else if ((I<10) == false) {
    [Final code block]
    ___LABEL = ___NOLABEL;
    break;
   }
 }
}
```

Figure 12. Example of Java code generation for a JOIN statement

In this example, only the *JOIN* part of the method is shown. The Java code generated is shown in the box on the right. As explained before, initially the ___*LABEL* variable is set to ___*INITIALLABEL*. Once the execution starts, the programs enters into the while, ___*LABEL* is evaluated by the switch clause and the statements below "*case* ___*INITIALSTATEMENT*" are executed (the Initial Block Code). Then it keeps executing, and evaluates the *if* statement (that represents the SDL decision) since no *break* is found.

Supposing that I is lower than 10, ___*LABEL* will be assigned to ___*LABEL_L1*, then a break is found. The execution gets back to the while, that is still true and the ___*LABEL* is once again evaluated by the *switch* clause. But this time, the execution goes directly by the statements below "*case* ___*LABEL_L1*", i.e. the *if* statement.

Now, supposing that I was not lower than 10, the Final Code Block would be executed, ___*LABEL* would be assigned to ___*NOLABEL* and, as a *break* would be found, the execution would go back to the while that would finish the method execution (once ___*LABEL* == ___*NOLABEL*).

As this example shows, the generated code has exactly the same functionality of the *JOIN* statement, as it does this without leading us to a very inefficient code.

6 Conclusion

The work presented in this paper proposes a methodology for the development of TINA systems. The first step is the SDL formal specification of the system, where the Access Session was used as an example. In order to achieve this formal specification, it is introduced a more suitable ODL to SDL mapping, since it generates a

specification that is closer to realiy, through the mapping of object methods to exported procedures and the exceptions handling.
By using this mapping, it was built a SDL object oriented model of the TINA Access Session, providing a behavioral formality complementing the formality achieved by ODL (that is limited to the specification of object interfaces). The SDL specification allows the observation of the system's dinamics, providing the simulation of the system's functionalities e its complete validation.
In order to achieve this, the SDT suite was used. The Simulator tool provided the simulation of most important use cases while the Validator provided the validation of the system. By using these tools, the correctness of the SDL model was guaranted.
In a final step, it was built a code generator that uses some SDL to Java mapping rules – based on the ODL to SDL mapping – to generate Java code that uses JavaIDL ORB as the target DPE. The generated code may be freely modified so that some enhancements can be made (such as the creation of a graphic interfaces, database access, etc), although these modifications should be done much carefully.
With this methodology, the creation of TINA systems is made much easier, since the logic of the system can be fully verified before its implementation. Moreover the system may be developed in a much more abstract manner, by using SDL, then if it was coded directly in an implementation programming language such as Java or C++.

References

[1] Chapman, M., Montesi, S. *Overall Concepts and Principles of TINA*. TINA-C, February 1995.
[2] Abarca, C. et al. *Service Architecture - Version 5.0*. TINA-C, June 1997.
[3] Parhar, A. *TINA Object Definition Language Manual - Version 2.3*. TINA-C, July 1996.
[4] ITU-T (CCITT Recommendation Z.100). *Specification and Description Language*. Geneve, Switzerland, 1993.
[5] Telelogic AB. *Telelogic Tau 4.1 SDL Suite Getting Started*. Sweden, 2000.
[6] Björkander, M. *Mapping IDL to SDL*. Telelogic AB, 1997.
[7] Kolberg, M., Sinnott, R., Magill, E. *Experiences modelling and using formal object-oriented telecommunication service frameworks*. Computer Networks: The International Journal of Computer and Telecommunications Networking, vol. 31, pp. 2577-2592, December 1999.
[8] Metamata Inc. *Java Compiler Compiler (JavaCC) – The Java Parser Generator*. Metama Inc., 2000.
[9] Leland, M. *OMG IDL to Java Language Mapping (2.4)*. OMG, November 2000.
[10] Sherratt, E., Loftus, C. *Designing distributed services with SDL*. IEEE Concurrency, vol. 08, n. 01, pp. 59-66, January-March 2000.