Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação

Modelo de Verificação de Processos de Negócios através de uma Máquina Virtual Pi-Calculus

Autor: Marcos Vanine Portilho de Nader
Orientador: Prof. Dr. Maurício Ferreira Magalhães

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: **Engenharia de Computação**.

Banca Examinadora

Prof. Dr. Maurício Ferreira Magalhães ..... DCA/FEEC/Unicamp
Prof. Dra. Alice Maria Bastos H. Tokarnia ..... DCA/FEEC/Unicamp
Prof. Dr. Mario Jino ..... DCA/FEEC/Unicamp
Prof. Dra. Cecília Mary Fischer Rubira ..... DSI/IC/Unicamp

Campinas, SP
Dezembro/2006

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

Nader, Marcos Vanine Portilho de
N125m Modelo de verificação de processos de negócios através de uma máquina virtual Pi-Calculus / Marcos Vanine Portilho de Nader. – Campinas, SP: [s.n.], 2006.

Orientador: Maurício Ferreira Magalhães.
Dissertação (Mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Gestão de negócios. 2. Serviços na Web. 3. Álgebra (Computadores). 4. Fluxo de trabalho. 5. Sistema de computação virtual. I. Magalhães, Maurício Ferreira. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título

Resumo

Duas áreas importantes estão em desenvolvimento: Gerência de Processos de Negócios (*Business Process Management*) e Orquestração de *Web Services* (*Web Services Orchestration*). Ambas têm um objetivo que é integrar aplicações ou outros processos que tenham interfaces *web services*, usando o paradigma de processos de negócios. Uma linguagem que vem sendo difundida para essas aplicações é a BPEL (*Business Process Execution Language*). Este trabalho apresenta um *framework* aplicável à análise e verificação de processos de negócios escritos em BPEL através do uso de Pi-Calculus. Pi-Calculus é uma álgebra de processos que possui mecanismos formais para criação e ativação de processos que se comunicam através da troca de mensagens em canais, usando o modelo de *rendez-vous* síncrono. Nesse *framework*, o processo BPEL é traduzido para um programa Pi-Calculus. Uma Máquina Virtual Pi-Calculus (MVP) recebe o programa Pi-Calculus e produz todas as reações possíveis, ou seja, gera todos os caminhos de execução que o programa pode seguir. A partir desse resultado, efetua-se a verificação de propriedades como: atendimento às especificações de mais alto nível, ordenação de eventos e ocorrência ou não de deadlocks. Em termos práticos, uma ferramenta desse tipo pode ser incorporada aos Sistemas de Gerência de Processos de Negócios (*Business Process Management Systems* - BPMS) para ampliar a cobertura de testes durante as fases de análise e implementação de um processo dentro do seu ciclo de vida. Nesses tipos de sistemas, a reparação de um erro durante a fase de execução é muito mais custosa que nos sistemas tradicionais.

Palavras-chave: Gerência de Processos de Negócios, Orquestração de *Web Services*, Sistemas de Gerência de *Workflow*, Sistema de Gerência de Processos de Negócio, Arquitetura Orientada a Serviços, Álgebra de Processos, BPEL, Pi-Calculus.

Abstract

Two important areas have been in development lately: Business Process Management and Web Service Orchestration. In both of them, the objective is to integrate applications with web services interface through business process paradigm. A number of languages have been proposed with consensus being formed around BPEL (Business Process Execution Language). This dissertation presents a framework for BPEL processes analysis and verification through Pi-Calculus. Pi-Calculus is a process algebra with formal mechanisms for processes creation and activation; these processes communicate sending and receiving messages through channels using the synchronous rendezvous model. In this framework, the BPEL process is translated to a Pi-Calculus program, A Pi-Calculus Virtual Machine (MVP) receives a Pi-calculus program and executes all possible reactions, that is, it generates all execution paths possible to be taken. With this result, the properties such as high-level specification accomplishment, events ordering and deadlock freedom are verified. In practical terms, a tool of this sort can be part of a Business Process Management System (BPMS) to broaden test coverage during the analysis and implementation phases within a process life cycle. In these kinds of systems, a repairing mistake during the execution phase is more complex than in traditional systems.

Keywords: Business Process Management, Web Services Orchestration, Workflow Management Systems, Business Process Management Systems, Service Oriented Architecture, Process Algebra, BPEL, Pi-Calculus.

À minha esposa e aos meus filhos.

Agradecimentos

Ao meu orientador, Prof. Maurício Magalhães, pela oportunidade do aprendizado e pela orientação no curso e nesta tese.

À minha esposa Cláudia Márcia, aos meus filhos Daniela e Vinícius, pelo incentivo e companheirismo durante todo o tempo que precisei para desevolver os trabalhos neste mestrado.

Aos meus pais Marina e Fued, que me proporcionaram a educação de vida, permitindo a realização de vários desafios, inclusive a conclusão deste curso.

Aos colegas Vera Bier e Isidro Lopes que me incentivaram a iniciar este curso de mestrado, depois de longo tempo afastado da Unicamp.

Ao meu irmão Pedro e à professora Flávia Tafner que me ajudaram nas revisões dos textos relacionados ao curso.

Ao meu irmão Tales e meu amigo Marcelo Elias, pessoas que tomo como exemplo.

Às empresas Cleartech e CPqD que me apoiaram nesta empreitada.

A todas as pessoas, que porventura eu tenha esquecido de mencionar, mas que também contribuíram para o desenvolvimento desta tese.

Sumário

Lista de Figuras **xiii**

Lista de Tabelas **xv**

Glossário **xvii**

Trabalhos Publicados Pelo Autor **xix**

1 Introdução **1**
1.1 O tema e o objetivo principais 1
1.2 Visão geral do *Framework* 2
1.3 Trabalhos relacionados 3
1.4 Estrutura da dissertação 6

2 Conhecimentos Preliminares **11**
2.1 Gerência de Processos de Negócio (BPM) 11
2.2 Arquitetura Orientada a Serviços (SOA) 15
2.3 *Web Services* 18
2.4 Modelagem de Comportamento de Software 22
2.4.1 Rede de Petri 22
2.4.2 Álgebras de Processos 24
2.4.3 Linguagem Z 25
2.5 Conclusão 26

3 BPEL e Pi-Calculus **27**
3.1 BPEL (*Business Process Execution Language*) 27
3.1.1 Elementos básicos de um processo 28
3.1.2 Atividades de um processo 29
3.1.3 Atributos-padrões e elementos-padrões 31
3.1.4 Exemplo: Agência de Viagem 33
3.1.5 Arquitetura de uma implementação BPEL 36
3.2 Pi-Calculus 38
3.2.1 Elementos básicos 39
3.2.2 Variáveis lógicas 40
3.2.3 Expressões lógicas 42

3.3 Conclusão 43

4 Mapeamento BPEL para Pi-Calculus 45
4.1 Considerações sobre canais Pi-Calculus e sobre o mapeamento 45
4.2 Regras de mapeamento para as atividades 46
4.3 Atributos-padrões e elementos-padrões 51
4.4 Atividades estruturadas e Eliminação de Caminho Morto (*Dead Path Elimination*) . . 53
4.5 Exemplo: Agência de Viagem 54
4.6 Conclusão 57

5 Máquina Virtual Pi-Calculus 59
5.1 A Linguagem da Máquina Virtual Pi-Calculus 60
5.2 Arquitetura da Máquina Virtual 61
5.3 Implementação da MVP 64
5.3.1 Módulo Processador 64
5.3.2 Módulo Gerador de Seqüências 65
5.3.3 Módulo Analisador 65
5.3.4 Módulo Básico 66
5.3.5 Módulo Shell 66
5.4 Análise e verificação de programas Pi-Calculus 69
5.4.1 Árvore de Execução 69
5.4.2 Canais de *status* 71
5.4.3 Tabela de Execução 72
5.4.4 O Comando SELECT 74
5.4.5 Aplicação do comando SELECT 76
5.5 Um exemplo avançado 77
5.6 Exemplo em Processos de Telecomunicações: Gerência de SLA 81
5.7 Conclusão 84

6 Conclusão e Trabalhos Futuros 85

Referências Bibliográficas 87

A Esquema da Linguagem da Máquina Virtual Pi-Calculus (MVP-L) 91

B Programa da Agência de Viagem na Linguagem da MVP 95

C Árvore de Execução Gerada pelo Processo da Agência de Viagem 99

D Expressão Pi-Calculus do Processo da Compra de Livro 103

E Verificação de Expressões Lógicas através da Máquina Virtual Pi-Calculus 107

F Desenvolvimento e Análise do Processo de Gerência de SLA **111**
F.1 Visão Geral . . . 111
F.2 Descrição do processo de Gerência de SLA . . . 112
F.3 Especificações . . . 113
F.3.1 Considerações iniciais . . . 113
F.3.2 Cenários de interações entre os processos . . . 114
F.4 Casos de uso para o processo fim-a-fim . . . 118
F.4.1 Fluxo normal . . . 119
F.4.2 Anormalidade de um recurso ou de um serviço externo . . . 120
F.4.3 Reclamação do cliente sobre violação de SLA . . . 121
F.5 Construção em BPEL . . . 122
F.5.1 Serviços dos processos . . . 122
F.5.2 Descrição simplificada em BPEL . . . 124
F.6 Mapeamento do processo BPEL para o programa Pi-Calculus . . . 127
F.7 Execução do programa Pi-Calculus . . . 127
F.8 Análise e verificação do processo . . . 128
F.9 Conclusão . . . 134

G Expressão Pi-Calculus do Processo de Gerência de SLA **135**

Lista de Figuras

1.1 Passos do processo de verificação usando MVP2. 3

2.1 Ciclo de vida de um processo. 13
2.2 Relação BPM-SOA. 18
2.3 Arquitetura Web Service. 20

3.1 Elementos da estrutura de um processo BPEL. 28

5.1 Elementos de um programa na Linguagem da Máquina Virtual Pi-Calculus. 60
5.2 Arquitetura da Máquina Virtual Pi-Calculus (MVP). 62
5.3 Janela da Interface Gráfica para a análise de programas. 68
5.4 Árvore de Execução do processo da Agência de Viagem. 71
5.5 Tabela de Execução do processo da Agência de Viagem. 75
5.6 Processo da Livraria. 77
5.7 Tabela de Execução do processo da Livraria 79
5.8 Lista de *deadlocks* do processo da Livraria 82
5.9 Árvore de Execução do Processo da Livraria em *deadlock* 83

E.1 Tabela de Execução do programa de verificação da função *And* de 2 variáveis 108

F.1 Processos do Grupo de Operações do modelo de eTOM - fonte: CPqD. 112
F.2 Macro-Processos usados nas especificações do processo de Gerência de SLA. 114
F.3 Processo de Gerência de SLA - fluxo normal. 115
F.4 Processo de Gerência de SLA - fluxo com alerta e sem violação. 116
F.5 Processo de Gerência de SLA - fluxo com violação de SLA. 118
F.6 Tabela de Execução do programa Pi-Calculus resultante do processo de Gerência de SLA. 128
F.7 Tabela de Execução *semCRM*. 129
F.8 Tabela de Execução *comCRM*. 130
F.9 Resultado do comando para verificar *deadlock*. 133
F.10 Árvore de Execução do programa de Gerência de SLA em *deadlock*. 133

Lista de Tabelas

1.1 Quadro comparativo entre as formalizações e análises de trabalhos relacionados. . . 6

5.1 Comandos da Console da MVP. 67

F.1 Nomes dos sistemas que correspondem aos macro-processos. 122
F.2 Serviços exportados dos sistemas que correspondem aos macro-processos. 123
F.3 Serviço exportado do sistema auxiliar. 123

Glossário

ACID	Atomicity, Consistency, Isolation, Durability
ACP	Algebra of Communicating Process
AP	Álgebra de Processos
API	Application Program Interface
BAM	Business Activity Monitoring
BPEL	Business Process Execution Language
BPM	Business Process Management
BPMI	Business Process Management Institute
BPML	Business Process Modeling Language
BPMN	Business Process Modeling Notation
BPMS	Business Process Management System
BPWS4J	Business Process Execution Language Web Service JavaTM Run Time
BSS	Business Support System
CCS	Calculus of Communicating Services
CDL	Choreography Description Language
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CSP	Communication Sequencial Processes
CWB	Concurrent Workbench
EAI	Enterprise Application Integration
ESB	Entreprise Service Bus

eTOM	enhanced Telecommunication Operation Map
F/P	Fornecedor/Parceiro
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
ISO	International Standard Organization
ITU-T	International Telecommunication Union Telecommunication Sector
J2EE	Java 2 Enterprise Edition
MVP	Máquina Virtual Pi-Calculus
MVP2	Modelo de Verificação de Processos usando uma Máquina Virtual Pi-Calculus
OASIS	Organization for the Advancement of Structured Information Standards
OSS	Operation Support System
PMG	Process Modeling Group
QoS	Quality of Service
RAD	Role Activity Diagram
RPC	Remote Procedure Call
RVP	Rede Virtual Pi-Calculus
SLA	Service Level Agreement
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TMForum	TeleManagement Forum
UDDI	Universal Description Discovery and Integration
UML	Unified Modeling Language
URI	Universal Resource Identifier
WSCI	Web Service Choreography Interface
WSDL	Web Service Description Language
WSFL	Web Service Flow Language
XML	Extended Markup Language

Trabalhos Publicados Pelo Autor

1. Nader M.V. “Gerência de Redes e Serviços em uma NGN - Perspectiva de Integração”. *Sociedade dos Usuários de Informática e Telecomunicações* (SUCESU´2003), Salvador, Bahia, Brasil, abril de 2003.

2. Nader M.V. e Magalhães M., “Proposta de uma Máquina Virtual Pi-Calculus para Verificação de Composição de Web Services”. *Workshop de Desenvolvimento Baseado em Componentes* (WDBC), Recife, Pernambuco, Brasil, dezembro de 2006.

Capítulo 1

Introdução

1.1 O tema e o objetivo principais

Gerência de Processos de Negócio é um campo de aplicação bastante amplo. Sistemas têm sido desenvolvidos com ênfase em modelagem de processos e em execução baseada em *workflow* (teoria e padrões) e integração de sistemas. Por exemplo, na Gerência de Telecomunicações, as principais organizações de estudo da área - TeleManagement Forum (www.tmforum.org), *International Telecommunication Union - Telecommunication Sector* (www.itu.int/), entre outras - sugerem que, no topo das camadas de gerência, tenha-se a Gerência de Processos de Negócios integrada aos sistemas OSS/BSS (*Operation Support System/Business Support System*) (18) e (16).

As empresas têm demonstrado nestes últimos anos um grande interesse na utilização da tecnologia dos *web services* aplicada à Gerência de Processos de Negócios. Neste sentido, esforços têm sido desenvolvidos para padronização de uma linguagem de especificação de processos, bem como a forma como esses processos interagem com outros serviços para execução do processo fim-a-fim. Este trabalho optou pela especificação BPEL (*Business Process Execution Language*) [1] definida pelo grupo OASIS, BEA, IBM e Microsoft, por ser a linguagem mais usada pelas organizações quando se trata da especificação de *web services* (27).

BPEL (2) é baseada em XML e usada para a construção de *web services* na forma de processos construídos a partir da composição de outros *web services*. Um processo em BPEL é um conjunto de atividades que suporta fluxo de controle, concorrência, interação com *web services*, tratamento de exceções e compensações. A comunicação com os *web services* é feita através de WSDL (*Web Service Definition Language*) .

BPEL é definida através da utilização de especificações XML (*Extended Markup Language*) (4), tais como: WSDL 1.1 (*Web Service Definition Language*) (6), *XML Schema* 1.0 (12) e *XPath* 1.0 (7).

[1] Utiliza-se também BPEL4WS (*Business Process for Web Services*) para referenciar a mesma linguagem.

WSDL e *XML Schema* provêem o modelo de dados usado pelos processos BPEL. *XPath* é utilizado como suporte para manipulação de dados. Todos os recursos externos, incluindo parceiros e fornecedores, são representados como serviços WSDL. BPEL também provê mecanismos que permitem o desenvolvimento de extensões para acomodar versões futuras (2).

As semânticas de criação de atividades concorrentes dos processos de negócio, comunicação e sincronismo entre processos e atividades do BPEL podem ser expressas em formalismos derivados das álgebras de processos (31).

Uma das possíveis álgebras citadas no parágrafo anterior é o modelo de computação para sistemas concorrentes denominado Pi-Calculus. A sintaxe de Pi-Calculus permite representar processos, composição de processos paralelos, comunicação síncrona entre processos através de canais, criação dinâmica de novos canais, replicação de processos e não-determinismo. Um processo é uma abstração de um fluxo de controle independente. Um canal é uma abstração de um enlace de comunicação entre dois processos cuja interação se dá pelo envio e recepção de mensagens através de canais (23).

O objetivo deste trabalho consiste em investigar o uso de Pi-Calculus para suportar a análise e a verificação de processos BPEL na fase de projeto (15). Neste sentido, definiu-se um *framework*, denominado Modelo de Verificação de Processos através da Máquina Virtual Pi-Calculus (MVP2), que contém:

- definição de regras para a conversão de programas BPEL para programas Pi-Calculus; a partir dessas regras, é possível desenvolver um compilador BPEL para Pi-Calculus (essa atividade não foi alvo deste trabalho);
- linguagem da MVP descrita em um esquema XML;
- Máquina Virtual Pi-Calculus (MVP), usada para a análise e verificação do programa Pi-Calculus traduzido a partir do programa BPEL.

1.2 Visão geral do *Framework*

A figura 1.1 ilustra os passos envolvidos na utilização do *framework*.

No primeiro passo, o processo BPEL é traduzido para Pi-Calculus. Essa tradução segue as regras definidas no capítulo 4. Valores específicos de variáveis não são considerados, o que permite que a análise seja feita com foco na concorrência e no fluxo de controle do processo. Temporizações são consideradas como eventos independentes de valores. Escopo, tratamento de falhas, eventos e compensações não são considerados para fins de simplificação.

No segundo passo, a MVP executa o programa Pi-Calculus e gera uma estrutura em árvore denominada *Árvore de Execução*, que contém todas as reações possíveis do programa Pi-Calculus. Essa

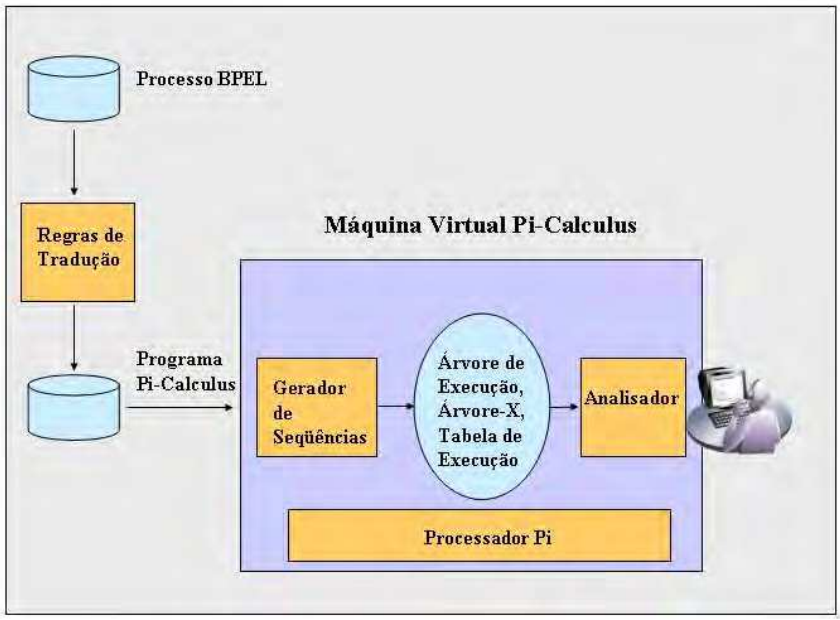

Fig. 1.1: Passos do processo de verificação usando MVP2.

estrutura passa por um filtro que seleciona as operações correspondentes às interações com os *web services* e às atividades relevantes do processo BPEL gerando uma *Tabela de Execução*. Essa tabela é a entrada para o terceiro passo, no qual o usuário utiliza o *Analisador* para a análise e verificação das propriedades de um processo. Foi definido nesse Analisador um comando que permite testar se os requisitos dos processos são atendidos pelo processo BPEL, como também as propriedades relacionadas a *liveness*, *safety* e *deadlock*.

1.3 Trabalhos relacionados

O trabalho desenvolvido por Koshkina (19) e Koshkina e Bruegel (20) propõe a tradução de processos BPEL para especificações de uma álgebra denominada BPE, desenvolvida especificamente para esse propósito. A ferramenta utilizada para a análise e verificação é a *Concurrency Workbench* (CWB), que permite a configuração dessa álgebra (8). CWB dispõe de três métodos de verificação: teste de equivalência para verificar se a implementação de um sistema atende às especificações; teste de pré-ordem que verifica requisitos que são especificados como a ordenação de eventos descrita em um nível mais alto que a implementação e teste de modelo que determina se um sistema satisfaz alguma propriedade formulada em lógica, como por exemplo, detecção de *deadlock*.

Sob o tema de verificação de composição de *web services*, Foster et. al. (15) e Foster (14)

propõem a composição dos *web services* em duas representações. A primeira é especificada pelo projetista através do uso de Diagramas de Seqüência de Mensagens (*Message State Charts*) do UML (*Unified Modeling Language*) e a outra usa BPEL para a implementação. As duas representações são traduzidas para a forma de Processos de Estado Finito (*Finite State Processes*)[2]. O processo de verificação consiste em comparar e observar os estados dos dois sistemas.

Salaün et. al. (31) propõe a especificação de orquestração e coreografia de *web services* através de álgebra de processos - a comunidade de desenvolvimento de *web services* pode se beneficiar das linguagens e ferramentas existentes nessa área. A tradução entre linguagens é feita nos dois sentidos. Por um lado, na fase de projeto, a álgebra de processos é usada para descrever uma especificação abstrata do processo produzindo um modelo preliminar, que pode ser validado pelas ferramentas de análise. Esse modelo é então usado como uma referência para implementação. A partir do programa escrito na álgebra, gera-se o esqueleto do código na linguagem executável. Por outro lado, por razões de engenharia reversa, a compilação no sentido reverso é utilizada para extrair a descrição algébrica que permite validar propriedades dos serviços existentes. As validações propostas são: testar se duas especificações de processos, uma mais abstrata e outra mais próxima da implementação, são equivalentes e verificar propriedades de lógica temporal como *safety* e *liveness*.

O *framework* apresentado por Ferrara (13) contém a tradução de processos BPEL para programas em Lotos e também a transformação reversa. Duas opções para o desenvolvimento e validação dos processos estão disponíveis. Na primeira, o projeto e a verificação são realizados em BPEL através do uso das ferramentas de álgebra de processos. Na segunda opção, o projeto e a verificação são feitos em álgebra e o *framework* gera o código BPEL correspondente. Essas duas abordagens podem também ser combinadas. A verificação de propriedades utiliza técnicas de lógica temporal, simulação e bissimulação[3]. O desenvolvimento de um processo é feito em refinamentos sucessivos em que é possível mostrar a eqüivalência entre duas especificações. Bissimulação é também usada para verificar compatibilidade de serviços diferentes de um dado processo.

Ouyang et. al. (26) utilizam rede de Petri para a análise e verificação de processos BPEL. O trabalho é restrito ao fluxo de controle, não considerando valores de dados nem comunicação com *web services*. São definidos dois conjuntos de regras aplicáveis ao mapeamento BPEL para rede de Petri: um informal e outro formal. O processo BPEL é traduzido para uma linguagem baseada em XML, representando a rede de Petri do processo. Em seguida, utiliza-se a ferramenta WofBPEL para a análise do processo. Esta ferramenta é capaz de determinar se uma dada atividade é alcançável ou não

[2] *Finite State Processes* é uma notação de álgebra de processo que especifica *Label Transition Systems*, em forma textual.

[3] Uma *bissimulação* é uma relação de eqüivalência entre sistemas de transição de estados que associa sistemas que têm o mesmo comportamento no sentido de que um sistema simula o outro e vice-versa. Isso significa que os dois sistemas não podem ser distinguidos por um observador externo.

no processo (*reachability property*) e detectar atividades que competem por uma mesma mensagem (operação de recepção com os mesmos atributos: *partnerLink*, *portType*, *operation* e *correlationSet*) em um dado instante. WofBPEL é também capaz de fornecer informações para o sistema de execução sobre as possíveis seqüências de eventos, de modo que este possa gerenciar mais eficientemente as filas de eventos durante a execução de um processo, descartando os eventos que nunca ocorrerão.

O desenvolvimento de Brogi et. al. (5) é sobre a linguagem WSCI (*Web Service Choreography Interface*). Essa linguagem é aplicada à descrição de coreografias e possui características semelhantes a BPEL. Nesse trabalho, uma descrição WSCI é traduzida para a álgebra de processos denominada CCS (Calculus of Communicating Services), predecessora de Pi-Calculus. Os autores justificam que não há necessidade de se usar Pi-Calculus porque mobilidade não é requerida para formalizar a coreografia descrita em WSCI (todos os canais de comunicação são conhecidos estaticamente e não existem elementos como fábricas de objetos e *web services*). A proposta consiste em verificar propriedades como compatibilidade entre *web services* em termos de comportamento dos processos (a compatibilidade de interfaces é obtida através das especificações WSDL) e substituição do uso de *web services* de forma transparente para os clientes externos - essa propriedade pode ser usada, por exemplo, quando há a necessidade da substituição de um *web service* por outro.

Puhlmann e Weske (30) discutem o uso de Pi-Calculus como uma base formal para gerência de processos de negócios que inclui *workflow*, orquestração e coreografia de *web services*. Ainda na linha de utilização de Pi-Calculus, Puhlmann (29) define regras para tradução dos *padrões de workflow* (38) para Pi-Calculus. Cada padrão é formalizado em expressões Pi-Calculus concisas e não-ambíguas, que definem precisamente o seu comportamento. Puhlmann argumenta que Pi-Calculus possui características, como suporte à mobilidade e comunicação dinâmica, que não são usadas na modelagem dos padrões. Porém, isso pode justamente provocar novos desafios para a evolução dentro desse domínio. Wohed et. al. (39) fazem um estudo comparativo entre BPEL e os padrões de *workflow*.

Recentemente, foi formado por pesquisadores um grupo denominado Process Modeling Group (28), cujo principal objetivo consiste no desenvolvimento de trabalhos teóricos para aplicação na melhoria da qualidade dos produtos de software comerciais. As linguagens escolhidas como alvo são: BPEL , CDL (*Choreography Description language*) e RAD (*Role Activity Diagram*). As técnicas formais a serem utilizadas são: Rede de Petri, Pi-Calculus e notação Z. Um outro objetivo do grupo é o aprofundamento do entendimento científico da modelagem de processos para o desenvolvimento de padrões universais.

A tabela 1.1 contém um quadro comparativo entre as formalizações e análises de quatro trabalhos relacionados e mencionados anteriormente - 1. Oyuang et. al. (26), 2. Ferrara (13), 3. Koshkina (19), 4. Foster (14) - e o que está sendo proposto nesta dissertação (MVP2).

	1	2	3	4	MVP2
Formalismo	rede Petri	Lotos	BPE	FSM	Pi-Calculus
Ligações de controle	S	N	S	N	S
BPEL completo	S	S	N	N	N
Sentido de tradução	1	2	1	2	1
Ferramenta para análise	S	N	S	S	S

Tab. 1.1: Quadro comparativo entre as formalizações e análises de trabalhos relacionados.

O primeiro item contém a técnica de formalização utilizada. A próxima linha indica se as ligações de controle[4] são consideradas na tradução. Em seguida, o próximo item assinala se a linguagem inteira está sendo analisada ou apenas uma parte, como, por exemplo, esse trabalho que não aborda escopo, tratamento de falhas, eventos e compensações. A linha seguinte contém o número de sentidos de tradução: o valor 1 significa a tradução BPEL para a ferramenta formal e o valor 2 representa a tradução em ambos os sentidos. O último item indica se uma ferramenta para análise e verificação é provida.

A motivação inicial para o desenvolvimento do trabalho apresentado nesta dissertação surgiu quando foram iniciados os estudos sobre as linguagens de descrição de processos de negócios baseadas na composição de *web services*. Primeiramente, o foco voltou-se para a linguagem BPML (*Business Process Modeling Language*) e, em seguida, para a linguagem BPEL. Algumas referências, como em Simth e Fingar (33), ressaltam que essas linguagens foram desenvolvidas tendo Pi-Calculus como base formal. A partir deste ponto, o objetivo do trabalho voltou-se para a construção de um modelo que, baseado nesta relação das linguagens com o Pi-Calculus, contribuísse para o ciclo de desenvolvimento dos processos. Fazendo um paralelo com os trabalhos desenvolvidos em Koshkina (19) e Foster (14), surgiu a idéia de construção de uma Máquina Virtual Pi-Calculus como fonte da geração dos caminhos possíveis de execução de um programa Pi-Calculus traduzido a partir de um processo BPEL e, desta maneira, permitir a verificação de propriedades do processo antes da fase de execução.

1.4 Estrutura da dissertação

Esta dissertação encontra-se organizada, além deste capítulo introdutório, da seguinte forma:

- o capítulo 2 introduz os aspectos básicos sobre Gerência de Processos de Negócios (BPM - *Business Process Management*), Arquitetura Orientada a Serviço (SOA - *Service Oriented Ar-*

[4]Ligações de controle são os elementos do BPEL que permitem o sincronismo entre atividades paralelas (seção 3.1.3).

chitecture) e ferramentas de validação de software exploradas na seqüência do trabalho. BPM é definido como o gerenciamento do ciclo dos processos de negócio de uma corporação e é apresentado nas perspectivas tecnológica e do ciclo de vida. A arquitetura SOA é conceituada como a modelagem de sistemas voltada para os processos de negócio. Na modelagem dos serviços em SOA, algumas propriedades, como por exemplo, acoplamento fraco, reusabilidade, escalabilidade e granularidade grande, devem ser consideradas. Em termos de tecnologia, *web services* é a que atende melhor às propriedades e aos requisitos de SOA (11). A arquitetura *web services* é definida e descrita através da pilha de tecnologias aplicáveis, entre as quais BPEL que está na camada denominada *composição*. O capítulo discute brevemente outros modelos de validação de software tais como Rede de Petri, Álgebra de Processos e Linguagem Z. Os dois primeiros são usados na modelagem da concorrência, controle de fluxo e computação distribuída, e o último é uma notação formal para modelagem de sistemas baseada na teoria de conjuntos e na lógica de predicados;

- o capítulo 3 contém uma introdução sobre BPEL e Pi-Calculus necessária ao entendimento do desenvolvimento dos capítulos seguintes da dissertação. BPEL é uma linguagem para a construção de processos a partir da composição de *web services*. Os elementos da linguagem permitem definir: os *parceiros* que provêem os *web services* usados por um processo; *variáveis de execução*, que por definição são persistentes ao longo do processo; *conjuntos de correlações* que permitem identificar diferentes instâncias de um processo através de parâmetros das mensagens; *tratadores de falha* que permitem separar os códigos de exceção do fluxo normal; *tratadores de compensação* que são elementos típicos de processos de longa duração e que são empregados para desfazer operações já completadas (usados em casos de falha ou cancelamento do processo); *tratadores de eventos*, que habilitam o processo a receber uma mensagem de forma assíncrona ao fluxo normal, ou seja, a qualquer momento o processo pode ser interrompido para tratar a ocorrência de um evento; e, por último, as *atividades* que são usadas para compor a lógica do processo em fluxos paralelos que se sincronizam através do uso de *ligações de controle*. A abordagem de Pi-Calculus dada nesse capítulo contém os elementos que são usados para atender o objetivo deste trabalho. Neste sentido, são definidos o conceito de *rendez-vous* síncrono, que é o mecanismo de comunicação do Pi-Calculus e as operações: leitura através de um canal, escrita em um canal, alocação dinâmica de um canal, construções de paralelismo, seqüência e soma, e chamada recursiva de processos. As variáveis lógicas definidas em Pi-Calculus são usadas para a construção das expressões lógicas que permitem o mapeamento das ligações de controle do BPEL;

- as regras de mapeamento dos processos BPEL para programas Pi-Calculus são definidas no

capítulo 4. Inicialmente, são estabelecidas algumas considerações. Os canais Pi-Calculus são classificados entre internos e externos. Os canais internos são usados na tradução do fluxo de controle e nas operações de sincronismo entre os fluxos paralelos. Os canais externos são aplicáveis à comunicação com os *web services* e para a execução de atividades BPEL que são relevantes na análise do processo. As regras para a tradução das atividades são apresentadas sem considerar os atributos-padrões e nos elementos-padrões[5], que atuam para formar o sincronismo entre as atividades. As regras para essas construções são tratadas em seguida, e se aplicam a qualquer atividade do BPEL. Para completar, são descritas as regras para as atividades estruturadas mostrando a necessidade de se atribuir valores para as ligações de controle de atividades internas à uma atividade estruturada que não é executada. O capítulo termina com o exemplo da tradução do processo de uma Agência de Viagem (19);

- o capítulo 5 elabora a descrição da Máquina Virtual Pi-Calculus (MVP). Inicialmente é apresentada a linguagem da MVP, cuja sintaxe é definida por um esquema XML contendo as seguintes operações: ler, escrever, alocar um canal, construir paralelismo, seqüência e soma, replicar um trecho de programa, executar um processo e terminar a execução. Na seqüência, é apresentada a arquitetuta da MVP que é estruturada em cinco blocos funcionais: o *Processador* que executa o programa Pi-calculus passo-a-passo; o *Gerador de Seqüências* que controla a execução passo-a-passo do *Processador* para produzir todas os caminhos de execução possíveis, gerando a Árvore de Execução; o *Analisador* que recebe a Árvore de Execução como entrada, gera a Tabela de Execução *xtable* e provê o comando SELECT que faz operações sobre Tabelas de Execução - é através desse comando que se faz a verificação das propriedades do processo -; o bloco funcional *Shell* permite ao usuário executar os serviços da MVP, entre os quais, carregar e executar um programa, depurar um programa, visualizar a Árvore de Execução e executar comandos SELECT; e por fim, o bloco *Básico* contém as classes que são comuns aos demais blocos. A *Árvore de Execução* é o resultado da execução do programa em que cada nó corresponde a um passo do processamento e contém os seguintes atributos: o canal em que ocorreu o *rendez-vous*, a expressão naquele momento e uma referência de equivalência[6]. A *Tabela de Execução* é uma estrutura derivada da Árvore de Execução em que cada coluna contém um caminho de execução descrito por uma seqüência de canais externos (*web services*, atividades relevantes e os canais de *status*[7].). O capítulo termina apresentando dois exemplos de análise de processos (de uma Agência de Viagem e de uma Livraria (19)), e um exemplo de desenvol-

[5] As operações sobre as ligações de controle são definidas nos atributos-padroões e nos elementos-padrões.

[6] A referência de equivalência aponta para um outro nó da árvore que contém a mesma expressão.

[7] No fim do processamento de um caminho, o Gerador de Seqüência introduz um canal externo, denominado canal de *status*, que informa se o término foi normal ou se houve *deadlock*. No último caso, o nome do canal de *status* contém a concatenação dos nomes dos canais que estão em *deadlock*

vimento e análise de um processo fim-a-fim de Gerência de SLA (*Service Level Agreement*) descrito a partir de especificações do TMForum (36). Nesse documento, o processo fim-a-fim é definido como fluxos de interações entre os processos do Modelo de Referência (eTOM - *enhanced Telecommunication Operation Map*). As partir dessas especificações, é construído o processo BPEL que é traduzido para Pi-Calculus para três tipos de análises: atendimento às especificações, otimização e situações de *deadlock*;

- finalmente, o último capítulo contém a conclusão e sugestões para a evolução dos estudos na linha de trabalho aqui proposta.

Capítulo 2

Conhecimentos Preliminares

Este capítulo apresenta os assuntos que formam a base do desenvolvimento deste trabalho. Inicialmente, a seção 2.1 introduz Gerência de Processos de Negócio (*Business Process Management* - BPM) ressaltando dois aspectos, a evolução dos sistemas de informação de suporte a processos e o ciclo de vida de um processo. A seção 2.2 conceitua Arquitetura Orientada a Serviços (*Service Oriented Architecture* - SOA) referente a negócios e requisitos tecnológicos. A tecnologia de *web services* é descrita na seção 2.3 como uma implementação de SOA. A seção 2.4 contém uma visão geral sobre aplicação de técnicas em modelagem de comportamento de programas.

2.1 Gerência de Processos de Negócio (BPM)

Entende-se por Gerência de Processos de Negócio o gerenciamento do ciclo de vida dos processos de negócio de uma corporação. Esse ciclo de vida consiste de fases que vão desde a descoberta ou especificação de um processo até a análise para a sua melhoria contínua. Sistemas de Gerência de Processos de Negócios (*Business Process Management Systems* - BPMS) formam uma área de aplicação de Sistemas de Informação que são dedicados a suportar esse ciclo de vida.

Evolução Tecnológica

Implicitamente, qualquer sistema de informação tem como objetivo dar suporte aos processos de negócio. Em suas primeiras gerações, esses sistemas eram construídos no topo dos sistemas operacionais, tendo como objetivo automatizar partes de um processo maior. Em geral, esses sistemas eram aplicações específicas de uma corporação. A evolução desse estágio ocorreu com a ampliação da camada básica que passou a abranger aplicações genéricas como, por exemplo, Sistemas Gerenciadores de Banco de Dados e aplicações de domínio específico, como ERP (*Enterprise Resource Planning*) e Sistemas de *Call Center*. Com esses tipos de aplicações, o suporte aos processos de

negócio foi ampliado, pois essas tratam de abstrações mais próximas do negócio. Simultaneamente, surgiram os Sistemas Gerenciadores de *Workflow* com o objetivo de gerenciar os fluxos das tarefas de um processo. Para que isso pudesse ocorrer de forma flexível, era necessário tirar as regras de negócio correspondentes às decisões sobre os fluxos de execução das tarefas dos sistemas específicos, disponibilizando interfaces que permitiam implementar essas regras nos processos definidos nos Sistemas de *Workflow*. Para dar suporte ao desenvolvimento dessas interfaces, surgiram as plataformas (*middleware*) de integração de aplicações, conhecidas como *Enterprise Application Integration* - EAI.

Outras evoluções têm contribuído para a abordagem atual em BPM. Uma delas é a mudança da visão de dados para a de processos. O foco dos sistemas de informação consistia no armazenamento e na busca de dados e, como resultado, a modelagem de dados era o ponto de partida desses sistemas. Os processos precisavam se ajustar aos modelos de dados (1). Outra evolução refere-se às estratégias no desenvolvimento de sistemas. Poucos sistemas atualmente são construídos a partir do zero. Cada vez mais há necessidade de mudanças e em tempos curtos. Tecnologias de componentização e de serviços web (*web services*) são usadas como suporte a esses requisitos, provendo encapsulamento e interoperabilidade, entre outras características favoráveis à integração e reusabilidade.

Considerando as tendências apresentadas nos parágrafos anteriores, mais o estágio atual das arquiteturas dos processos das empresas, em que existe uma forte participação de fornecedores e parceiros - o *outsourcing* vem sendo usado de forma intensa -, pode-se entender porque as comunidades que desenvolvem sistemas de BPM estão definindo linguagens de construção de processos que permitem estruturar processos de negócio como uma composição de *web services*. BPEL é a linguagem mais usada nos tempos recentes. Outras linguagens foram definidas com os mesmos objetivos, tais como: BPML, WSCI, XLANG e WSFL.

Ciclo de Vida dos Processos

O ciclo de vida no gerenciamento de processos de negócio abrange oito capacidades: descoberta, projeto, implantação, execução, interação, controle, monitoração e análise do processo (33).

A figura 2.1 ilustra o ciclo de vida de um processo. Um processo é colocado em produção a partir da fase de Implantação. As quatro capacidades que ocorrem em paralelo - Execução, Monitoração, Interação e Controle - constituem a etapa de Operação do processo.

As fases de Descoberta, Projeto e Implantação dos processos podem ser confundidas com as etapas do desenvolvimento tradicional de sistemas. A diferença começa na primeira fase, denominada Descoberta. Os requisitos a serem capturados na abordagem BPM são apenas aqueles importantes para serem considerados na perspectiva do negócio. Um processo de negócio é modelado como um conjunto de tarefas. Em geral, essas tarefas correspondem a funcionalidades que são implantadas em

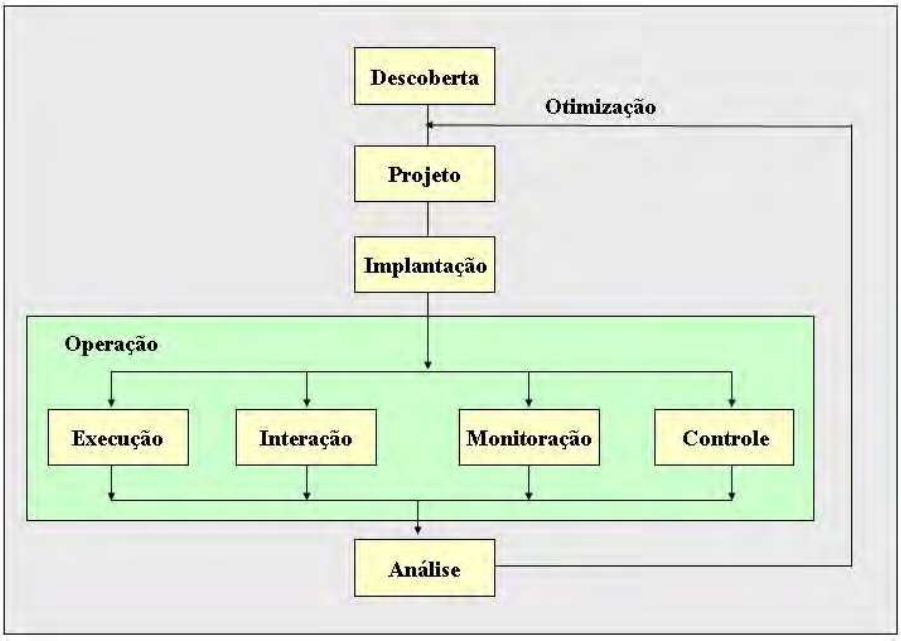

Fig. 2.1: Ciclo de vida de um processo.

outros sistemas ou mesmo em outros processos.

A fase de Descoberta consiste em capturar os requisitos de negócio a partir de um funcionamento já existente dos processos ou mesmo definir novos processos que virão a ser implantados na organização. Como resultado dessa fase, é de se esperar um artefato que mostre como o processo de negócio deve operar, incluindo o conhecimento que todos os participantes devem ter do processo, bem como o inter-relacionamento entre os sistemas e os papéis desempenhados pelos clientes, fornecedores, parceiros e funcionários da corporação.

Na fase de Projeto, o processo é modelado em termos de atividades, regras, participantes, interações e relacionamentos. Algumas organizações de padronização têm trabalhado na definição de uma linguagem gráfica para a modelagem de processos que possam ser traduzidas facilmente para as linguagens de execução. Um exemplo é BPMN (*Business Process Modeling Notation*) da organização BPMI (*Business Process Management Institute*) que pode ser mapeada diretamente para BPEL (25).

Essa fase é iniciada pela estruturação do processo em atividades. Estas podem ser mapeadas como tarefas que são executadas por pessoas ou sistemas internos ou externos à corporação. Quando ocorre a interação com sistemas, há a necessidade de se ter as interfaces bem definidas. As plataformas conhecidas como EAI (*Enterprise Application Integration*) ou ESB (*Enterprise Service Bus*) dão suporte para a implementação dessas interfaces.

Alguns BPMS suportam a validação de modelos ou simulações para que os processos quando im-

plantados tenham um mínimo de defeitos e já estejam em um certo grau de otimização. Simuladores usam as estimativas de tempo de execução e custo, recursos requeridos pelas atividades e probabilidades de ocorrência de eventos. É possível obter resultados tais como: valores médios e desvios-padrões de tempos de transações e do processo fim-a-fim, identificação de gargalos no processo e alocação de recursos, tempo médio de utilização de atividades humanas, etc.

Essa fase também inclui a definição das métricas de desempenho do processo para que o analista de negócio possa avaliar o processo implantado e, possivelmente, reestruturá-lo em função de oportunidades ou de outros motivos. Os estudos realizados na simulação ajudam a definir as variáveis de desempenho.

A fase de Implantação consiste na configuração do repositório do BPMS para suportar o novo processo e na implantação e configuração das aplicações externas que interagem com o processo.

A fase de Operação é iniciada pelas execuções dos processos. Um BPMS é responsável por manter os contextos de execução dos processos, que inclui a atividade que está em execução, valores de variáveis, contextos de sub-processos, *logs*, etc. Um BPMS pode ficar indisponível por algum instante, mas os processos reais continuam acontecendo e evoluindo. Quando o sistema for reativado, devem existir mecanismos que recuperem os processos do ponto em que houve a indisponibilidade para o estado real. Os BPMS devem possuir transações distribuídas e de longa duração, pois outros sistemas são envolvidos na composição dos processos e uma tarefa pode durar dias e não poucos segundos, como em sistemas on-line. Os BPMS devem ter facilidades de integração com um conjunto de outras plataformas que executam os sistemas externos.

Através de *desktops* ou de portais, as pessoas interagem com os processos. Essas interações podem ocorrer tanto por motivos de participação ou colaboração no progresso do processo, como para o monitoramento e intervenção para a solução de problemas.

Monitoramento e Controle têm como foco garantir o andamento normal da execução do processo, tanto na visão do negócio como na sustentação da plataforma na qual o processo está sendo executado. São duas classes de gerenciamento diferentes, mas ambas têm o mesmo propósito: que o andamento do processo dê um resultado efetivo para os negócios. Em função de resultados obtidos na monitoração, ações podem ser tomadas, tais como: atualização do programa que descreve um processo - quando, por exemplo, detecta-se um erro no programa - ou alteração de participantes de um processo em função de novas pessoas terem sido contratadas, etc.

Na fase de Análise, a medida do desempenho do processo provê informações de melhorias operacionais como, também, para a descoberta de novas oportunidades de inovação. Nessa fase, deve-se obter uma visão clara dos recursos e tempos relacionados no andamento fim-a-fim do processo. É importante ter-se a visão fim-a-fim, pois é a partir dela que se pode fazer uma análise *top-down* para determinar a influência de cada passo ou sub-processo no resultado global. Alguns sistemas possuem

linguagens que permitem efetuar consultas nos processos de maneira similar às linguagens para banco de dados.

Para dar suporte a essas duas fases, Monitoramento e Análise, existe uma classe de sistemas denominada *Business Activity Monitoring* (BAM). Processos são instrumentados com sensores para monitorar suas atividades e variáveis. Através desse tipo de sistema, é possível construir regras para definir eventos compostos a partir das métricas individuais. Um *dashboard* [1] permite a monitoração, análise e respostas aos eventos e às exceções que ocorrem em tempo real.

Por fim, a Otimização traz para a metodologia o conceito de melhoria contínua, fechando o ciclo. A otimização é o resultado da fase de análise. Os BPMS devem ser flexíveis de modo a permitir que melhorias possam ser implantadas em diversas situações, tais como: processos que ainda estão em execução; processos que serão iniciados a partir de um dado instante; e processos que serão iniciados sob uma nova condição a ser definida.

2.2 Arquitetura Orientada a Serviços (SOA)

SOA do ponto de vista de negócios

Partindo da concepção de Processos de Negócios, a principal meta da Arquitetura Orientada a Serviços (SOA) é alinhar o mundo de negócios com o mundo da tecnologia da informação (TI), de modo a tornar ambos mais efetivos. A prática de se modelar sistemas de informação a partir dos *projetos* dos processos de negócio define uma Arquitetura Orientada a Serviços. Os processos e as tarefas que os sistemas de informação implementam são definidos como *serviços* (17). Uma segunda meta em SOA é prover uma infra-estrutura técnica que possa ser reconfigurada de maneira fácil quando ocorrerem mudanças nos requisitos de negócio. Isso se obtém através da separação das interfaces dos serviços de suas implementações. As operações de cada serviço podem ser implementadas usando tecnologias tais como J2EE e CORBA, mas nenhum detalhe de quaisquer dessas tecnologias pode ser visível para os consumidores dos serviços (32).

Não existe nenhuma regra que define se uma dada função, ou uma aplicação, é ou não um serviço na arquitetura SOA (17). Em geral, as pessoas de negócio entendem claramente o conceito de serviço porque é o mecanismo básico pelo qual negócios são feitos (35).

Na definição e modelagem dos serviços em SOA, convém considerar algumas propriedades, das quais as principais são descritas a seguir.

[1] *Dashboard* em BPM é uma janela da interface de operação contendo um painel de controle com os principais indicadores de desempenho dos processos gerenciados.

- Acoplamento fraco (*loose coupling*): SOA é um estilo de arquitetura cujo principal objetivo é ter um acoplamento fraco entre o consumidor e o provedor do serviço. Isso significa que o consumidor tem um conhecimento mínimo sobre o servidor. Esse conhecimento deve ser o suficiente para execução dos serviços. Por exemplo, não deve haver nenhuma restrição ao consumidor em termos de plataforma de implementação e linguagem de programação do provedor. O serviço deve ser invocado a partir do envio de mensagens através de uma interface publicada, e não do uso de uma API. Atualmente, a descrição das interfaces através de documentos em XML é aceitável para dizer que existe um acoplamento fraco entre consumidor e provedor.

- Reusabilidade: na modelagem dos serviços, encoraja-se a reusabilidade, mesmo que não haja nenhum requisito imediato. Ainda quanto a esse aspecto, o uso de serviço em composições deve ser considerado. Veja "Capacidade de ser usado em composições" adiante.

- Interface bem-definida: todo serviço define uma interface que expõe suas funcionalidades e os modos de invocar o serviço de maneira interoperável. A definição da interface deve ser independente dos detalhes de implementação do serviço.

- Baseada em padrões abertos: o uso de padrões no projeto e implementação dos serviços aumenta a independência de um único fornecedor bem como amplia a possibilidade de uso por um número maior de clientes.

- Ser publicado para descoberta: a publicação de serviços deve permitir que estes sejam descobertos sem que o cliente necessite conhecer previamente a localização dos provedores do serviço. Um provedor deve usar metadados para definir as funcionalidades e restrições dos serviços.

- *Stateless*: um serviço não deve depender de nenhum estado definido previamente, ou seja, esse deve ser autocontido de forma que uma única interação seja capaz de levar o sistema de um estado consistente para outro, no que se refere ao serviço que é prestado. Esse modelo é mais escalável, pois a qualquer momento o servidor está apto a prover o serviço. Também a confiabilidade é maior, pois os mecanismos de recuperação de falhas são mais simples.

- Segurança: provedores de serviço precisam considerar os aspectos de segurança quando usarem a Internet como meio de transporte.

- Versionamento: devido à característica do acoplamento fraco, há a necessidade de se implementar o versionamento do serviço no nível da interface, de modo que num dado instante o provedor possa disponibilizar várias versões de um mesmo serviço. Isso permite que os clientes não precisem evoluir na mesma velocidade dos provedores.

- Granularidade grande: é recomendado o uso de uma granularidade grande para se obter maior consistência e menor *overhead* na interação entre o provedor do serviço e o cliente. Em alguns casos, pode-se fazer o uso de orquestração de serviço para criar uma interface de grão maior, a partir de operações de grão menor.

- Qualidade de serviço: os principais requisitos de qualidade de serviço são: prover a dinâmica de operação de maneira segura, protegendo os conteúdos das mensagens e os acessos aos serviços; ter o nível de confiabilidade para garantir que uma mensagem é entregue no destinatário ou que uma notificação de falha seja recebida pelo agente que origina a mensagem; capacidade transacional para as tarefas de negócio de maneira que, se ocorre uma falha em uma tarefa, então a lógica de exceção é executada.

- Capacidade de ser usado em composições: uma aplicação ou um serviço de um provedor deve ser modelado de forma que suas implementações usem composições de outros serviços, com abordagem ou não de orientação a processos (veja a seguir em "BPM e SOA"). Portanto, é importante que os serviços tenham a característica de serem facilmente usados em composições. Isso se alcança através de um projeto que considere fortemente as características de reusabilidade e modularidade.

Observando-se cada uma das propriedades acima, vê-se que elas realçam um determinado aspecto, porém, não são necessariamente independentes pois existem alguma forma de inter-relacionamentos entre elas.

BPM e SOA

É importante nesse ponto definir a relação BPM e SOA. Partindo da metodologia BPM, como já descrita anteriormente, um processo de negócio é modelado como um conjunto de tarefas. Essas tarefas são funcionalidades - em geral executadas em outros sistemas - que podem ser modeladas como serviços da arquitetura SOA. Porém, é interessante que um serviço associado ao conceito de negócio possa ser implementado como um processo e, portanto, possa utilizar a metodologia e a infra-estrutura BPM. Assim, tem-se uma relação complementar e recursiva entre as duas abordagens (24).

A figura 2.2 ilustra a relação entre BPM e SOA. Na camada superior, localizam-se os processos de negócio. No exemplo, o primeiro processo (à esquerda) é iniciado através de um evento externo e implementado como uma composição de serviços oferecidos na camada de serviços. O segundo processo é a implementação de um serviço que pode ou não usar outros serviços oferecidos. A figura não mostra, mas vale ressaltar que esses serviços podem ser disponibilizados externamente a uma corporação por fornecedores ou parceiros. A última camada, Aplicações e Recursos, ilustra que um

serviço pode ser implementado por sistemas desenvolvidos em tecnologias proprietárias ou mesmo usar sistemas legados.

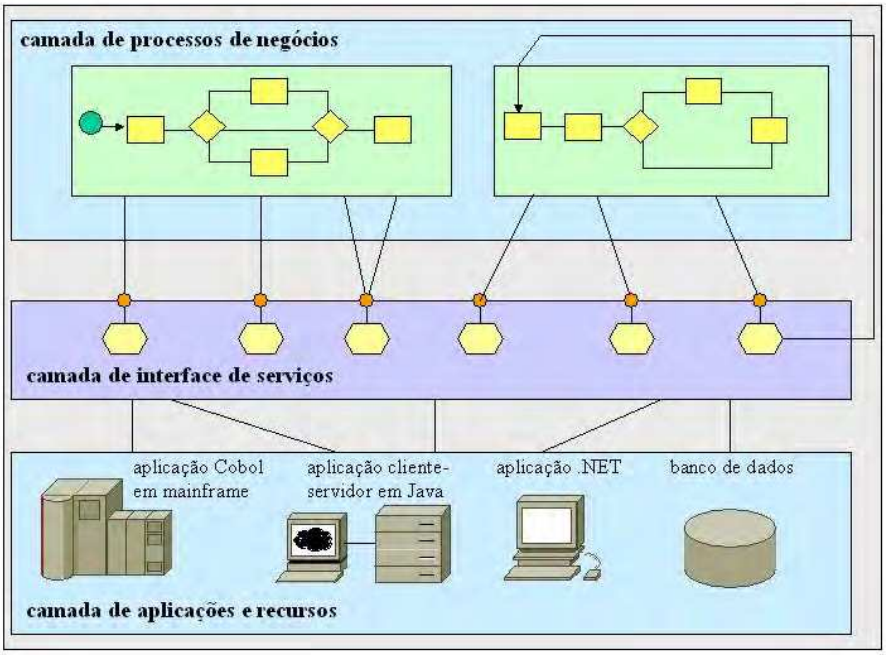

Fig. 2.2: Relação BPM-SOA.

SOA na perspectiva tecnológica

No que tange à dimensão tecnológica, SOA é considerada uma evolução da tecnologia de sistemas distribuídos. As plataformas devem dar suporte às implementações dos serviços em conformidade com as propriedades descritas acima. Neste trabalho, estamos interessados nos serviços implementados como *web services*, visto que BPEL define que as tarefas externas são executadas através de interfaces nesse padrão.

2.3 *Web Services*

O consórcio W3C (World Wide Web Consortium) define *web services* da seguinte maneira:

"Um *Web Service* é um sistema de software projetado para suportar interoperação máquina-máquina sobre uma rede. A interface de um *web service* é descrita em um formato que é interpretado de forma automatizada (WSDL). Outros sistemas interagem com o *web service* através do uso da descrição de mensagens SOAP, tipicamente usando HTTP com serialização XML, em conjunto com outros padrões relacionados à *Web*".

O modelo básico de *web services* possui três agentes: provedor do serviço, consumidor do serviço e um servidor de diretório, que é opcional. Um cenário típico de funcionamento pode ter os seguintes passos:

- o provedor do serviço implementa e define sua interface em um documento WSDL e registra esse serviço em um diretório através de UDDI (*Universal Description Discovery and Integration*);
- o consumidor do serviço localiza o serviço desejado no diretório e obtém a sua descrição (em WSDL);
- O consumidor invoca as operações definidas na interface do provedor através do uso de SOAP.

Esse modelo básico passou a ser aceito pela comunidade como um padrão para interoperabilidade entre *web services*. Porém, a comunidade de TI (Tecnologia da Informação) sentiu a necessidade de definir novos padrões para endereçar problemas mais complexos com o objetivo de suportar a interoperabilidade em um nível mais elevado. Os principais assuntos tratados são: transações, segurança, confiabilidade, coreografia e orquestração (composição).

A figura 2.3 ilustra uma pilha da família de tecnologias aplicáveis à arquitetura *web services*.

Um conceito principal na arquitetura *web service* é que a linguagem básica em que todas as especificações são definidas é o XML. Os principais objetivos de se construir as especificações através de XML são: possibilitar o desenvolvimento de extensões e obter neutralidade em termos de plataforma e linguagem. A seguir, apresentam-se os principais temas e as respectivas especificações adotadas em *web services*.

Serviços de Transporte

A camada de transporte define os protocolos de rede mais familiares, tais como, HTTP, FTP, SMTP e outros, para promover a comunicação entre *web services*.

Serviços de Mensagens

Os serviços de mensagens incluem duas especificações e tecnologias: SOAP e WS-Addressing. SOAP é um protocolo que estrutura a comunicação entre *web services*. Uma mensagem SOAP contém um envelope, que por sua vez contém um cabeçalho (opcional) e um corpo (obrigatório). O cabeçalho é um elemento genérico, que permite extensões. O corpo contém a mensagem que é para ser entendida pelo destinatário do envelope. Embora não seja obrigatório um protocolo de transporte específico, o mais usado é o HTTP. SOAP permite padrões de comunicação variados, como, por

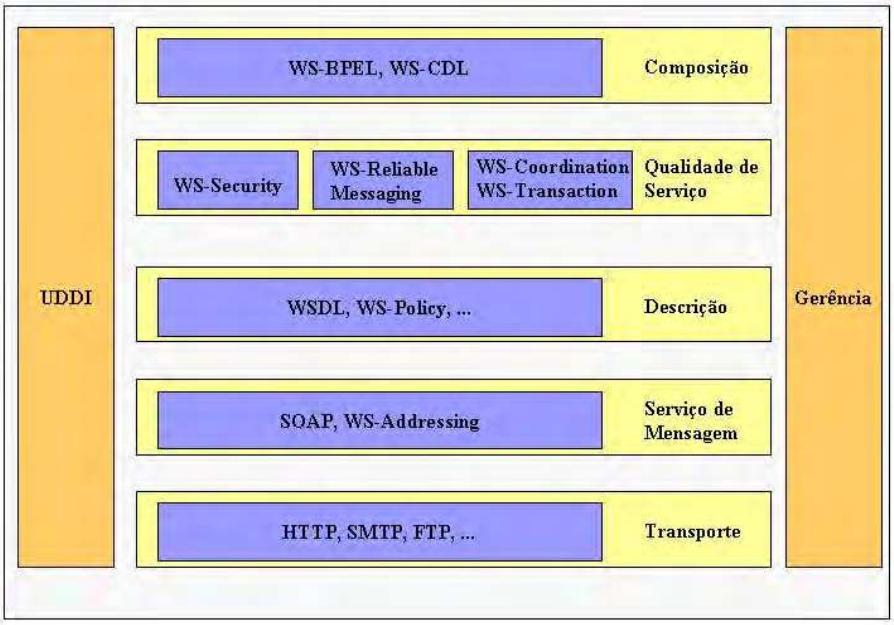

Fig. 2.3: Arquitetura Web Service.

exemplo, RPC (*remote procedure call*) pode ser adotado através de comunicação síncrona (pedido/-resposta) e, para os casos em que a latência não é controlada, utiliza-se o mecanismo de comunicação assíncrona. Uma mensagem SOAP enviada de um nó origem passa por nós intermediários até chegar ao destino. A mensagem SOAP é encaminhada com base nos cabeçalhos e no corpo. O cabeçalho da mensagem pode ser alterado no percurso para conter informações que são relevantes para análise e envio de resposta.

WS-Addressing provê mecanismos independentes de protocolo de transporte para endereçar *web services* e mensagens. São definidas duas construções: a referência a um ponto final (EPR - *endpoint reference*) e o cabeçalho de informação de mensagens (*MI header*). Um EPR é composto dos seguintes atributos: um *endereço* especificado através de uma URI; *propriedades* e *parâmetros* dependentes do web service envolvido na operação; *tipo de porta* e *tipo de serviço* que identificam a localização da descrição do serviço; e a *política* (conforme as especificações WS-Policy) que fornece informações das regras de comportamento relevantes para a interação entre os *web services*.

Um *MI header* inclui: a *identificação* da mensagem; um atributo de *relacionamento* entre duas mensagens; endereço de *destino* da mensagem especificado através de uma URI; a *ação* a ser executada e três EPRs. Os EPRs servem para referenciar a *origem* da mensagem, o local da *resposta* e o local para o qual deverá ser enviada a notificação de *falha*, caso ocorra.

Descrição dos Serviços

Para descrição dos serviços oferecidos por um *web service*, definem-se metadados que contêm as características dos serviços disponíveis na rede. Duas especificações servem para compor as descrições dos serviços: WSDL (*Web Service Descrption Language*) e WS-Policy.

Através do WSDL, definem-se as interfaces públicas dos *web services*. Um documento WSDL é estruturado em serviços. Um serviço é uma coleção de *endpoints* e implementa uma interface. Uma interface é um conjunto de operações que têm parâmetros de entrada e saída. Pode-se empregar o mecanismo de extensão similar ao usado em programação orientada a objetos. Um documento WSDL também inclui a definição de novos tipos de dados e construções de mapeamento para SOAP e HTTP.

WS-Policy provê informações sobre como o serviço é disponibilizado na rede e o que o provedor do serviço espera do consumidor. Essas informações são referentes principalmente a ambiente, segurança, transações e confiabilidade.

Descoberta dos Serviços

UDDI (*Universal Description and Discovery Interface*) define um serviço de agregação de metadados e especifica protocolos para consulta e registro em repositórios de informações comuns aos *web services*. As consultas aos repositórios pelos programas servem para buscar onde um determinado serviço é implementado e quais são as propriedades de mapeamento.

WS-Policy propõe um *framework* que estende as descrições dos serviços providos nos documentos WSDL. Essas descrições adicionais devem também estar registradas para que um consumidor possa obtê-las no momento da busca da implementação do *web service* que melhor atenda aos requisitos. Por exemplo, um consumidor pode querer um serviço de compra de livro *on-line* (interface WSDL) e que o mecanismo de débito no cartão de crédito use políticas de segurança de máxima eficiência (WS-Policy).

WS-MetaDataExchange define um protocolo que suporta a troca dinâmica de dados de WS-Policy e outros metadados relevantes para a interação dos serviços (esquema XML e WSDL).

Qualidade de Serviço

Essa camada inclui segurança, confiabilidade da entrega de mensagens e suporte a transações (11).

WS-Security compõe uma família de especificações para a segurança dos *web services*. Trata principalmente de definir protocolos que garantem integridade de dados e confidencialidade na troca de mensagens entre *web services*. O protocolo inclui detalhes para uso de SAML (*Security Assertion Markup Language*), Kerberos (protocolo de autenticação) e certificados de chaves-públicas como em X.509.

WS-ReliableMessaging endereça os problemas de garantia de entrega de mensagens entre *web services*. Apesar de serem utilizados protocolos de transporte que fornecem serviços que eliminam duplicação e perda de mensagens, no âmbito dos *web services* essas garantias não se mantêm. As propriedades básicas especificadas são: *entrega em ordem, no mínimo uma entrega e no máximo uma entrega*. Essas garantias podem ser combinadas entre elas.

Para dar suporte a transações distribuídas entre *web services*, existem duas especificações. WS-Coordination é a principal e define um mecanismo para o início e o acordo sobre o resultado de tarefas que envolvem múltiplos *web services* trocando múltiplas mensagens. WS-Transaction estende as especificações de WS-Coordination e tem dois mecanismos para prover transações: Transação Atômica (*Atomic Transaction*) e Atividade de Negócio (*Business Activity*). O primeiro especifica um protocolo para implementar a transação em duas fases com as quatro propriedades conhecidas: ACID (Atomicity, Consistency, Isolation, Durability). O segundo provê protocolos baseados em compensações aplicáveis a transações de longa duração.

Composição de *web services*

Os termos orquestração e coreografia tratam de dois aspectos da criação de processos de negócios a partir da composição de *web services*. Orquestração se refere a um processo de negócio executável que interage com *web services*. BPEL é a principal especificação que define a linguagem para se construir esses processos. Coreografia é a descrição da seqüência de mensagens que deve ocorrer em um *web service* para que esse possa produzir um resultado efetivo. Um padrão para essa descrição é o WS-CDL (*Choreography Description Language*) (21).

2.4 Modelagem de Comportamento de Software

Várias técnicas de modelagem de comportamento de software têm sido objeto de pesquisa e desenvolvimento. A idéia é que um processo, quando executado pelo sistema de execução de um BPMS, tenha o comportamento de um programa que contém lógica, fluxo, paralelismo e comunicação. Esta seção apresenta uma introdução sobre algumas técnicas que vêm sendo referenciadas nas pesquisas relacionadas à verificação de processos de negócios ou programas de um modo geral.

2.4.1 Rede de Petri

Rede de Petri é uma técnica de especificação de sistemas que possibilita sua representação matemática e possui mecanismos de análise que permitem a verificação de propriedades e a correção

do sistema. Usando-se Rede de Petri, pode-se modelar sistemas paralelos, concorrentes, assíncronos e não-determinísticos (22). Rede de Petri tem uma representação gráfica dos processos e da comunicação entre eles que facilita o entendimento de seu funcionamento.

Uma Rede de Petri é um grafo dirigido, formado por dois tipos de componentes: um ativo, denominado transição, e outro passivo denominado lugar. Arcos interconectam lugares através de transições. Cada arco tem um peso, que é a condição para que uma transição possa ocorrer. Cada lugar tem uma marca (ou um conjunto de fichas) que habilita ou não a condição de transição. Após a realização de uma transição, os lugares envolvidos terão suas informações alteradas.

Existem várias definições formais para Rede de Petri. A que se segue foi obtida de Desse a Juhás (10). Uma Rede de Petri é uma tupla:

$PN = (S, T, F, Mo, W, K)$, onde:

S é o conjunto de lugares;
T é o conjunto de transições;
F é o conjunto de arcos: $F \subseteq (S \times T) \cup (T \times P)$; F é sujeito à restrição na qual nenhum arco pode conter dois lugares ou duas transições;
$Mo : S \rightarrow \mathbf{N}$ é uma marca inicial, onde para cada $s \in S$ existem $n \in \mathbf{N}$ fichas;
$W : F \rightarrow \mathbf{N}^+$ é um conjunto que atribui para cada arco $f \in F$ um $n \in \mathbf{N}^+$, indicando quantas fichas são consumidas de um lugar quando ocorre uma transição;
$K : S \rightarrow \mathbf{N}^+$ é um conjunto de restrições de capacidade, que atribui para cada $s \in S$ um $n \in \mathbf{N}^+$, designando o número máximo de fichas que podem ocupar um lugar.

Van der Aalst et. al. (37) citam três bons motivos para se usar Rede de Petri na modelagem de *workflow* e de processos:

- a conjunção de semântica formal e de representação gráfica;
- descrição baseada em estados ao invés de eventos - a modelagem em álgebra de processos é baseada em eventos, sendo que os estados existem implicitamente entre eventos subseqüentes; o autor argumenta que alguns padrões de workflow WP16 e WP18 (38) mostram a importância dessa diferença;
- abundância de técnicas de análise.

Todavia, existem também problemas quando um processo é modelado em termos de Rede de Petri: carência de suporte a múltiplas instâncias de sub-processos; carência de suporte a padrões avançados para a sincronização de fluxos e o fato que o disparo de uma transição é baseado somente em *fichas*

de lugares locais - isso traz problema, por exemplo, na modelagem do padrão de cancelamento (*cancelation pattern*), que emprega um mecanismo de temporização global para a remoção de fichas de lugares selecionados.

2.4.2 Álgebras de Processos

Álgebras de Processos são linguagens formais usadas para especificar sistemas de software, principalmente aqueles constituídos de componentes concorrentes que se comunicam. Essas linguagens provêem um meio para descrever interações em alto nível, comunicações e sincronizações entre uma coleção de agentes ou processos independentes. São definidas leis que permitem manipular e analisar descrições algébricas dos processos como, por exemplo, verificar eqüivalências usando bissimulação. Exemplos dessas álgebras incluem CSP (*Communication Sequential Processes*), CCS (*Calculus of Communicating Systems*), ACP (*Algebra of Communicating Process*), Lotos, Promela, Pi-Calculus e Ambient Calculus.

Além de existir uma variedade de álgebras de processo, essas também têm variantes que incorporam comportamentos estocásticos, informações de tempo, etc. Porém, as principais funcionalidades comuns a todas elas são:

- representar interações entre processos independentes através de comunicação (mensagens), ao invés de variáveis compartilhadas;
- descrever processos e sistemas usando um conjunto pequeno de primitivas e operadores que combinam essas primitivas;
- definir leis algébricas para os operadores de processos permitindo que expressões de processos sejam manipuladas algebricamente.

A matemática de processos inicia-se com um conjunto de nomes (canais) que servem como meio de comunicação. Em algumas propostas, canais têm estruturas mais complexas, mas isso é abstraído nos modelos teóricos. Adicionalmente, há a necessidade de se formar novos processos. As principais operações comuns presentes na maioria das álgebras são:

- composição de processos paralelos;
- envio e recepção de mensagens através de canais;
- composição seqüencial;
- pontos de interações locais (sem interferência de outros processos);

- recursão e replicação de processos.

Existem algumas áreas em que as pesquisas em álgebra de processos são aplicadas; aquelas relacionadas ao objetivo desse trabalho são estudos de comportamento dos processos e técnicas de modelagem. A primeira trata de, a partir de uma descrição do processo, efetuar a análise e verificação do seu comportamento. A segunda é empregada para modelar processos a partir de suas especificações.

O uso de canais de comunicação é uma das características que distingue álgebra de processos de outros modelos como Rede de Petri, tratada na seção anterior.

Pi-Calculus, a álgebra de processo usada para a verificação de processos neste trabalho, caracteriza-se principalmente por permitir a mobilidade dos processos de um sistema.

2.4.3 Linguagem Z

A linguagem Z é uma notação de especificação formal usada para descrever e modelar sistemas computacionais, baseada em teoria dos conjuntos e lógica predicada de primeira ordem. A linguagem foi desenvolvida pelo *Programming Research Group* da Universidade de *Oxford* na década de 1970 (3). A linguagem foi padronizada pela ISO (*International Standard Organization*), sob o código ISO/IEC 13568 (primeira edição em 2002).

Uma característica da linguagem Z é o uso de tipos de dados matemáticos para modelar dados em um sistema. Esses tipos não são orientados às representações em computadores; eles obedecem a uma rica coleção de leis matemáticas que tornam possível descrever efetivamente o comportamento de um sistema.

A linguagem Z é definida de maneira a permitir a decomposição de uma especificação em partes menores chamadas esquemas. Os esquemas descrevem os aspectos estáticos e dinâmicos de um sistema. Os aspectos estáticos incluem estados e relações invariantes que são mantidas quando um sistema muda de um estado para outro. As propriedades dinâmicas são: operações possíveis, relações entre entradas e saídas e mudanças de estado.

A linguagem permite que diferentes aspectos de um sistema sejam descritos separadamente e posteriormente relacionados e combinados. Por exemplo, o funcionamento de um sistema, ao receber uma entrada válida, pode ser primeiramente descrito e depois ser estendido para mostrar o tratamento de erros. Os esquemas para definição das extensões não alteram os já construídos.

Esquemas podem ser usados para descrever uma transformação de uma perspectiva para outra, permitindo explicar porque uma especificação abstrata está corretamente implementada por uma outra que contém mais detalhes de um projeto mais concreto. Ao se construir uma seqüência de especificações em que cada uma contém mais detalhes que a anterior, obtém-se uma maior confiabilidade no programa final no que se refere à sua especificação (34).

Por exemplo, podemos considerar a especificação do funcionamento de uma base de dados que tem operações de leitura de uma página, escrita de uma página, *backup* e *restore*. Em um primeiro nível, pode-se descrever essas operações considerando-se que a base de dados tem duas estruturas iguais: uma para trabalho e outra para *backup*; num segundo nível, reescrevem-se todas as operações definindo um esquema em que se tem uma cópia mestre e um registro das mudanças. O segundo nível é uma especialização do primeiro que adiciona uma propriedade de funcionamento em direção à implementação. A linguagem Z permite provar que o funcionamento das operações do segundo nível é matematicamente equivalente ao primeiro.

2.5 Conclusão

Este capítulo apresentou os principais conceitos utilizados no desenvolvimento deste trabalho: BPM, SOA, *Web Services* e modelos de comportamento de software. BPM está associado ao ciclo de vida de um processo de negócios e inclui desde a fase de descoberta até a implantação e o gerenciamento de um processo para melhoria contínua. SOA é uma arquitetura que tem como objetivo alinhar o mundo dos negócios às soluções em tecnologia da informação. Foram descritos os diversos requisitos para se implantar uma arquitetura SOA. *Web Service* é uma tecnologia aceita para a implementação de SOA. A pilha de especificações da arquitetura inclui a camada de composição de *web services*. É nessa camada que a linguagem BPEL é aplicada. Foram apresentados também os principais modelos de descrição de comportamento de software, citados pelo PMG (28) como a base teórica para a modelagem de processos, ou seja, Rede de Petri, linguagem Z e Pi-Calculus.

O próximo capítulo contém uma descrição sobre BPEL e Pi-Calculus complementando os conceitos necessários para o entendimento dos demais capítulos.

Capítulo 3

BPEL e Pi-Calculus

Este capítulo descreve resumidamente BPEL e Pi-Calculus, enfatizando os conceitos que são usados nos capítulos seguintes. O tratamento completo desses assuntos são encontrados em (2) e (23). A seção 3.1 contém: uma descrição geral dos elementos básicos de um processo; a apresentação das atividades, o que inclui construções, operações e comandos da linguagem; e o programa do exemplo da agência de viagem. A seção 3.2 apresenta as operações do Pi-Calculus implementadas na Máquina Virtual, descreve como variáveis lógicas são implementadas em Pi-calculus segundo Milner e, por fim, contém a construção das expressões lógicas básicas que serão utilizadas nos programas Pi-Calculus obtidos dos mapeamentos dos processos BPEL. Essas expressões são uma contribuição deste trabalho.

3.1 BPEL (*Business Process Execution Language*)

BPEL define um modelo e uma gramática para descrever o comportamento de um processo de negócio baseado nas interações entre os processos e seus parceiros. A interação do processo com cada parceiro ocorre através de interfaces *web services* e a estrutura de relacionamento no nível das interfaces é encapsulada em uma ligação entre parceiros (*partner link*). O processo BPEL estabelece a forma como interações múltiplas com esses parceiros são coordenadas para alcançar os objetivos do negócio e a lógica necessária para essa coordenação. BPEL também possui mecanismos para tratar exceções, falhas de processamento e compensações para os casos nos quais o processo não termina no fluxo normal, havendo a necessidade de se desfazer operações (2).

3.1.1 Elementos básicos de um processo

O elemento básico na construção de processos de negócio em BPEL é o *processo* (*process*). Os elementos constituintes de um processo são ilustrados na figura 3.1.

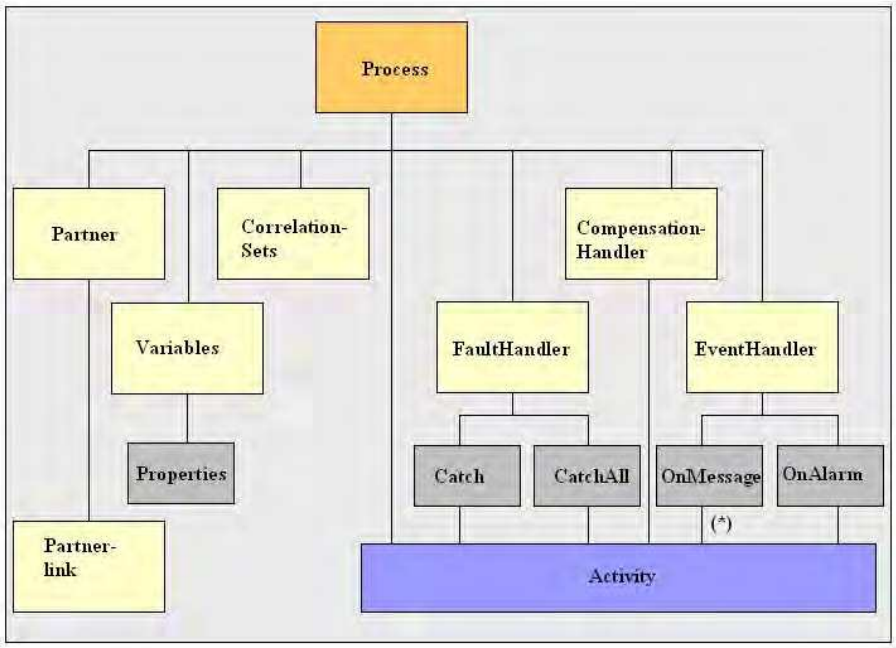

Fig. 3.1: Elementos da estrutura de um processo BPEL.

(*) A estrutura *OnMessage* pode conter o elemento *correlate* (vide descrição das atividades na seção 3.1.2) que aponta para as declarações *CorrelationSets*.

O elemento *ligação entre parceiros* (*partnerLink*) representa uma relação de conversação entre dois processos e especifica os papéis (*roles*) desempenhados pelas entidades que participam do processo através dos tipos de portas (*portTypes*) dos *web services*. Os elementos denominados *parceiros* (*partners*) especificam um conjunto de ligações entre parceiros (*partnerLinks*).

O estado de um processo inclui as mensagens que são trocadas entre *web services* e os dados usados na lógica do processo e na composição das mensagens. *Variáveis* (*variables*) contêm as mensagens e os dados que constituem o estado de um processo.

Propriedades (*properties*) são definições sobre uma mensagem. Para cada propriedade, associa-se um nome a uma parte de uma mensagem através do uso de uma operação *query* da linguagem Xpath (7).

As especificações WSDL definem que uma operação entre *web services* é executada através de uma *porta* (*port*). Entretanto, isso não é suficiente para identificar uma instância de um processo.

Portanto, há a necessidade de um mecanismo em que se possa definir que parte da mensagem identifica a instância de um processo. Por exemplo, um processo que negocia seguros de pessoa física define o CPF do indivíduo como um identificador para a instância. O campo CPF pertence à parte de aplicação do protocolo e não da infra-estrutura (cabeçalho). Para atender a essa funcionalidade foi definido o mecanismo denominado *conjunto de correlações*.

Um *conjunto de correlações* (*correlation sets*) define uma associação de um nome a um conjunto de propriedades e serve como identificador de uma conversação no nível da aplicação, permitindo, assim, que processos remotos identifiquem o objeto da aplicação (a instância do processo).

Atividades (*activities*) são as construções que estão associadas a uma ação ou a uma definição de escopo. As ações definidas em BPEL são descritas na seção seguinte - exemplos dessas ações são: recepção ou envio de uma mensagem, ativação de fluxos paralelos e comando de repetição. Um escopo permite construir um conjunto de atividades aninhadas no processo com suas próprias definições.

Tratadores de falhas (*fault handlers*) são usados para executar atividades quando ocorrem falhas no processamento do fluxo normal do processo. Eles contêm um conjunto de cláusulas *catch* em que cada *catch* é associado a uma falha definida por um *nome* (*fault name*) e contém uma seqüência de atividades.

Tratadores de compensação (*compensation handlers*) fazem o encapsulamento das atividades definidas no escopo de um processo. Em geral, servem para desfazer operações já efetuadas e que não devem permanecer devido ao término anormal do processo. Um tratador de compensação é invocado por uma atividade *compensate* (veja adiante na próxima seção) que referencia o escopo ao qual se aplica o efeito da compensação.

Por fim, na estrutura de um processo tem-se os *tratadores de eventos* (*event handlers*). Tanto o processo como um escopo podem ser associados a um conjunto de tratadores de eventos, ativados concorrentemente quando o evento correspondente ocorre. As ações executadas dentro de um tratador de evento podem ser qualquer tipo de atividade, exceto a execução de um *compensate*. Um evento pode ser a recepção de uma mensagem (*onMessage*) associada a uma operação Web Service ou a um timeout (*onAlarm*).

3.1.2 Atividades de um processo

As atividades[1] definidas para a construção de processos BPEL estão apresentadas a seguir.

Receive [c][2]

[1] Neste texto serão usados também os termos *construção*, *operação* e *comando* para referenciar *atividade*.

[2] A indicação [c] ao lado do nome da atividade significa que estas podem usar a construção correlação (*correlation*) para especificar um conjunto de correlações (*correlation set*).

O processo espera por uma mensagem de um parceiro através de uma porta, com uma operação específica. A mensagem é lida em uma variável definida no processo. Um parâmetro (*createInstance*) define se uma nova instância de processo é criada com a chegada da mensagem ou não.

Reply [c]

O processo envia uma mensagem que é uma resposta a uma outra mensagem lida através de um *receive*. Essa atividade está relacionada a um parceiro e a uma porta. A mensagem é obtida de uma variável do processo. Opcionalmente, a resposta é uma falha quando o parâmetro *faultName* é usado.

Invoke [c]

O processo efetua uma chamada (apenas pedido ou pedido-resposta) a um serviço (*web service*). Esse serviço é definido por um parceiro, através de uma porta, com uma operação específica. A mensagem enviada é obtida de uma variável do processo e a mensagem recebida é armazenada em uma variável. Opcionalmente, pode-se inserir o elemento *catch* para definir o processamento em caso de falha (vide *reply* com opção de falha). Nessa atividade, é possível também definir um tratador de compensação (*compensation handler*).

Assign

Atualiza uma variável com um novo valor.

Throw

Gera uma falha a partir de um processo.

Terminate

Termina imediatamente um processo, sem manipulação de erro e execução de tratadores de compensação.

Wait

Espera até que uma determinada quantidade de tempo tenha passado (*delay*) ou até que um determinado momento ocorra (*deadline*).

Empty

Não executa nada.

Sequence

Essa construção permite definir uma coleção de atividades executadas seqüencialmente.

Switch

Seleciona um e somente um ramo de atividade de uma coleção dependendo dos resultados obtidos da condição estabelecida na construção *case*.

While

Indica a repetição de uma atividade enquanto uma condição associada é verdadeira.

Pick

O processo é bloqueado e passa a esperar por uma mensagem que casa (*match*) com um elemento *onMessage* ou um determinado tempo da construção *onAlarm*. O elemento *onMessage* está associado à chegada de uma mensagem de um parceiro através de uma porta com uma operação específica. A mensagem é armazenada em uma variável do processo. A construção *onAlarm* contém uma definição de tempo.

Flow

Especifica atividades que são executadas em paralelo (concorrentemente). Ligações (*links*)podem ser definidas permitindo efetuar o sincronismo entre atividades paralelas. Essa atividade termina quando todas as outras dentro dela terminam, o que inclui a possibilidade da atividade não ser executada (seção 3.1.3).

Scope

Permite construir um conjunto de atividades aninhadas no processo com suas próprias definições de: *variables, correlation sets, fault handlers, compensation handler e event handlers*. Excetuando-se a definição de *partner* e *partner link*, um escopo contém os mesmos elementos de um processo.

Compensate

É usado para chamar um tratador de compensação em um escopo interno que já tenha sido completado normalmente. Essa atividade somente pode ser chamada a partir de um tratador de falha ou de um tratador de compensação.

3.1.3 Atributos-padrões e elementos-padrões

BPEL tem também construções para atributos-padrões e elementos-padrões (*standard-attributes* e *standard-elements*) que podem ser usados para todas as atividades de um processo.

Atributos-padrões incluem:

- a definição de um nome para a atividade (*name*);
- uma condição (*joinCondition*), que é uma expressão booleana, à qual a decisão da execução da atividade em questão está vinculada;

- uma indicação para suprimir a execução da falha ou não (*bpws:joinFailure*).

Os elementos-padrões são *source* e *target*. Esses elementos, juntamente com os dois últimos atributos padrões (*joinCondition* e *bpws:joinFailure*), são utilizados para o sincronismo entre os fluxos concorrentes. Os elementos *source* e *target* possuem o atributo *linkname* (*nome de ligação*) para definir uma *ligação* (construção *link*) entre as atividades.

Quando uma atividade que tem elementos *target* está "pronta para executar", então os seguintes passos são executados:

1. espera que todas as ligações (dos elementos *target*) tenham valores definidos;

2. avalia a expressão dada pelo atributo *joinCondition*; caso não exista o atributo, a expressão default é um *Ou lógico* de todas as ligações definidas nos elementos *target*; a expressão do atributo *joinCondition* somente pode conter operadores lógicos e a função *bpws:getLinkStatus*. Essa função se aplica às ligações definidas como *linknames* nos elementos *target* de uma dada atividade;

3. se o resultado da expressão é verdadeiro, então a atividade é executada, senão uma falha padrão, denominada *bpws:joinFailure* é gerada (equivalentemente à execução de uma atividade *throw*).

A determinação de valor para uma ligação ocorre em duas situações [3]:

- após a execução de uma atividade que tem essa ligação como um elemento *source*: o valor a ser atribuído ao *linkstatus* é o resultado da avaliação da expressão do atributo *transitionCondition* do elemento *source*; caso esse atributo não seja declarado, o valor *default* é *verdadeiro*;

- após a decisão de que uma atividade contendo essa ligação em um elemento *source* não será executada: nessa situação, existem dois casos, sendo que em ambos, o atributo *linkstatus* recebe o valor *falso*. No primeiro caso, a decisão ocorre após a avaliação do atributo *joinCondition* (esse atributo contém uma expressão booleana dos elementos *target* da atividade). O segundo caso é aplicado às atividades estruturadas. Quando é decidido que uma atividade interna não será executada, atribui-se o valor *falso* aos *linkstatus* de todas as ligações dos elementos *source* dessa atividade. Caso a atividade interna seja também uma atividade estruturada, o efeito é propagado para essas atividades.

[3] Essa explicação considera que o valor do atributo *supressJoinFailure* é sempre *verdadeiro*. A seção 12.5 de (2) contém a descrição completa.

3.1.4 Exemplo: Agência de Viagem

Esta seção introduz o exemplo da Agência de Viagem (19). Alguns detalhes que não são relevantes para os resultados são omitidos.

A descrição do processo inicia-se com algumas definições:

```
<process name="travelAgentExample"
   xmlns = http://schemas.xmlsoap.org/business-process/
   xmlns:tns = "urn:travelAgentExample"
   supressJoinFailure="yes">

   <partners>
      <partner name = "client"/>
      <partner name = "AmericanAirlines"/>
      <partner name = "BritishAirways"/>
      <partner name = "AirCanada"/>
      <partner name = "weatherService"/>
      <partner name = "carRental"/>
   </partners>

   <containers>
      <container name="request"
                 messageType="tns:RequestMessageType />
      <container name="confirmation"
                 messageType="tns:ConfirmationMessageType />
      <container name="forecast"
                 messageType="tns:WeatherMessageType />
      <container name="reply"
                 messageType="tns:ReplyMessageType />
      <container name="rentalConfirmation"
                 messageType="tns:ConfirmationMessageType />
   </containers>
```

Essas definições consistem de:

- descrições iniciais de um processo;
- lista de parceiros do processo;
- relação das variáveis. A tag *containers* corresponde à tag *variables* da versão 1.1. Essas variáveis contêm as mensagens trocadas entre os *web services*.

A seguir, é apresentada a lógica do processo:

```
<sequence>
 <receive partner="client" portType="clientPT"
          operation="makeTravelArrangements"
        container="request" />
 <flow>
   <links>
     <link name="travel-canada" />
     <link name="travel-us" />
     <link name="rent-to-assign" />
   </links>
   <switch>
     <case condition="bpws:getContainerData ('request','
           'destination-country' = 'Canada'")
        <invoke partner="AirCanada" portType="reservationsPT"
                operation="makeReservation"
              inputContainer="request" outputContainer="confirmation">
           <source linkName="travel-canada" />
        </invoke>
     </case>
     <case condition="bpws:getContainerData ('request','
           'destination-country' = 'US'")
        <invoke partner="AmericanAirlines" portType="reservationsPT"
                operation="makeReservation"
              inputContainer="request" outputContainer="confirmation">
           <source linkName="travel-us"
                   transitionCondition="bpws:getContainerData
                     ('request','destination-city') != 'New York'" />
        </invoke>
     </case>
     <otherwise>
        <invoke partner="BritishAirways" portType="reservationsPT"
                operation="makeReservation"
              inputContainer="request" outputContainer="confirmation">
        </invoke>
     </otherwise>
   </switch>
   <invoke partner="weatherService" portType="weatherPT"
           operation="getForecast"
         inputContainer="request" outputContainer="forecast">
   </invoke>
   <invoke partner="carRental" portType="carRentalPT"
           operation="rent"
```

```
          inputContainer="request"
            outputContainer="rentalConfirmation"
            joinCondition="bpws:getLinkStatus ('travel-canada') or
                           bpws:getLinkStatus ('travel-us')">

       <target linkName="travel-canada" />
       <target linkName="travel-us" />
       <source linkName="rent-to-assign" />
    </invoke>
    <assign>
      <target linkName="rent-to-assign" />
      <copy>
        <from container="rentalConfirmation" />
        <to container="reply"part="part2" />
      </copy>
    </assign>
  </flow>
  <assign>
    <copy>
      <from container="confirmation" />
      <to container="reply"part="part1" />
    </copy>
    <copy>
      <from container="forecast" />
      <to container="reply"part="part3" />
    </copy>
  </assign>
  <reply partner="client" portType="clientPT"
         operation="makeTravelArrangements"
       container="reply" />
 </sequence>
</process>
```

O processo é iniciado executando-se uma atividade composta *sequence*, cuja primeira atividade interna é a espera por uma mensagem de um cliente para a execução do serviço. Ao receber a mensagem, esta é armazenada na variável `request`. Em seguida, é disparada uma atividade *flow* que executa em paralelo os quatro comandos descritos abaixo:

1. um *switch* para decidir a companhia aérea na qual a reserva será feita e, em seguida, efetuar a reserva;

2. um *invoke* para o serviço de tempo;

3. Um *invoke* para o serviço de aluguel de carro;

4. um *assign* para mover o campo de confirmação da reserva do aluguel do carro para a resposta do processo ao cliente.

Cada um dos comandos são explicados a seguir:

1. o *switch* tem três ramificações.

   A primeira é executada se a mensagem contida na variável `request` tem o atributo que indica que o destino é "Canada"; no caso verdadeiro, um *web service* para fazer a reserva é chamado através de um *invoke*. As variáveis de entrada e saída dessa operação são, respectivamente, `request` e `confirmation`. Esta última será usada para dar a resposta ao cliente (parte 1 da variável de resposta).

   O segundo ramo do *switch* é análogo ao anterior, sendo o destino "US". Nesse caso, a declaração *source* é condicional. O atributo *linkStatus* será *verdadeiro* se a cidade não for "New York". No último ramo, assume-se por *default* que o destino é coberto pela companhia aérea "BritishAirways". Cabe observar que não há a cláusula *source* nesse ramo;

2. a chamada para o serviço do tempo não está condicionada a nenhuma ligação e é sempre executada dentro da atividade *flow*, em paralelo com as demais atividades;

3. a condição para a execução do serviço de aluguel de carro é a expressão *Ou lógico* (expressão default do atributo *joinCondition*) das ligações `travel-canada` e `travel-us`. Em termos práticos, o aluguel do carro será efetuado se a reserva tiver sido feita para o Canadá ou para os Estados Unidos, exceto "NewYork";

4. a atribuição (*assign*) para mover o campo de confirmação da reserva do aluguel do carro está condicionada ao *linkStatus* da ligação `rental-to-assign`, que recebe o valor *verdadeiro* se e somente se o serviço de aluguel do carro foi executado.

Depois que a atividade *flow* termina (quando todas atividades internas terminam), o fluxo segue seqüencialmente fazendo uma atribuição (comando *assign*) das partes 1 e 3 da confirmação para a resposta e, finalmente, enviando a resposta para o cliente através de um *reply*.

3.1.5 Arquitetura de uma implementação BPEL

Esta seção contém uma breve descrição de uma arquitetura de implementação de um sistema de execução BPEL. O sistema escolhido foi o BPWS4J (9).

O sistema possui três componentes principais:

- *Runtime*: contém as informações sobre os processos;
- Contêiner: provê um ambiente de implantação e execução para os processos;
- Interpretador: executa as atividades de um processo BPEL dentro do ambiente do *Contêiner*.

O componente *Runtime* contém todas informações sobre a definição do processo. Esse módulo é usado pelo Contêiner na implantação e instanciação de um processo. Ele é também usado pelo Interpretador para controlar a execução do processo.

Os principais serviços do Contêiner são: implantação de um processo, gerenciamento do ciclo de vida, roteamento de mensagens e suporte à chamada de serviço.

Em BPEL, as especificações das interações dos processos com os parceiros são feitas em um nível abstrato (tipos de portas em WSDL). No momento da implantação de um processo, há a necessidade de se definir os endereços dos serviços dos parceiros ou a estratégia que deve ser seguida para a descoberta desses endereços. Pelo menos os endereços dos parceiros, cujas interações são iniciadas pelo processo, devem ser resolvidos no instante da implantação do processo.

Um processo BPEL é criado quando ele recebe uma mensagem ou um alarme da atividade *pick* e é destruído quando a última atividade de uma instância completa sua execução. No gerenciamento do ciclo de vida dos processos, o Contêiner decide, na recepção de uma mensagem, se esta é para ser roteada para algum processo existente ou se é para criar um novo processo. O algoritmo que faz essa decisão leva em consideração: a operação contida na mensagem; as definições dos processos implantados para determinar se a operação é para instanciar um processo; e os valores possíveis dos *correlation sets* para verificar para qual processo a mensagem deve ser roteada.

O Interpretador é implementado em uma arquitetura controlada por eventos e executa uma instância de um processo usando o Contêiner para qualquer interação com o ambiente externo. A estrutura do componente praticamente corresponde à estrutura do processo, sendo que a semântica de execução do BPEL está codificada nas atividades. Atividades são classificadas entre simples e complexas. As atividades complexas, incluindo escopo, controlam as atividades que elas contêm. Cada instância é executada como uma *thread*, com o paralelismo implementado pela arquitetura de disparos de eventos.

Uma atividade é ativada quando ela recebe o controle da atividade que a contém e se torna "habilitada para execução" quando os *status* de todas ligações de entrada (*incoming links*) são conhecidos e a condição avaliada no atributo *join condition* é *verdadeira*.

Os eventos que ocorrem no Interpretador são classificados em três níveis: processo, escopo e atividade complexa. Somente o nível de processo troca eventos com o Contêiner. Os eventos desse nível são: operações de envio e recepção de mensagens com *web services*, operações de atribuição que usam o módulo de manipulação Xpath e notificação de *status*. Uma fila de mensagens é usada para

guardar as mensagens entrantes até que sejam roteadas para as atividades dentro de uma instância. No nível de escopo, dois eventos são manipulados: eventos de falhas e de ligação. Finalmente, no nível de atividade complexa, os eventos são para o controle de execução.

Quando ocorre uma falha na execução em uma atividade, essa dispara um evento que vai para o nível do escopo que a contém. O processamento desse evento desabilita todas atividades contidas no escopo - que inclui estabelecer os valores iniciais para as variáveis de estado e o valor *falso* para todas as ligações de saída (*outgoing links*) - e ativa a primeira atividade do Tratador de Falhas (*Fault Handler*).

Qualquer atividade que tenha o atributo *suppressJoinFailure* com valor *true* é compilada de modo a gerar um escopo com um Tratador de Falhas vazio especificamente para essa falha (*joinFailure*). Essa tradução é necessária para suportar a combinação da modelagem estruturada por grafo do BPEL e a eliminação do caminho morto (seção 4.4).

3.2 Pi-Calculus

Pi-Calculus é uma álgebra para descrever sistemas interativos e concorrentes. Programas em Pi-Calculus são desmembrados em processos paralelos que se comunicam através de canais usando a semântica do *rendez-vous* síncrono. Mensagens trocadas entre processos são nomes, que por sua vez podem representar processos, canais ou outras abstrações. Processos podem chamar outros processos recursivamente ou não. Esses são os elementos básicos de Pi-Calculus.

Essa seção descreve informalmente como esses elementos básicos funcionam, introduz variáveis lógicas e desenvolve algumas expressões lógicas usadas nos capítulos subseqüentes. A construção dessas expressões, a partir das definições de Milner (23), é uma contribuição deste trabalho.

Rendez-vous Síncrono

O *rendez-vous* síncrono, entre dois processos que desejam comunicar-se através de um canal, funciona da seguinte maneira:

- um processo sinaliza sua intenção de transmitir uma mensagem em um canal e então espera;
- o outro processo sinaliza que está pronto para receber uma mensagem através do mesmo canal e espera;
- dado que os dois processos estão prontos para se comunicarem, então ocorre a transferência da mensagem entre os dois programas que continuam a execução a partir desse ponto.

3.2.1 Elementos básicos

Um programa Pi-Calculus é decomposto em processos paralelos que interagem sobre *canais* (*channels*). As mensagens que são transmitidas e recebidas nos canais são *nomes* (*names*) para o Pi-Calculus. *Nomes* podem ser *livres* (*free*) ou *restritos* (*bound*). Nomes livres têm significado global enquanto os restritos têm escopo local, conforme é visto a seguir.

As operações *leitura* (*read*), *escrita* (*write*) e *paralelo* (*parallel*) são explicadas através do exemplo a seguir.

```
~u<x>.P | u(y).Q
```

A expressão acima representa dois processos executando em paralelo, em que o primeiro, `~u<x>.P`, está pronto para enviar um dado `x` através do canal `u` e, em seguida, executar `P`. Em paralelo, o segundo processo, `u(y).Q`, está pronto para receber um dado do canal `u` em `y`, e continuar executando `Q`. O nome `y` é um nome restrito, funciona como um argumento formal, e é local a `Q`. Quando acontece o *rendez-vous*, ou seja, a troca de mensagem e conseqüentemente o desbloqueio dos processos envolvidos, ocorre uma redução representada da seguinte maneira:

```
~u<x>.P | u(y).Q --> P | Q{x/y}
```

A expressão do lado direito corresponde ao estado do programa após a reação e significa que um processo está executando `P` em paralelo com um processo executando `Q`, com todos os valores formais `y` substituídos por `x`.

Para criar um novo canal e atribuir a um nome restrito de um programa P, usa-se o operador `new`, como no exemplo dado a seguir:

```
(new x)P
```

O canal criado é único e é atribuído ao nome restrito `x`, que tem escopo limitado a `P`.

Termos podem ser replicados através da operação *replica* (*replicate*). Isso permite programas terem comportamento infinito. Um termo replicado é equivalente a muitas cópias de si. O símbolo `!`, precedendo um termo, é usado para expressar a replicação. Assim, por exemplo:

```
~u<x> |!u(y).~y -->

~u<x> | u(y).~y | !u(y).~y -->

~x | !u(y).~y
```

Na primeira passagem, o termo replicado é copiado; na segunda, ocorre o *rendez-vous* em `u`, tendo como conseqüência a substituição do nome local `y` pelo nome `x` [4].

A outra operação definida em Pi-Calculus é denominada *soma* (*summation* ou *sum*). Uma operação de soma possui dois ou mais termos. Essa operação é bloqueada até que ocorra um *rendez-vous* em um dos termos. Quando isso ocorre, os demais termos são cancelados. O exemplo a seguir ilustra o uso da soma:

```
~u<z> | (u(x).P + v(y).Q) --> P {z/x}
```

A redução ocorre em função do *rendez-vous* em `u` no primeiro termo da soma. O segundo termo é cancelado.

Até agora, foram descritas as operações que transferem somente um nome através dos canais. Porém, são desejáveis, e estão definidas em Pi-Calculus, operações poliádicas, em que num *rendez-vous* ocorre a transferência de um vetor de nomes. A representação dessa operação é exemplificada abaixo.

```
x<y1,..,yn>.P | ~x<z1,..,zn)>.Q
```

onde todos `yi` e `zi` são distintos. A reação da expressão acima produziria:

```
P{(z1,..,zn)/(y1,..,yn)}|Q
```

o que significa dizer que a execução de `P` em paralelo com `Q` tem os nomes `yi` substituídos por `zi` para `i=1,..,n`.

Outra construção do Pi-Calculus relevante é a definição recursiva de processos com parâmetros. Essa definição tem a forma:

```
A (x1,..,xn) ==> QA,
```

onde `A` é o identificador de um processo e os nomes `x1,..,xn` são distintos e restritos a `A`. `QA` é uma expressão que pode conter chamadas para `A` e outros processos parametrizáveis.

3.2.2 Variáveis lógicas

Os valores *True* e *False* em Pi-Calculus são processos que têm as seguintes definições[5]:

```
True (l) ==> l(t,f).~t
False (l) ==> l(t,f).~f
```

[4] Na implementação da Máquina Virtual, a replicação do termo é feita sob demanda, ou seja, no momento em que ocorre uma redução no termo.

[5] Para maiores detalhes, veja o item 10.3 (*Data revisted*) do livro do Milner (23).

Na primeira declaração, a definição de um valor `True` corresponde à ativação de um processo que tem como parâmetro um canal (`l`), que é a localização onde os canais que representam os valores *verdadeiro* (`t`) e *falso* (`f`) são escritos. O processo `True` lê os valores enviados para essa localização e em seguida escreve no canal que representa o valor *verdadeiro* (`t`).

Uma aplicação do uso dessas variáveis é o processo `Cond` definido da seguinte forma:

```
Cond (P,Q)(l) ==> (new t,f)~l<t,f>.(t.P + f.Q)
```

Esse processo recebe como parâmetros dois outros processos `P` e `Q` e uma localização `l`. Caso seja ativado um processo `True` em `l`, então o resultado da expressão é `P`. Caso `False` seja ativado em `l`, então o resultado é `Q`. A seguir, a execução do processo `Cond` é detalhada passo-a-passo:

- aloca dois canais locais e armazena-os nos nomes restritos (*bound names*) `t` e `f`;
- prepara para escrever, no canal dado pelo parâmetro `l`, os valores dos canais locais alocados [que significam os valores *verdadeiro* (`t`) e *falso* (`f`) nesse contexto]. Espera ocorrer o *rendez-vous*;
- o *rendez-vous* ocorre quando um outro processo (*True* ou *False* ou que desempenha função equivalente) lê no canal passado como parâmetro em `l`;
- após o *rendez-vous*, é executada uma soma, na qual há o bloqueio para ocorrência do *rendez-vous* no canal que representa o valor *verdadeiro* (t) ou *falso* (f);
- dependendo do termo em que ocorre o *rendez-vous*, um dos processos passados como parâmetro (`P` ou `Q`) é executado.

A seguir, exemplifica-se de maneira algébrica como um programa - correspondente à ativação do processo `Cond` em paralelo com um processo `True` - reage para produzir `P` .

```
True <l> | Cond <P,Q><l> ->* P
```

Substituindo os processos pelas suas definições:

```
l(t,f).~t | (new t,f)~l<t,f>.(t.P + f.Q)
```

rendez-vous em `l`:

```
~t | (t.P + f.Q)
```

rendez-vous em `t`:

```
P
```

3.2.3 Expressões lógicas

Como é visto no capítulo 4, há a necessidade de se ter expressões lógicas Pi-Calculus para a manipulação do sincronismo entre atividades dos processos BPEL. As expressões são construídas à medida que são necessárias. Nesta seção, são apresentadas algumas das expressões usadas nos exemplos do texto.

Função *And* de duas variáveis

```
And (a,b,z) ==> new(t1,f1,t2,f2).z(t,f).(~a<t1,f1>|~b<t2,f2>|
                (t1.(t2.~t+f2.~f)+ f1.(t2.~f+f2.~f))
```

Os parâmetros `a` e `b` são localizações nas quais serão ativados processos que resultem no equivalente a um processo *True* ou *False*. O parâmetro `z` corresponde ao resultado, ou seja, o cliente que estiver usando a função faz uma escrita em `z` dos valores que representam *verdadeiro* e *falso* da expressão.

Caso os processos que foram ativados em `a` e `b` sejam equivalentes ao processo *True*, este processo `And` fará a escrita no valor recebido em `z` que representa *verdadeiro*.

Caso um dos processos que foram ativados em `a` e `b` sejam equivalentes ao processo *False*, este processo `And` fará a escrita no valor recebido em `z` que representa *falso*. A título de exemplo, verifica-se que a expressão abaixo produz como resultado a execução de `P`:

```
new (x,y,z) . (True<x> | True<y> | And<x,y,z> | Cond <P,Q><z>)
```

Após alocar canais para os nomes restritos `x`, `y` e `z`, todos os quatro processos são ativados em paralelo produzindo a seguinte expressão:

```
(x(t,f).~t) | (y(t,f).~t) |
(new (t1,f1,t2,f2). z(t,f). (~x<t1,f1> |~y<t2,f2> |
(t1.(t2.~t+f2.~f)+ f1.(t2.~f+f2.~f))) |
(new(t,f)~z<t,f>.(t.P + f.Q))
```

O apêndice E contém a verificação da função lógica através do uso da Máquina Virtual.

Função *Or* de duas variáveis

```
Or (a,b,z)  ==> new (t1,f1,t2,f2).z(t,f).(~a<t1,f1>|~b<t2,f2>|
                (t1.(t2.~t+f2.~t)+ f1.(t2.~t+f2.~f))
```

O funcionamento é análogo ao da função anterior.

Função *Not*

```
Not (a,z)   ==> new (t1,f1).z(t,f).(~a<t1,f1>.(t1.f+f1.t))
```

A função *Not* recebe como parâmetro `a`, no qual será lançado o processo equivalente a um *True* ou *False*, e o parâmetro `z` é o resultado. O cliente dessa função escreve em `z` os valores que representam *verdadeiro* e *falso* (canais `t` e `f`, respectivamente). Se o processo que é lançado em `a` é equivalente ao *True*, então o resultado é o valor que representa *falso*; caso contrário, o processo lançado é equivalente ao *False* e o resultado é o valor que representa *verdadeiro*.

3.3 Conclusão

Este capítulo descreveu os conceitos de BPEL e Pi-Calculus usados na seqüência deste trabalho. Adicionalmente, foi apresentado o desenvolvimento das expressões lógicas Pi-Calculus que serão usadas para a tradução do processo BPEL para Pi-Calculus. O próximo capítulo contém as regras para essa tradução - essas regras são definidas a partir da consideração de que se deseja gerar todos caminhos de execução do processo BPEL para a análise do fluxo de controle e das interações com os *web services*.

Capítulo 4

Mapeamento BPEL para Pi-Calculus

Este capítulo descreve as regras de conversão de um processo BPEL para um programa Pi-Calculus. Essas regras são particulares e aplicáveis ao objetivo desse trabalho, que é a análise e verificação de processos BPEL através da Máquina Virtual Pi-Calculus (MVP). Outras variantes dessas regras podem ser desenvolvidas em função de objetivos diferentes.

As seções 4.1 e **??** apresentam as considerações básicas necessárias para o desenvolvimento do mapeamento. Essas considerações são relativas à classificação dos tipos de canais Pi-Calculus e das restrições de escopo estabelecidas em relação à linguagem BPEL. As regras do mapeamento estão contidas em três seções: a primeira (4.2), contém as regras para cada atividade, desconsiderando atributos-padrões e elementos-padrões, que são tratados na seção 4.3; a última seção de mapeamento (4.4) estabelece as regras para a não-execução de atividades internas às atividades compostas e para a eliminação de caminhos mortos (*dead path elimination*).

Por fim, a seção 4.5 contém a expressão Pi-Calculus resultante da tradução do processo BPEL, para a Agência de Viagem, apresentado no capítulo anterior.

4.1 Considerações sobre canais Pi-Calculus e sobre o mapeamento

No processo de construção da expressão Pi-Calculus traduzida do processo BPEL, há a necessidade de se diferenciar o uso dos canais Pi-Calculus. Esses canais são classificados em *internos* e *externos*. Um canal interno é aquele em que o *rendez-vous* ocorre dentro da expressão, ou seja, as operações de leitura e escrita são realizadas através do processo de redução, sem a necessidade de um ambiente externo. Canais externos são os que têm uma operação realizada na expressão e a outra por algum ambiente externo.

Os canais internos são usados para concorrência e controle do fluxo das construções do BPEL, tais como: *source*, *target*, *switch*, *pick*, etc. Os canais externos são aplicáveis às interações do processos

com os *web services* e para as atividades do processo BPEL que são relevantes para a análise do processo.

A diferenciação de tipos de canais é explicitada através dos nomes. Os nomes dos canais externos iniciam-se com a substring `//`. Os canais externos que representam os *web services* têm como prefixo a substring `//ws/` e os associados às atividades internas do BPEL que são relevantes para a análise iniciam-se com `//act/`.

Para atingir o objetivo proposto neste trabalho, algumas considerações sobre o mapeamento foram tomadas. Valores específicos de variáveis são desconsiderados. Quando um valor de uma variável influenciar na criação de um processo ou na alteração de um fluxo, são geradas todas as possibilidades dentro do domínio definido. Valores de tempo também não são considerados, apenas os eventos das temporizações.

Por simplificação, não serão definidos mapeamento para escopo, tratamento de falhas, eventos e compensação. Em um trabalho futuro, essas restrições deverão ser tratadas.

As construções *CorrelationSets* não são mapeadas: elas só fazem sentido para instanciação dos processos e não para a análise de concorrência e fluxo de controle.

4.2 Regras de mapeamento para as atividades

Cada atividade em BPEL tem opcionalmente atributos-padrões e elementos-padrões (*standard-attributes* e *standard-elements*). Eles serão tratados na seção 4.3, pois as regras de tradução para esses atributos e elementos são comuns a todas as atividades.

Receive, Reply, Invoke

Essas atividades são mapeadas para leitura (*read*) e escrita (*write*) em um canal externo. O *receive* é traduzido para uma leitura, enquanto que o *reply* e o *invoke* são transformados em escrita. A rigor, uma atividade *invoke* que tenha resposta deveria ser traduzida para uma escrita e uma leitura em seqüência; porém, como a execução do *invoke* é indivisível em relação ao controle do fluxo e à concorrência, simplifica-se sua tradução para uma única operação.

A execução de uma operação de leitura ou de escrita está associada a um *rendez-vous*. Na Máquina Virtual, essas operações em canais externos pressupõem a existência de uma entidade externa (que, por exemplo, corresponde a um *web service* na rede), que faz a operação que completa o *rendez-vous*. Como é visto no capítulo 5, um canal interno atinge o estado "pronto para *rendez-vous*"quando existe uma operação de leitura e uma operação de escrita. O canal externo atinge esse estado com uma operação - o que equivale a dizer que, hipoteticamente, existe uma entidade externa efetuando a operação dual nesse canal.

A regra para a formação do nome do canal é dada pela concatenação das *substrings*, separadas por "/", contendo mnemônicos para a atividade e os parâmetros WSDL da operação. Esses parâmetros são: *partnerLink*, *portType* e *operation*.

Exemplo

```
<invoke partnetLink="invoicing"
       portType="lns:computePricePT",
       operation="initiatePriceCalculation"
       inputVariables="PO"
       outputVariables="price1">
</invoke>
```

A atividade acima é traduzida para a escrita em um canal que representa um *web service* (os nomes dos parâmetros foram compactados para *strings* menores):

```
~//ws/inv/inv/compriPT/iniprical
```

Switch

A sintaxe BPEL para o *switch* é:

```
<switch standard-attributes>
   standard-elements
   <case condition="bool-expr">+
       activity
   </case>
   <otherwise>?
       activity
   </otherwise>
</switch>
```

A construção *switch* contém uma lista de casos (*cases*) e opcionalmente uma declaração *otherwise*. Somente um dos ramos é executado. O termo `activity` da sintaxe designa qualquer atividade do BPEL, podendo esta ser simples ou estruturada.

A tradução para Pi-Calculus tem a seguinte forma:

```
1    new(sw1,sw2,...,swn).
2    ((~sw1+~sw2+...+~swn) |
3     (sw1."tradução da atividade 1" +
4      sw2."tradução da atividade 2" +
5      ... +
6      swn."tradução da atividade n" ))
```

A execução do trecho do programa correspondente à tradução de uma atividade *switch* do BPEL é iniciada pela alocação de um canal para cada ramo do switch (linha 1). Em seguida, dois processos paralelos são ativados, um processo `p1` que corresponde à linha 2 da expressão e um processo `p2` correspondente às linhas 3 a 6. Nesse instante, `p1` está preparado para escrever em um dos canais enquanto que `p2` está preparado para ler. Na semântica do Pi-Calculus, das `n` possibilidades de *rendez-vous*, uma é escolhida e o resultado é a execução de uma única atividade dentro da computação - as demais são canceladas. Como a MVP gera todas as computações possíveis, o resultado final para a análise do programa conterá todos os casos possíveis em um *switch*, separadamente em cada contexto.

Se não existir a declaração *otherwise*, então a expressão que segue `swn` é `0`.

While

A sintaxe para o comando *while* é dada a seguir:

```
<while condition="bool-expr" standard-attributes>
   standard-elements
   activity
</while>
```

A tradução produz uma expressão Pi-Calculus como:

```
1    new(t,f).
2   ((~t+~f)|
3    (t."tradução da atividade" +
4     f.0))
```

Para efeito da análise de controle do fluxo e de concorrência, não há necessidade de simular o efeito repetitivo do *while*.

A expressão Pi-Calculus gerada pela tradução de um *while* corresponde à de um *switch* com um *case* e sem a cláusula *otherwise* - substituindo `t` e `f` por `sw1` e `sw2`, respectivamente.

Pick

A sintaxe para a construção da atividade *pick* é:

```
<pick createInstance="yes|no"? standard-attributes>
   standard-elements
   <onMessage partnerLink="ncname" portType="qname"
              operation="ncname" variable="ncname"?>+
      <correlations>?
```

```
            <correlation set="ncname"initiate=yes|no"?>+
        </correlations>
        activity
    </onMessage>
    <onAlarm (for="duration-expr" | until="deadline-expr")>*
        activity
    </onAlarm>
</pick>
```

O trecho de programa Pi-Calculus traduzido possui a forma:

```
1    new(pi1,pi2,...,pim,...,pin).
2    ((~pi1+~pi2+...+~pim+...+~pin) |
3    (pi1.//ws/service1."tradução da atividade 1" +
4     pi2.//ws/service2."tradução da atividade 2" +
5     ... +
6     pim.//ws/servicem."tradução da atividade m" +
7     ... +
8     pin."tradução da atividade n"))
```

Cada declaração *onMessage* (supondo `m>0`) é uma possibilidade de recepção de uma mensagem associada a um serviço, representada pela leitura na porta `//ws/service-i` ; ao receber uma mensagem, o processo executa uma atividade `i(1<=i<=m)` . Cada declaração *onAlarm* (supondo n>m) é uma possibilidade da ocorrência de um evento de temporização; ao ocorrer um desses eventos, uma atividade é executada `("tradução da atividade j" (m<j<=n))`.

A execução da expressão Pi-Calculus pela MVP inicia-se com a alocação de um canal para cada ramo do *pick* (linha 1). Em seguida, dois processos paralelos são ativados, um processo `p1` que corresponde à linha 2 da expressão e um processo `p2` correspondente às linhas 3 a 8. Nesse instante, `p1` está preparado para escrever em um dos canais enquanto `p2` está preparado para ler. Das `n` possibilidades de *rendez-vous*, uma é escolhida e o resultado é a execução de uma operação de *web service* seguida de uma atividade (os primeiros `m` canais) ou somente de uma atividade (os canais seguintes) dentro da computação. Como a MVP gera todas as computações possíveis, o resultado final para a análise do programa conterá todos os casos possíveis em um *pick*, separadamente em cada contexto.

Sequence

A atividade *sequence* tem a seguinte sintaxe:

```
<sequence standard-attributes>
   standard-elements
```

```
    activity 1
    activity 2 ...
    activity n
</sequence>
```

A tradução para Pi-Calculus usa o separador de *seqüência* para cada atividade, como mostrado a seguir.

```
("tradução da atividade 1" . "tradução da atividade 2" . ...
 . "tradução da atividade n")
```

Flow

Da mesma maneira que a tradução da atividade *sequence*, esta atividade é convertida diretamente para uma construção do Pi-Calculus, que é o operador *paralelizar* (|).

A atividade *flow* tem a seguinte sintaxe:

```
<flow standard-attributes>
    standard-elements
    activity 1
    activity 2
    activity n
</flow>
```

A tradução para Pi-Calculus usa o símbolo de *paralelização* para cada atividade traduzida, como apresentado abaixo:

```
(("tradução da atividade 1" | "tradução da atividade 2" | ...
 | "tradução da atividade n")
```

Terminate

Para a tradução dessa atividade, foi adicionada à Máquina Virtual uma operação denominada *exit*, cuja semântica consiste em terminar um programa normalmente, sem deixar nenhuma pendência. Em termos de Pi-Calculus, o resultado da execução dessa operação é o valor 0.

Assign, Wait, Scope, Empty, Thrown, Compensate

Considerando as simplicações feitas neste trabalho - em que escopo, tratamento de falhas, compensações e eventos não são relevados -, essas atividades não precisam ter tradução ou, opcionalmente, quando for relevante para análise, podem ser convertidas para a leitura ou escrita em um canal externo da Máquina Virtual que representa uma atividade interna do BPEL (`//act`).

4.3 Atributos-padrões e elementos-padrões

Os atributos-padrões e elementos-padrões são aplicáveis a qualquer atividade. Os atributos-padrões têm a seguinte sintaxe:

```
name="ncname"?
joinCondition="bool-expr"? supressJoinFailure="yes|no"?>
```

Na tradução para Pi-Calculus, somente é considerado o atributo *joinCondition*, pois o atributo *name* não possui nenhuma semântica e o atributo *supressJoinFailure* é sempre igual a *yes*, visto que tratamento de falhas está fora do escopo deste trabalho.

A expressão lógica do atributo *joinCondition* contém somente operadores lógicos sobre os atributos *linkStatus* das ligações definidas como *target* nos elementos-padrões de uma atividade. A atividade somente é executada se o resultado dessa expressão for *verdadeiro*.

Os elementos padrões são definidos pela seguinte sintaxe:

```
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<target linkName="ncname"/>*
```

Os elementos *target* correspondem às declarações das ligações que fazem parte da expressão do atributo *joinCondition*. Os elementos *source* são os que definem os valores dos *linkStatus* das ligações. Ao término da execução de uma atividade (ou na conclusão da avaliação que a atividade não será executada) que possui elementos *source*, atribuem-se os valores dos *linkStatus*, conforme explicado na seção 3.1.3.

As regras para as construções Pi-Calculus que produzem a semântica que o BPEL define para o sincronismo de atividades, em função dos atributos e elementos-padrões, são:

1. define-se um canal interno para cada *link* existente;

2. na saída de uma atividade (após a declaração), se existe mais de um elemento *source* então uma construção paralela é gerada. Os membros dessa construção são expressões que correspondem a cada elemento *source*, produzidas pelas seguintes regras:

2.1. a atividade é executada e não existe o atributo *transitionCondition*: gera-se a ativação de um processo *True*. Assim, por exemplo, se o *linkName* é `ready-to-start` , gera-se:

```
True(ready-to-start)
```

2.2. a atividade é executada e existe o atributo *transitionCondition*: a expressão Pi-Calculus gera todos resultados possíveis através da operação *soma* contendo as ativações dos processos *True* e *False*. Para o linkName do exemplo anterior, a expressão gerada é:

```
(True(ready-to-start) + False (ready-to-start))
```

2.3. a atividade não é executada: gera-se em paralelo ativações do processo *False* para cada elemento *source*. Se a atividade é composta, então essa regra é aplicada recursivamente para todas as atividades internas;

3. na entrada de uma atividade (antes da declaração), é necessário esperar que todas as ligações definidas como *target* tenham valores definidos, para em seguida aplicar a expressão do atributo *joinCondition*. Em BPEL, essa expressão possui somente variáveis que são os *linkStatus*. Como cada ligação BPEL é traduzida para um canal Pi-Calculus, o código gerado é uma expressão lógica Pi-Calculus correspondente. Por exemplo, se uma atividade tem dois elementos *target* cujos *linkNames* são `payment-to-ship` e `schedule-to-ship` e a expressão do atributo *joinCondition* é um *And* entre essas duas ligações, a expressão Pi-Calculus gerada é:

```
1   new (true,false,jc).
2   ~jc<true,false> |
3   And(payment-to-ship,schedule-to-ship,jc)|
4   (true.//act/activity."gera sources conforme regras 2.1 ou 2.2" +
5   false.0."gera sources conforme regra 2.3")
```

A execução do trecho do código acima inicia-se com a alocação dos canais que definem os valores *verdadeiro* (`true`) e *falso* (`false`) e um canal para o resultado (`jc`) da expressão boolena definida pelo atributo *joinCondition* (linha 1). Em seguida, três subprocessos paralelos são ativados: p1 (linha 2), p2 (linha 3) e p3 (linhas 4 e 5). O subprocesso p1 envia os canais de condição para o canal `jc` e p2 ativa o processo da expressão (`And`). Caso o resultado da expressão seja *verdadeiro*, há uma escrita (*rendez-vous*) no canal `true` de p3 e a atividade `//act/activity` é executada; em seguida, executa-se o trecho para as ligações definidas nos elementos *sources* (linha 4). Se o resultado da expressão é *falso*, a escrita (*rendez-vous*) ocorre no canal `false`. Nesse caso, executa-se somente o trecho para as ligações definidas nos elementos *sources* (linha 5).

4.4 Atividades estruturadas e Eliminação de Caminho Morto (*Dead Path Elimination*)

Se durante a execução de uma atividade estruturada S existe uma atividade interna que não será executada, então para os *linkStatus* de todas as ligações dos elementos *source* desta atividade interna são atribuídos o valor *falso* (2). Essa regra da linguagem BPEL é traduzida para uma expressão Pi-Calculus que contém chamadas em paralelo ao processo *False* para os canais correspondentes às ligações.

O mecanismo de eliminação de caminho morto é descrito da seguinte maneira: a semântica de execução de uma atividade - cujo atributo *supressJoinFailure* tem o valor `"yes"` e a avaliação do atributo *joinCondition* é *falso* - é não executar a atividade e atribuir o valor *falso* para os *linkStatus* de todas as ligações dos elementos *source* da atividade. A regra Pi-Calculus para atender esse requisito é a definida no passo 2.3 da seção 4.3, isto é, a tradução gera processos *False* em paralelo para os canais correspondentes às ligações dos elementos *source*.

Assim, por exemplo, numa atividade estruturada *switch*, apuram-se as ligações dos elementos *target* de todos os ramos. Seja LS o conjunto dessas ligações. A tradução de um ramo do switch gera, além do código correspondente à atividade daquele ramo, chamadas a processos *True* e *False* para as ligações dos elementos *source* e também geram, em paralelo, processos *False* para as ligações de LS que não estão nos elementos *source* do ramo.

O exemplo apresentado a seguir ilustra a aplicação desses conceitos:

```
 <switch>
   <case condition="bpws:getConteinerData (...) = 'Canada'>
     <invoke partner="AirCanada" ... >
       <source linkname="travel-canada" />
     </invoke>
   </case>
   <case condition="bpws:getConteinerData (...) = 'US'
     <invoke partner="AirCanada" ... >
       <source linkname="travel-US"
             transitionCondition= "bpws:getContainerData (...) != 'New York' />
     </invoke>
   </case>
   <otherwise>
     invoke partner="BritishAirwways"
   <otherwise>
</switch>
```

O código gerado para Pi-Calculus é:

```
(new(sw1,sw2,sw3).(~sw1+~sw2+~sw3)|
(sw1. ~//inv/ws/AC/resPT/makres.
  (True(trvcan)|False(trvus)) +
 sw2. ~//inv/ws/AA/resPT/makres.
  (False(trvcan)|(True(trvus)+False(trvus))) +
 sw3. ~//inv/ws/BA/resPT/makres.
  (False(trvcan)|False(trvus)))
```

Percorrendo as atividades internas do *switch*, verifica-se que existem ao todo duas ligações nos elementos *source*. Portanto, para cada ramo geram-se os valores para os *linkStatus* dessas duas ligações através da ativação de processos *True* e *False* nos canais Pi-Calculus correspondentes.

No primeiro ramo (`sw1`), o elemento *source* é incondicional e é aplicado à ligação `travel-canada`. Portanto, o código gerado é a ativação do processo `True` para o canal correspondente a essa ligação, em paralelo com a ativação do processo `False` para o outro canal da outra ligação, que não foi referenciada nessa atividade.

No segundo ramo (`sw2`), foi gerada a expressão que é a soma do Pi-Calculus para a ativação de `True` e `False` no canal correspondente à ligação que está na expressão condicional, em paralelo com a ativação `False` para o outro canal.

A expressão traduzida do ramo *otherwise* contém a ativação, em paralelo, dos processos `False` para os dois canais, visto que esse ramo não contém nenhum elemento *source*.

4.5 Exemplo: Agência de Viagem

Esta seção contém um exemplo de tradução do processo da Agência de Viagem. O programa BPEL encontra-se no capítulo anterior. A seguir é listada a expressão Pi-Calculus correspondente à tradução:

```
1    //ws/rec/cli/cliPT/maktrvarr .
2    (new ( sw1,sw2,sw3) .
3     ( ( ~sw1 +
4         ~sw2 +
5         ~sw3
6       ) |
7       ( sw1 .
8         ~//ws/inv/AC/resPT/makres .
9         ( True<trvcan> |
10          False<trvus>
11        ) +
```

```
12          sw2 .
13          ~//ws/inv/AA/resPT/makres .
14          ( False<trvcan> |
15            ( False<trvus> +
16              True<trvus>
17            )
18          ) +
19          sw3 .
20          ~//ws/inv/BA/resPT/makres .
21          ( False<trvcan> |
22            False<trvus>
23          )
24        )
25      ) |
26      ~//ws/inv/weasrv/weaPT/getfcs |
27      new ( true,false,out) .
28      ( ~out<true,false> |
29        Or<trvcan, trvus, out> |
30        ( true .
31          ~//ws/inv/carren/carrenPT/rent .
32          True<ren2ass> +
33          false .
34          False<ren2ass>
35        )
36      ) |
37      new ( true,false) .
38      ~ren2ass<true,false> .
39      ( true .
40        ~//act/assign1 +
41        false
42      )
43    ) .
44    ~//act/assign2 .
45    ~//ws/rep/cli/cliPT/maktrvarr
```

A primeira atividade do processo é um comando *sequence* composto de quatro atividades: *receive, flow, assign* e *reply*. Esse comando é traduzido para uma construção de seqüência do Pi-Calculus, onde cada termo é separado pelo operador **"."** e corresponde à tradução de uma atividade. Os operadores aparecem no fim das linha 1, 43 e 44.

A próxima atividade é a operação *receive* que é traduzida para uma leitura em um canal que representa um *web service*, seguindo a nomenclatura sugerida neste capítulo (linha 1).

Segue-se um comando *flow*, que é convertido para a construção de paralelismo, ou seja, uma construção das traduções das atividades internas (*switch, invoke weatherService, invoke carRental* e *assign*) separadas por operadores `" | "`. Esses operadores estão no fim das linhas 25, 26 e 36.

As declarações *link* do BPEL não têm tradução para formar a expressão Pi-Calculus.

O comando *switch* possui três declarações *case*. A execução da expressão, que é a tradução desse comando, inicia-se alocando três canais internos (linha 2), em sequência ativa dois processos paralelos (separados pelo operador `" | "` na linha 6), sendo que o primeiro executa uma soma de escritas nos três canais (linhas 3, 4 e 5) e o outro uma soma de leituras nesses canais (linhas 7, 12 e 19). O *rendez-vous* em cada um desses canais está relacionado a um ramo do *switch*.

O primeiro ramo contém a cláusula *condition*, que não tem tradução para Pi-Calculus. Em seguida, a operação *invoke* é convertida para a escrita em um canal externo que representa um *web service* (linha 8). A cláusula *source* é traduzida para executar em seqüência à leitura do canal. A expressão gerada é uma construção paralela (linha 9) da chamada a um processo *True* no canal correspondente à ligação `travel-canada` (linha 9) e a um processo *False* no canal da ligação `travel-us` (linha 10).

O segundo ramo é similar ao primeiro, exceto que a cláusula *source* tem o atributo *transitionCondition*, podendo gerar valores *verdadeiro* ou *falso*. Essa cláusula é traduzida para uma construção de *soma* na qual os termos são chamadas aos processos *True* e *False* (linhas 15 e 16).

O terceiro ramo é a cláusula *otherwise* contendo uma atividade *invoke* que é traduzida para uma escrita em um canal externo (linha 20). Em seguida, o código Pi-Calculus é resultante da aplicação das regras de eliminação de caminho morto (linhas 21 e 22), gerando os processos *False* para os canais que pertencem aos elementos source dos outros ramos.

Terminada a tradução do comando *switch*, tem-se uma operação *invoke* sem elementos *source* e *target*. Portanto, a tradução é simplesmente aquela escrita no canal externo (linha 26).

A próxima operação é um *invoke*, que tem duas declarações *target*, que gera inicialmente o código para alocar três canais auxiliares (linha 27), sendo que os dois primeiros definem os valores *verdadeiro* e *falso*, respectivamente, e o terceiro é usado para obter o resultado da operação lógica (`out`). Em seguida, gera uma escrita dos valores lógicos no canal de resultado (linha 28), em paralelo com a chamada ao processo que executa a função lógica *Ou* entre os dois canais correspondentes às ligações `travel-canada` e `travel-us` (linha 29). O código seguinte é uma soma executada quando há um *rendez-vous* em um dos dois canais que representa os valores lógicos. Na primeira parcela, que representa o resultado *verdadeiro*, gera-se uma leitura no canal `true` (linha 30) e, em seqüência, a execução da escrita no canal externo (linha 31). Como existe uma cláusula *source*, gera-se a chamada ao processo *True* no canal da ligação `rent-to-assign` (linha 32). A segunda parcela (resultado *falso*) contém uma leitura do canal `false` (linha 33) e, na seqüência, uma chamada ao processo

False no canal da ligação `rent-to assign` (linha 34).

O próximo código a ser traduzido é um *assign* que tem uma cláusula *target*. O trecho de programa gerado é similar ao anterior, com uma simplificação, já que não existe expressão lógica. O código traduzido contém a alocação de dois canais para representar os valores *verdadeiro* e *falso* (linha 37). Esses valores são escritos no canal que representa a ligação `rent-to-assign` (linha 38); é gerada uma soma em que o primeiro termo corresponde ao valor *verdadeiro*, leitura no canal `true` (linha 39) e a escrita em um canal externo que representa uma atividade interna do processo BPEL (linha 40); o segundo termo representa o valor *falso*, leitura no canal `false` (linha 41) e não executa nada.

O comando seguinte ao *flow* é um *assign*, que é traduzido para uma escrita em canal externo (linha 44). Por fim, a operação seguinte (*reply*) é convertida para uma escrita em um canal externo de *web service* (linha 45).

4.6 Conclusão

Este capítulo descreveu o conjunto de regras para a tradução do programa BPEL para Pi-Calculus, enfatizando a obtenção dos mecanismos de concorrência e controle do fluxo dos processos. Essas regras são particulares para o objetivo deste trabalho e leva em consideração que a Máquina Virtual gera todas as computações possíveis para análise e verificação.

O capítulo seguinte apresenta o protótipo da Máquina Virtual que foi construído para executar um programa Pi-Calculus traduzido de um processo BPEL, gerando todas as reações possíveis e produzindo o resultado que é usado pelo módulo de Análise, para análise e verificação de propriedades do processo.

Capítulo 5

Máquina Virtual Pi-Calculus

Para a avaliação da aplicação dos conceitos desenvolvidos neste trabalho, foi construído um protótipo de uma Máquina Virtual Pi-Calculus (MVP) para atender aos requisitos de análise e verificação de programas Pi-Calculus baseados na geração de todos os caminhos de execução que um programa pode tomar. O interesse neste trabalho é referente aos programas que são traduções de processos de negócios escritos em BPEL.

Uma *linha de execução* de um programa na MVP é iniciada na ativação do processo *main*. Toda vez que uma linha de execução tiver a possibilidade de executar mais que um *rendez-vous*, então essa linha de execução é desdobrada em tantas quantos forem os *rendez-vous* possíveis, sendo que para cada uma delas é atribuído um *rendez-vous* diferente. Esse procedimento é repetido até que uma linha de execução se encerre. O término de uma linha de execução ocorre nas seguintes condições: a expressão resultante é 0; um comando *exit* foi executado; ou não tem nenhum *rendez-vous* pendente, mas existe pelo menos um canal esperando uma operação de leitura ou escrita (*deadlock*).

Um *caminho de execução* corresponde a um conjunto de passos que uma linha de execução percorre. Assim, por exemplo, um caminho de execução pode ser formado pela seqüência de canais em que ocorrem os *rendez-vous* desde o início até o fim de uma linha de execução. Essa seqüência pode conter todos os canais ou somente os canais externos.

Este capítulo está estruturado da seguinte forma: a seção 5.1 é composta pela descrição da linguagem da MVP; a seção 5.2 introduz a arquitetura da MVP em termos de blocos funcionais e interfaces; a seção 5.3 apresenta a implementação da MVP no que se refere aos pacotes e às classes que implementam os elementos da arquitetura; a seção 5.4 descreve o módulo de Análise na perpectiva do usuário e sua aplicação no exemplo da Agência de Viagem; a seção 5.5 contém outro exemplo, mais complexo, que acrescenta outros cenários de análise, incluindo a detecção de *deadlocks*.

5.1 A Linguagem da Máquina Virtual Pi-Calculus

A linguagem da Máquina Virtual é definida em um esquema XML. Ela contém construções que correspondem às primitivas do Pi-Calculus, ou seja, contém comandos para: definir e chamar processos, alocar, ler e escrever em canais, construir seqüências, paralelismos, somas e replicações. Para permitir a tradução da atividade *terminate* do BPEL, foi adicionado o comando *exit*.

A figura 5.1 ilustra os tipos de declarações e comandos da Linguagem da Máquina Virtual e as principais entidades envolvidas:

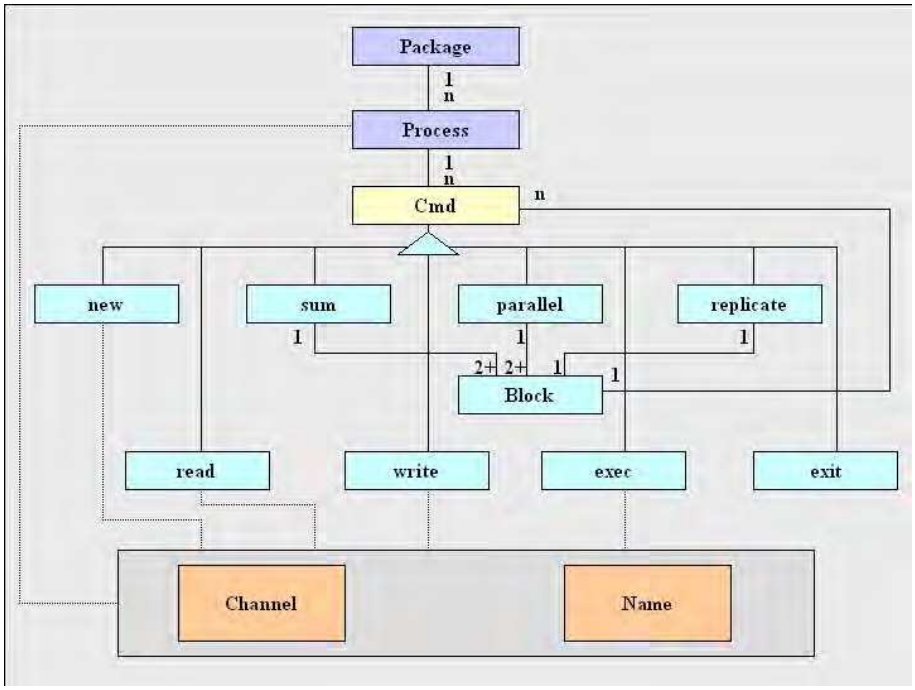

Fig. 5.1: Elementos de um programa na Linguagem da Máquina Virtual Pi-Calculus.

No topo da figura tem-se a declaração *package* (*pacote*) [1] que está associada a um arquivo com definições de processos (declarações *process*). Um processo contém uma ou mais declarações de comandos representadas na figura como uma entidade abstrata denominada *Cmd*. Esses comandos são agrupados em três classes.

A primeira classe consiste dos comandos compostos: *sum* (*somar*), *parallel* (*paralelizar*) e *replicate* (*replicar*). Cada termo de um comando composto é delimitado pelo construtor *block* (*bloco*), que por sua vez contém um ou mais comandos *Cmd*.

A segunda classe é constituída dos comandos *new* (*alocar canal*), *read* (*ler*) e *write* (*escrever*). O primeiro aloca um canal e atribui o valor obtido para um nome restrito. O comando *read* lê de um

[1] Um pacote define um programa Pi-Calculus. Os termos "programa" e "pacote" são usados intercambiavelmente.

canal uma tupla de nomes livres e atribui os nomes livres aos nomes restritos que são passados como parâmetros. O comando *write* escreve em um canal uma tupla de nomes passados como parâmetros. Se o nome de um parâmetro é um nome restrito, então o valor a ser escrito é o nome livre associado.

A terceira, e última, classe contém os comandos: *exec* (*executar processo*) e *exit* (*terminar programa*). O primeiro é usado para ativar um processo que está definido através da declaração *process*. O comando *exit* termina incondicionalmente uma *linha de execução*.

Entidades externas - outras funcionalidades

Nas especificações do mapeamento BPEL para Pi-Calulus do capítulo 4, foi definido que os canais Pi-Calculus são classificados entre canais internos e externos. Esse requisito é atendido pela MVP, ou seja, existem os canais internos em que o *rendez-vous* ocorre pelas operações de leitura e escrita em um canal por processos paraleles, e os canais externos em que uma das operações é realizada na MVP e a outra por uma *Entidade Externa*.

Entidades Externas são elementos que se comunicam com a MVP através de canais externos e que de uma maneira geral podem implementar diferentes funcionalidades. Para flexibilizar o desenvolvimento dessas novas funcionalidades, foi definida na linguagem da MVP a declaração *external* que associa um conjunto de nomes de canais a uma função, que é definida por uma URI (*Universal Resource Identifier*).

Na versão atual do protótipo, os canais externos são usados para:

- as operações *web services* (canais iniciados com a substring `"//ws"`);
- as atividades internas do programa BPEL consideradas relevantes para análise (nomes iniciados com `"//act"`);
- a MVP comunicar os resultados dos programas que ela executa (canais iniciados por `"//status"`).

O esquema XML da linguagem está no Apêndice A, enquanto o apêndice B contém o programa completo correspondente à tradução do processo BPEL do exemplo da Agência de Viagem, descrito no capítulo 4.

5.2 Arquitetura da Máquina Virtual

A figura 5.2 ilustra a arquitetura da MVP em termos de blocos funcionais e interfaces:

Existem quatro tipos de interface: a interface *Carga* contém a função que faz a carga do programa Pi-Calculus de um arquivo para o processador; a interface *Monitor* é usada para o acompanhamento

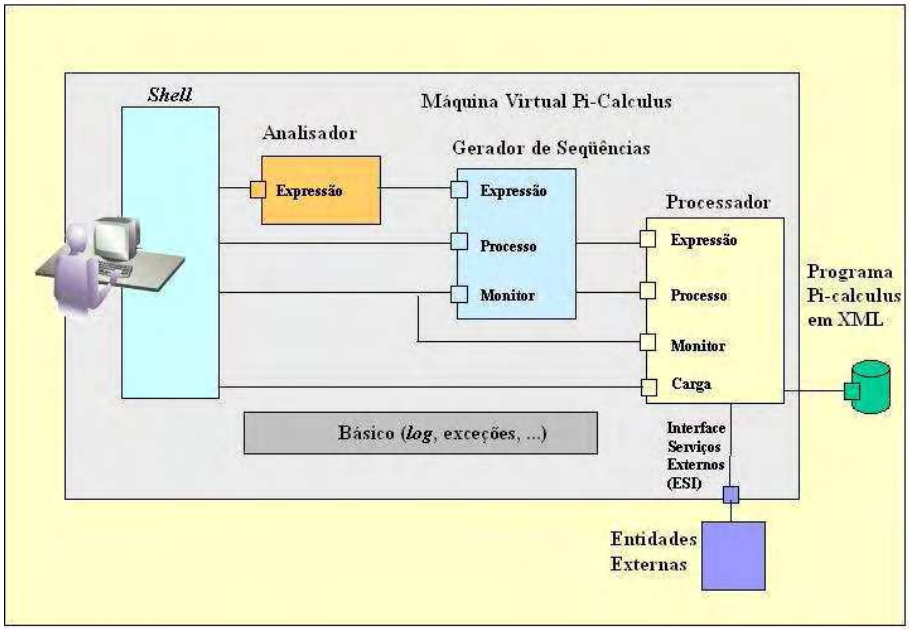

Fig. 5.2: Arquitetura da Máquina Virtual Pi-Calculus (MVP).

passo-a-passo da execução do programa, permitindo também visualizar as estruturas internas da máquina; a interface *Processo* contém métodos para coordenar o processo passo-a-passo; finalmente, as operações sobre estruturas de dados e métodos de conversão de estado de programa para expressões são providas pela interface *Expressão*.

Existem duas implementações do bloco funcional[2] *Shell*: a *Console* e a *Interface Gráfica*. A Console é usada principalmente para o acompanhamento passo-a-passo da execução do programa quando o objetivo é o entendimento detalhado das estruturas internas ou a depuração da MVP. A Interface Gráfica é amigável e se aplica à análise do programa Pi-Calculus.

A Máquina Virtual é iniciada quando uma *Shell* é ativada do sistema operacional. Inicialmente, a *Shell* registra o objeto que receberá as mensagens de monitoração e, em seguida, carrega o programa e inicia o *Gerador de Seqüência*. Este módulo controla o *Processador* visando executar o programa e, assim, obter todos os caminhos de execução possíveis e armazenar na *Árvore de Execução*. Durante a fase de construção, cada nó da Árvore de Execução tem os seguintes atributos preenchidos: o *canal*, a *expressão* e a *referência de equivalência*. Genericamente, o *canal* é obtido do começo de uma reação - é o canal em que ocorre o *rendez-vous* -; a *expressão* é determinada no instante imediatamente anterior à transferência de dados no próximo *rendez-vous* e, se essa expressão já existe em algum outro nó da árvore, então a *referência de equivalência* aponta para esse nó; caso contrário, contém o

[2]Os termos "bloco funcional" e "módulo" estão sendo usados com o mesmo significado e são relativos à figura da arquitetura.

valor nulo. A Árvore de Execução é construída com a raiz contendo os seguintes valores: o atributo *canal* recebe simbolicamente o nome do programa que está sendo executado, o atributo de *expressão* contém a expressão inicial do programa e a *referência de equivalência* assume o valor nulo.

O Gerador de Seqüência inicia a Árvore de Execução com o nó raiz e chama o Processador para ativar o processo *main*. O Processador interpreta e executa instruções até que a próxima execução possível seja a troca de dados através de um canal, isto é, a ocorrência de um *rendez-vous*. Esse estado finaliza o primeiro passo. Nesse ponto, o controle é retornado para o Gerador de Seqüências criando um nó que contém: o valor simbólico "main" para o atributo *canal*, a *expressão* determinada em função do estado corrente do programa e a *referência de equivalência* com o valor nulo. Esse nó é anexado à Árvore de Execução.

Se existe pelo menos um *rendez-vous* pronto para ocorrer, então para cada *rendez-vous* pendente armazena-se na *Pilha de Contexto* a seguinte tupla: o contexto do Processador, a referência para o *rendez-vous* que ocorrerá no próximo passo e o nó corrente da Árvore de Execução.

O controle é passado de volta para a *Shell*. Nesse momento, o usuário da *Console* pode executar comandos de monitoração para listar estruturas internas da MVP.

Em seguida, a *Shell* chama o Gerador de Seqüências para executar o próximo passo. Uma tupla é desempilhada da Pilha de Contextos. O contexto do Processador é restaurado e este é ativado para executar o *rendez-vous* daquele contexto e continuar processando instruções até que a próxima execução possível seja a ocorrência de um novo *rendez-vous* ou a linha de execução (programa nesse contexto) termina (executou uma operação *exit* ou não tem mais *rendez-vous* para acontecer). Nesse ponto, o passo é finalizado e o controle volta para o Gerador de Seqüências.

No caso em que uma linha de execução é concluída, o programa cria mais um nó (folha) na Árvore de Execução com o atributo *canal* assumindo o nome de um canal de status que indica se esta contém, ou não, uma situação de *deadlock* (veja detalhes em 5.4.2). Os demais atributos são irrelevantes.

No caso em que a linha de execução não terminou, o Gerador de Seqüências anexa à Árvore de Execução um nó contendo: o canal em que ocorreu o *rendez-vous*; a expressão corrente determinada a partir do estado do Processador; se já existe uma expressão igual na árvore, então a referência de equivalência aponta para esse nó; e nesse caso a execução desse contexto pelo Gerador de Seqüência pára, pois a linha de execução restante é igual ao nó equivalente; caso não exista a expressão igual, a referência de equivalência contém o valor nulo.

As interações entre a *Shell*, o Gerador de Seqüência e o Processador descritas nos cinco parágrafos anteriores são repetidas até que a Pilha de Contextos esteja vazia. Quando a execução atinge essa condição, é porque o processo de gerar todas as reações terminou. O resultado é a Árvore de Execução.

O Analisador provê a ferramenta para a análise e verificação de propriedades do programa. A

entrada desse módulo é a Árvore de Execução. O Analisador extrai os nós que contêm os *rendez-vous* nos canais externos e produz uma outra árvore, denominada *Árvore-X*. Isso é feito porque, para a análise, o interesse são as interações com os *web services*, execuções de atividades relevantes e os canais de *status*. A Árvore-X é convertida para um formato de tabela denominado *Tabela de Execução* e armazenada em uma variável do Analisador cujo nome é *xtable*. Uma *Tabela de Execução* é uma estrutura em que cada coluna representa um caminho de execução através das interações nos canais externos. As colunas são distintas entre si, ou seja, não existem duas colunas com o mesmo caminho de execução.

O módulo *Básico* contém funções genéricas usadas pelo outros módulos. A interface *Interface de Serviço Externo* é reservada para o futuro, quando será definido um protocolo para adaptar entidades externas à MVP.

5.3 Implementação da MVP

O protótipo da MVP foi implementado em Java. A ativação da MVP para um programa Pi-Calculus corresponde à ativação de um programa em uma Máquina Virtual Java.

O sistema é composto de pacotes, sendo que cada pacote está associado a um bloco funcional da arquitetura.

5.3.1 Módulo Processador

O pacote que implementa o bloco funcional Processador tem o nome `Processor` . Esse pacote contém dois outros pacotes, denominados *Loader* e *Command*, e um conjunto de classes que implementam as estruturas de dados e algoritmos que suportam a execução do programa. O pacote *Loader* tem a classe que faz a carga do programa na memória do Processador. A memória é um vetor de objetos, em que cada objeto é uma instância da classe que representa o comando da linguagem da MVP. Essas classes estão definidas no pacote *Command*. Cada instância de um comando composto é associada a um comando que delimita o seu término.

Ao todo, o pacote contém trinta e oito classes. O acesso a qualquer funcionalidade do pacote por outro módulo é feito através da classe `PROServer`. Essa classe contém os métodos das quatro interfaces ilustradas na arquitetura. A interface[3] de carga contém um método para carregar o programa de arquivo para a memória. A interface de monitoração possui métodos para registrar os objetos que recebem mensagens de acompanhamento do andamento da execução e também métodos para listar as principais tabelas da MVP: Tabela de Processos, Tabela de Canais e Tabela de *Rendez-vous*.

[3]O termo "interface" se refere à ilustração da arquitetura e não ao modelo da linguagem Java.

A interface de processo é subdividida em dois conjuntos de métodos. O primeiro é sobre o gerenciamento de processos e contém métodos para criar o programa principal, ativar um passo de processamento, obter dados como: número de *rendez-vous* pronto para ocorrer, canal pronto para *rendez-vous* e processos em estado de espera. O outro conjunto está relacionado ao contexto do programa e contém métodos para criar, salvar e restaurar um contexto global. A interface de expressão possui métodos para: listar a expressão inicial do programa, e determinar a expressão corrente de uma linha de execução. A Interface Externa de Serviço não está definida nessa versão.

5.3.2 Módulo Gerador de Seqüências

O pacote denominado `Generator` implementa esse bloco funcional. O pacote contém nove classes, sendo três para interface com os outros módulos. Duas classes são para operações na Árvore de Execução `GENExecutionTree` e `GENExecutionTreeNode` , sendo que a classe principal é a `GENServer` .

A classe `GENServer` possui métodos na interface de monitoração para registrar objetos que recebem as mensagens. Na interface de processo, a classe possui métodos para iniciar um programa, executar o primeiro passo de um programa, executar o próximo passo e métodos para obtenção de dados como: verificar se uma linha de execução tem mais passos a serem executados, obter a Árvore de Execução resultante do processamento completo de um programa e obter, em caso de *deadlock*, os canais que estão nessa condição.

A classe `GENExecutionTree` , além dos construtores e métodos para estabelecer valores e ler valores dos atributos, contém métodos para construir a Árvore-X, verificar se um programa é livre de *deadlock* e listar (gerar uma *string* formatada) a árvore. A classe `GENExecutionTreeNode` contém construtores e métodos para estabelecer e ler valores dos atributos.

5.3.3 Módulo Analisador

O bloco funcional Analisador é implementado pelo pacote `Analyzer` que possui quatro classes, todas acessíveis pelas entidades externas. Essas classes são: `PANExecutionTree` , `PANExecutionTreeNode`, `PANExecutionTable` e `PANServer` . As duas primeiras são encapsulamentos das classes do módulo Gerador de Seqüências.

A Classe `PANServer` contém métodos para criar a Tabela de Execução a partir da Árvore de Execução, retornando um objeto da classe `PANExecutionTable` , e para executar o comando *Select* sobre uma Tabela de Execução. Esse comando é a ferramenta de análise desse protótipo e está detalhado na seção 5.4.

A classe `PANExecutionTable` possui construtores e métodos que permitem construir tabelas de

execução de diversas maneiras. A classe contém também métodos para obter os nomes das colunas, listar a tabela (gerar uma *string* formatada) e gerar uma cópia.

5.3.4 Módulo Básico

O pacote que implementa o módulo Básico é o *Basic*. Esse pacote possui classes que são de uso geral pelos outros módulos. As funcionalidades implementadas são: gerenciamento do *log* de software, regras de nomeação dos canais, indentação de uma expressão Pi-Calculus, exceções e definição da interface Java do cliente que recebe as mensagens de monitoração.

5.3.5 Módulo Shell

Como já foi dito, esse módulo tem duas implementações: a Console e a Interface Gráfica.

Console

A Console foi a primeira a ser construída e contém o conjunto completo de comandos para a operação da MVP. Esse módulo deve ser usado para a monitoração do programa, quando se deseja depurar uma funcionalidade da MVP ou ainda entender como as estruturas de dados se comportam com a evoloução da execução de um programa. O pacote que implementa a Console tem o nome `Shell`.

A Console apresenta em seu menu somente os comandos que são aplicáveis a cada fase de execução do programa. Ao todo são quatro fases:

1. após a carga do programa e antes de iniciar a execução;
2. durante a execução do programa;
3. após o término do programa;
4. durante a análise e verificação do programa.

A tabela 5.1 contém a lista de comandos disponíveis para o usuário e as fases às quais os comandos se aplicam.

Interface Gráfica

A Interface Gráfica foi desenvolvida para facilitar a análise do programa Pi-Calculus. O pacote que implementa a Interface Gráfica tem o nome *GUI*. O menu principal contém duas colunas com

Comando	Descrição	Fases
ListPackage	Lista a expressão Pi-Calculus do programa.	1, 2 e 3.
CloseLog	Fecha o log para permitir listar o arquivo do diretório.	1, 2 e 3
Trace	Habilita ou desabilita as mensagens de trace.	1 e 2
Resume	Executa um passo do programa e retorna para a Console.	1 e 2
Goto	Executa até alcançar a linha de trace dada como parâmetro; se o parâmetro é omitido, executa até o fim do programa.	1 e 2
ListExpression	Lista a expressão Pi-Calculus do estado corrente do programa.	2
ListTree	Lista os nós da Árvore de Execução.	2 e 3
Dump	Lista uma das três principais tabelas da MVP: Tabela de Processos, Tabela de Canais e Tabela de Rendez-vous.	2
Analyze	Vai para a execução do Analisador.	3
CheckDeadlock	Verifica se existe caminhos de execução que resultam em *deadlock*.	4
Select	Seleciona os caminhos de execução de uma Tabela de Execução conforme os parâmetros passados. A descrição detalhada desse comando está na seção 5.4.	4
ListVariables	Lista as variáves do Analisador que contém tabelas de execução.	4
Exit	Termina a execução da MVP.	todas

Tab. 5.1: Comandos da Console da MVP.

as opções de comandos. A primeira coluna, *Execução*, contém opções para executar um programa, listar a expressão inicial do programa e terminar a execução da MVP.

A outra coluna, *Análise*, dispõe as opções para a análise do programa. A opção *listar Árvore* apresenta graficamente a Árvore de Execução em que cada nó contém a sua identificação, o nome do canal e a referência de equivalência. Ao se clicar na representação gráfica do nó, a Interface Gráfica lista a expressão daquele nó. A opção *listar Árvore-X* apresenta de forma análoga a Árvore-X. A opção *verificar deadlock* apresenta como resultado uma caixa de diálogo informando que o programa é livre de *deadlock* ou uma janela com a lista dos *deadlocks* encontrados e as respectivas identificações dos nós da Árvore de Execução em que foram detectados.

Por último, a opção *executar select* reorganiza a janela principal para a disposição mostrada na figura 5.3. Na parte de cima da janela e à esquerda, a Interface Gráfica apresenta a lista das Tabelas de

Execução construídas durante a análise e que estão disponíveis para serem mostradas em uma nova janela através do botão *Listar*. À direita é sempre apresentada a *xtable*. A parte de baixo da janela contém uma caixa que permite construir os comandos *select*. O botão de *Ajuda* lista a sintaxe do comando. O botão *Limpar* apaga o que estava sendo editado na caixa de entrada de comandos. O botão *Executar* invoca o Analisador para processar a linha de comando montada na caixa de entrada. O resultado de um comando *select* é a criação ou a atualização de uma Tabela de Execução e um *status* que é mostrado na caixa de baixo. Esse *status* inclui o número de colunas selecionadas da tabela de entrada do comando e o nome da tabela de saída.

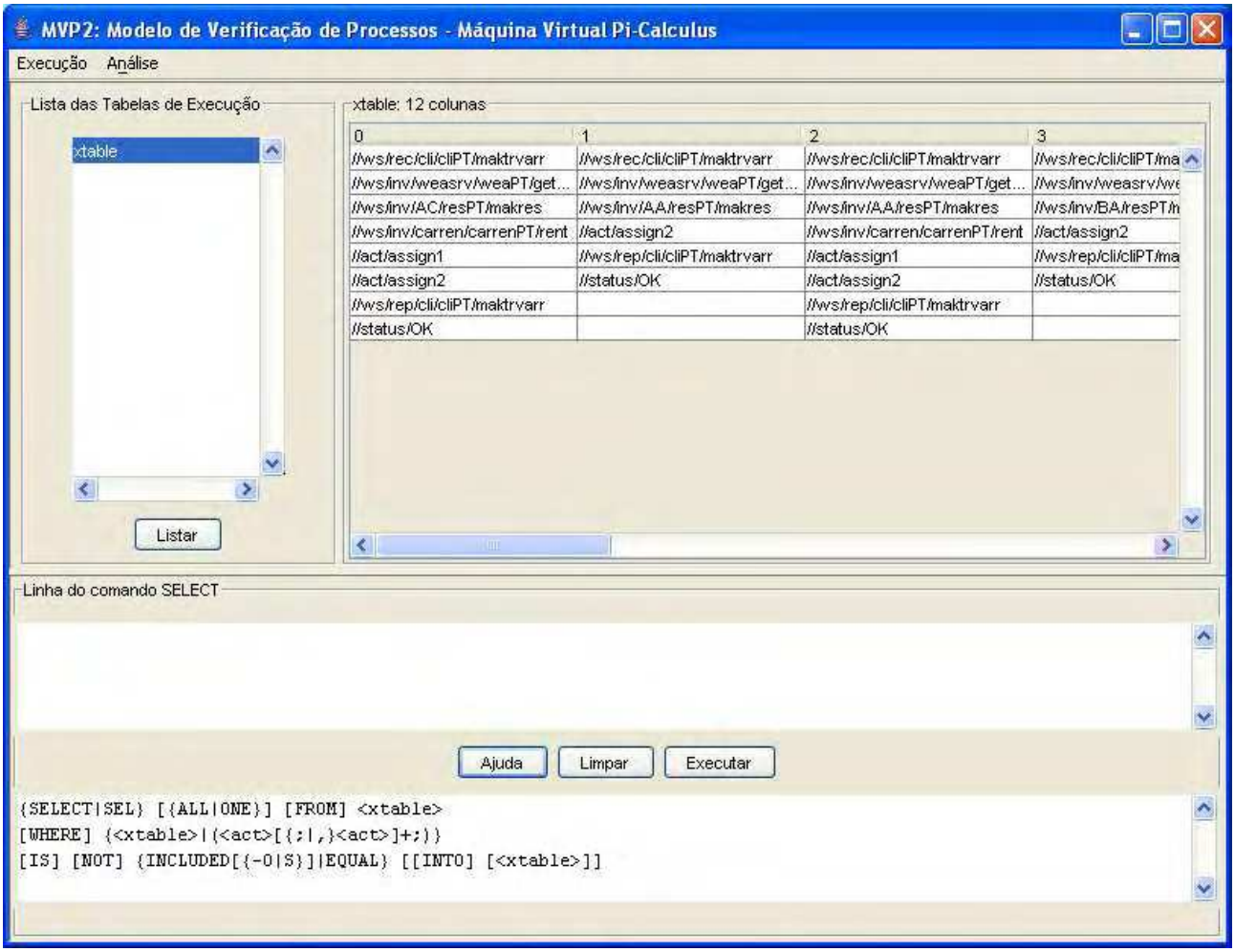

Fig. 5.3: Janela da Interface Gráfica para a análise de programas.

5.4 Análise e verificação de programas Pi-Calculus

5.4.1 Árvore de Execução

O ponto de partida para a análise e a verificação do programa Pi-Calculus é a Árvore de Execução. Esta árvore contém todos os caminhos possíveis que o programa pode tomar.

Como já foi mencionado na seção 5.2, os nós de uma Árvore de Execução contêm os atributos *canal*, *expressão* e *referência de equivalência*. Após a construção final da árvore, esta é percorrida, e para cada nó são determinados os valores dos atributos *chave* e *identificação*. Ambos servem para identificar unicamente um nó na árvore. O primeiro é um número estruturado para ser usado pela Console e o outro é um inteiro positivo usado pela Interface Gráfica.

Uma parte da Árvore de Execução do programa exemplo da Agência de Viagem, listada através da Console, é mostrada a seguir (a árvore completa encontra-se no Apêndice C):

```
0 C:\6Vanine\02Unicamp\tese\mvp\mvp\input\TravelAgent.xml
1 main
2 //ws/rec/cli/cliPT/maktrvarr
2.1 //ws/inv/weasrv/weaPT/getfcs
2.1.1 sw1#1
2.1.1.1 out#6
2.1.1.1.1 //ws/inv/AC/resPT/makres
2.1.1.1.1.1 trvcan
2.1.1.1.1.1.1 trvus
2.1.1.1.1.1.1.1 true1#9
2.1.1.1.1.1.1.2 false2#12
2.1.1.1.1.1.1.3 true#4
2.1.1.1.1.1.1.4 //ws/inv/carren/carrenPT/rent
2.1.1.1.1.1.1.5 ren2ass
2.1.1.1.1.1.1.6 true#7
2.1.1.1.1.1.1.7 //act/assign1
2.1.1.1.1.1.1.8 //act/assign2
2.1.1.1.1.1.1.9.0 //ws/rep/cli/cliPT/maktrvarr
2.1.1.1.1.1.2 true1#9
2.1.1.1.1.1.2.1 trvus <==> 2.1.1.1.1.1.1.1
2.1.1.1.1.2 trvus
2.1.1.1.1.2.1 trvcan <==> 2.1.1.1.1.1.1
2.1.1.2 //ws/inv/AC/resPT/makres
2.1.1.2.1 out#6 <==> 2.1.1.1.1
```

Esta listagem apresenta para cada nó da árvore os atributos *chave*, *canal* e a *referência de equivalência* (quando este não possui o valor nulo). As regras para a formação das chaves dos nós são

descritas a seguir:

- o número 0 representa a raiz da árvore e o número 1 o primeiro passo, que vai do início da execução do programa até o instante imediatamente anterior à ocorrência do primeiro *rendez-vous*;

- quando o nó possui filhos, então cada incremento significa uma linha de execução distinta. Assim, por exemplo, o programa, após ter executado o *rendez-vous* do nó 2 (recepção do pedido de serviço), é desdobrado nas seguintes linhas de execução, cujos primeiros *rendez-vous* são:

  1. nó 2.1: chamada ao serviço de previsão do tempo;
  2. nós 2.2 a 2.4: *rendez-vous* que permitem a execução, respectivamente, de cada um dos três ramos da atividade *switch* do programa BPEL;
  3. nó 2.5: *rendez-vous* no canal *out* que é a saída da expressão `(Or (trvcan,trvus,out))`. Essa expressão define se o *web service* que faz o aluguel do carro será executado ou não;

- quando o nó não possui mais que um filho, então o incremento indica seqüência de execução em uma computação, como exemplificado pelos nós de 2.1.1.1.1.1.1.1 até 2.1.1.1.1.1.1.8;

- quando o nó é uma folha da árvore, então ele é terminado com 0. O item 2.1.1.1.1.1.1.9.0 é a folha de uma linha de execução que contém os itens de 2.1.1.1.1.1.1.1 até 2.1.1.1.1.1.1.8.

A figura 5.4 mostra a mesma árvore na Interface Gráfica. Cada nó contém a *identificação*, o *canal* e a *referência de equivalência*. Esta forma de apresentação também inclui a visualização da expressão Pi-Calculus, que é o resultado da execução do *rendez-vous* no canal em destaque. Na figura, esse canal é o `(11)false2#12` que corresponde ao nó 2.1.1.1.1.1.1.2 da árvore mostrada pela Console.

A Árvore de Execução completa inclui os *rendez-vous* dos canais que são usados para controle de concorrência e fluxo do processo. Em termos do processo BPEL, o que interessa são os canais que representam os *web services*, as atividades internas BPEL que são relevantes e os canais inseridos pela MVP para indicar os *status* de um caminho de execução. Esses canais são representados pelos canais externos da MVP.

O módulo de Análise constrói uma outra árvore, denominada *Árvore-X*, que contém a mesma estrutura da Árvore de Execução, mantendo-se somente os nós que representam *rendez-vous* em canais externos.

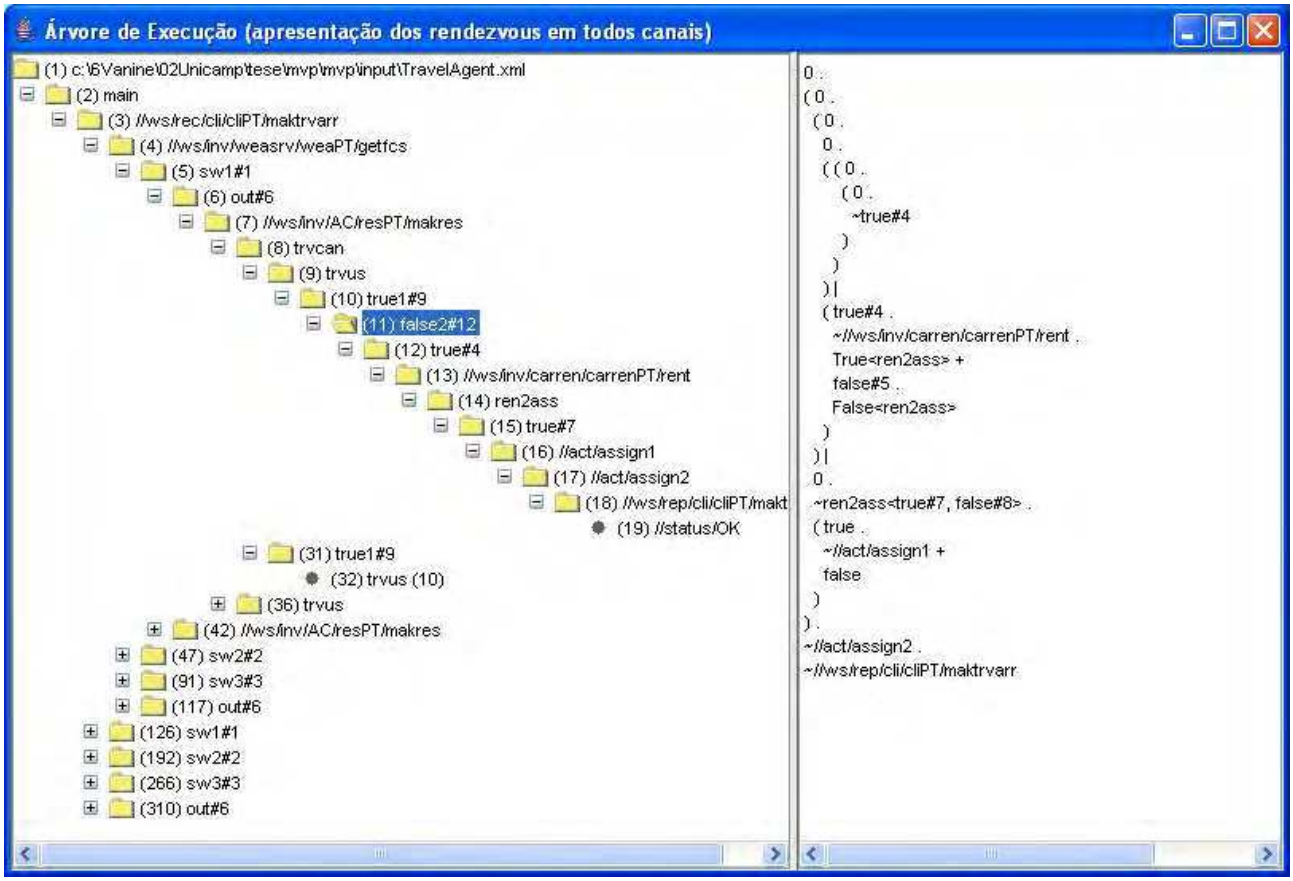

Fig. 5.4: Árvore de Execução do processo da Agência de Viagem.

5.4.2 Canais de *status*

Cada folha da Árvore de Execução contém um canal que representa o *status* do caminho de execução. Este *status* indica se o programa é livre de *deadlock* ou não. No primeiro caso, o canal é dado pelo nome `//status/OK` . Quando o caminho de execução termina em *deadlock*, o nome do canal é formado pela concatenação da *substring* `//status/deadlock/` com uma sequência de nomes de canais separados pelo caracter `&`. Estes canais são os que, ao término de uma linha de execução, possuem uma operação em andamento e estão esperando pela operação que completa o *rendez-vous*, ou seja, são os canais que provocaram o *deadlock*. Assim, por exemplo, se um caminho de execução termina em *deadlock* nos canais `buy` e `sell` , então o nome do canal de *status* é: `//status/deadlock/buy&sell` .

5.4.3 Tabela de Execução

A estrutura utilizada para a análise do programa Pi-Calculus gerado a partir do processo BPEL é a *Tabela de Execução*. Essa tabela é de duas dimensões em que cada coluna contém uma seqüência de canais externos de um caminho de execução. Ao ser construída, uma Tabela de Execução preserva a propriedade de todas as suas colunas serem distintas entre si.

O módulo de Análise, quando iniciado, recebe a Árvore de Execução e gera uma Tabela de Execução inicial, denominada *xtable*. A ferramenta de análise gera outras Tabelas de Execução que permitem mostrar propriedades do processo.

A *xtable* do exemplo da Agência de Viagem é mostrada a seguir:

```
[0,0] //ws/rec/cli/cliPT/maktrvarr
[0,1] //ws/inv/weasrv/weaPT/getfcs
[0,2] //ws/inv/AC/resPT/makres
[0,3] //ws/inv/carren/carrenPT/rent
[0,4] //act/assign1
[0,5] //act/assign2
[0,6] //ws/rep/cli/cliPT/maktrvarr
[0,7] //status/OK

[1,0] //ws/rec/cli/cliPT/maktrvarr
[1,1] //ws/inv/weasrv/weaPT/getfcs
[1,2] //ws/inv/AA/resPT/makres
[1,3] //act/assign2
[1,4] //ws/rep/cli/cliPT/maktrvarr
[1,5] //status/OK

[2,0] //ws/rec/cli/cliPT/maktrvarr
[2,1] //ws/inv/weasrv/weaPT/getfcs
[2,2] //ws/inv/AA/resPT/makres
[2,3] //ws/inv/carren/carrenPT/rent
[2,4] //act/assign1
[2,5] //act/assign2
[2,6] //ws/rep/cli/cliPT/maktrvarr
[2,7] //status/OK

[3,0] //ws/rec/cli/cliPT/maktrvarr
[3,1] //ws/inv/weasrv/weaPT/getfcs
[3,2] //ws/inv/BA/resPT/makres
[3,3] //act/assign2
[3,4] //ws/rep/cli/cliPT/maktrvarr
```

```
[3,5] //status/OK

[4,0] //ws/rec/cli/cliPT/maktrvarr
[4,1] //ws/inv/AC/resPT/makres
[4,2] //ws/inv/weasrv/weaPT/getfcs
[4,3] //ws/inv/carren/carrenPT/rent
[4,4] //act/assign1
[4,5] //act/assign2
[4,6] //ws/rep/cli/cliPT/maktrvarr
[4,7] //status/OK

[5,0] //ws/rec/cli/cliPT/maktrvarr
[5,1] //ws/inv/AC/resPT/makres
[5,2] //ws/inv/carren/carrenPT/rent
[5,3] //ws/inv/weasrv/weaPT/getfcs
[5,4] //act/assign1
[5,5] //act/assign2
[5,6] //ws/rep/cli/cliPT/maktrvarr
[5,7] //status/OK

[6,0] //ws/rec/cli/cliPT/maktrvarr
[6,1] //ws/inv/AC/resPT/makres
[6,2] //ws/inv/carren/carrenPT/rent
[6,3] //act/assign1
[6,4] //ws/inv/weasrv/weaPT/getfcs
[6,5] //act/assign2
[6,6] //ws/rep/cli/cliPT/maktrvarr
[6,7] //status/OK

[7,0] //ws/rec/cli/cliPT/maktrvarr
[7,1] //ws/inv/AA/resPT/makres
[7,2] //ws/inv/weasrv/weaPT/getfcs
[7,3] //act/assign2
[7,4] //ws/rep/cli/cliPT/maktrvarr
[7,5] //status/OK

[8,0] //ws/rec/cli/cliPT/maktrvarr
[8,1] //ws/inv/AA/resPT/makres
[8,2] //ws/inv/weasrv/weaPT/getfcs
[8,3] //ws/inv/carren/carrenPT/rent
[8,4] //act/assign1
[8,5] //act/assign2
[8,6] //ws/rep/cli/cliPT/maktrvarr
```

```
[8,7] //status/OK

[9,0] //ws/rec/cli/cliPT/maktrvarr
[9,1] //ws/inv/AA/resPT/makres
[9,2] //ws/inv/carren/carrenPT/rent
[9,3] //ws/inv/weasrv/weaPT/getfcs
[9,4] //act/assign1
[9,5] //act/assign2
[9,6] //ws/rep/cli/cliPT/maktrvarr
[9,7] //status/OK

[10,0] //ws/rec/cli/cliPT/maktrvarr
[10,1] //ws/inv/AA/resPT/makres
[10,2] //ws/inv/carren/carrenPT/rent
[10,3] //act/assign1
[10,4] //ws/inv/weasrv/weaPT/getfcs
[10,5] //act/assign2
[10,6] //ws/rep/cli/cliPT/maktrvarr
[10,7] //status/OK

[11,0] //ws/rec/cli/cliPT/maktrvarr
[11,1] //ws/inv/BA/resPT/makres
[11,2] //ws/inv/weasrv/weaPT/getfcs
[11,3] //act/assign2
[11,4] //ws/rep/cli/cliPT/maktrvarr
[11,5] //status/OK
```

A figura 5.5 ilustra parte da mesma tabela de execução na Interface Gráfica.

5.4.4 O Comando SELECT

A análise do programa Pi-Calculus é feita através do comando SELECT do módulo de Análise. Esse comando é uma ferramenta de nível baixo, que, usado em diversas combinações, permite fazer as análises e verificações de mais alto nível.

A sintaxe do SELECT é dada a seguir:

```
{SELECT|SEL} [{ALL|ONE}] [FROM] <xtable>
[WHERE] {<xtable>|(<act>[{;|,}<act>]+;)}
[IS] [NOT] {INCLUDED[{-O|S}]|EQUAL} [[INTO] [<xtable>]]
```

O comando recebe como entrada uma Tabela de Execução e produz na saída outra tabela desse tipo. A análise começa a partir da Tabela de Execução inicial (*xtable*).

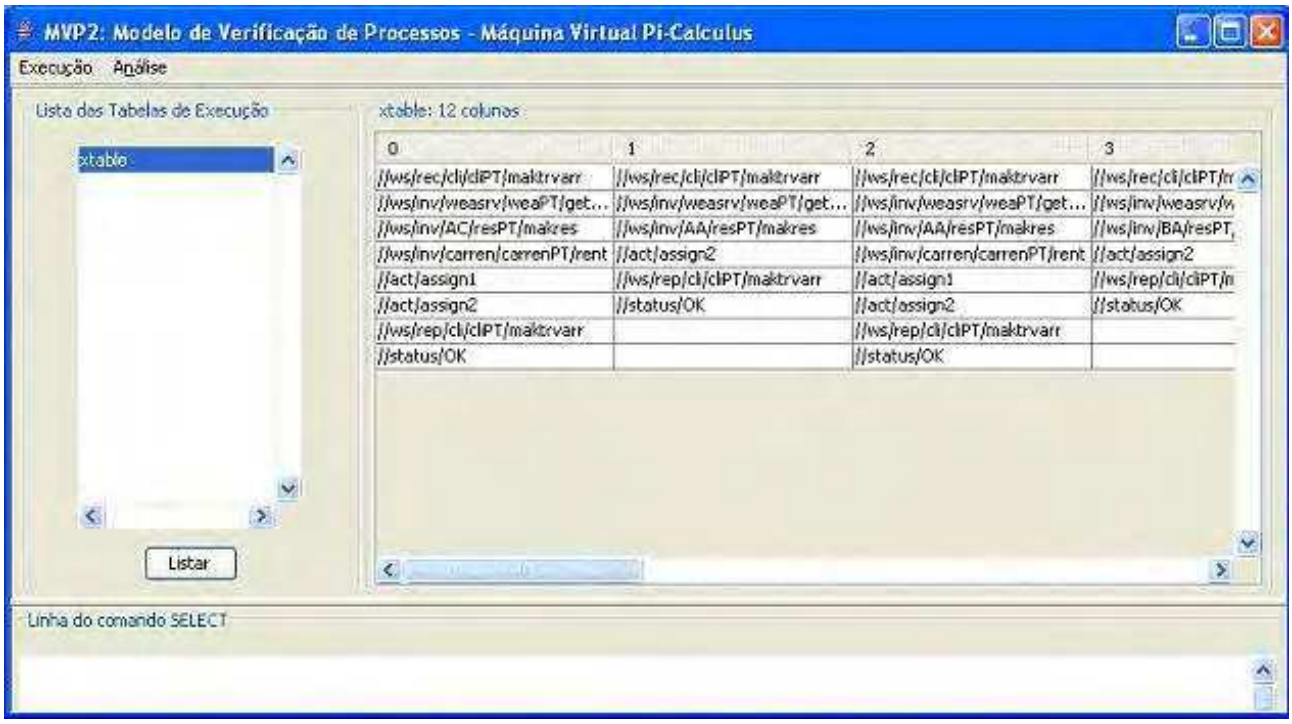

Fig. 5.5: Tabela de Execução do processo da Agência de Viagem.

O objetivo desse comando é selecionar colunas da tabela, ou seja, caminhos de execução. O primeiro parâmetro `(ALL|ONE)` indica se se deseja extrair para a saída apenas um caminho de execução que satisfaz a condição ou todos os caminhos.

O próximo parâmetro (`FROM`) é a tabela de entrada. O parâmetro `WHERE` contém o objeto a ser usado na condição a ser aplicada na tabela de entrada. Esse objeto pode ser uma tabela obtida de outras operações ou uma especificação de atividades (canais externos). Essa especificação pode ser considerada como uma tabela. A especificação é delimitada pelo abre e fecha parêntesis. Internamente, as colunas da tabela são separadas por ponto e vírgula, enquanto que dentro de uma coluna as atividades são separadas por vírgula.

A condição básica é testar se um conjunto de atividades está ou não na tabela. Se não houver nenhum modificador no parâmetro `INCLUDED`, o significado é: "seleciona todas (supondo parâmetro de seleção igual a `ALL`) as colunas da tabela de entrada (`FROM`) que contenham todas as atividades de uma coluna da especificação da tabela objeto; essa regra é aplicada para todas colunas da tabela objeto".

Caso o modificador seja `-O` (ordenado), a seleção ocorre nas colunas da tabela de entrada que tenham a mesma ordem da coluna da tabela objeto. Para o modificador `-S` (seqüência), a exigência é que a seqüência da tabela objeto tem que existir na tabela de entrada.

O parâmetro `NOT` serve para produzir a operação negada, ou seja, a coluna da tabela de entrada é

selecionada, caso a condição do `INCLUDED` e seus modificadores não seja válida.

Por fim, o resultado da seleção é armazenado em uma tabela existente ou criada pelo comando. Essa tabela serve como resultado para uma análise ou como entrada ou objeto de um outro comando SELECT .

5.4.5 Aplicação do comando SELECT

Através do exemplo da Agência de Viagem, esta sub-seção mostra a aplicação do comando SELECT para a análise e verificação de propriedades de um programa Pi-Calculus derivado de um processo BPEL.

O seguinte comando verifica que o programa não tem *deadlock*:

```
SELECT ALL FROM xtable WHERE (//status/ok) IS INCLUDED
```

O programa *Shell* imprime o número de colunas selecionadas. Como esse número é igual ao número de colunas de `xtable` , então todas as colunas têm o *status* OK , não existindo portanto *deadlock*.

Pode-se verificar que em todos os fluxos possíveis o processo responde (envia *reply* para o *web service* que fez o pedido):

```
SELECT ALL FROM xtable WHERE (//ws/rep/cli/cliPT/maktrvarr) IS INCLUDED
```

De novo, basta verificar que o número de colunas selecionadas é igual ao da `xtable`. Outro tipo de teste tem como objetivo assegurar que sempre ocorra uma reserva em uma das companhias aéreas. A idéia aqui é eliminar sucessivamente, a partir da `xtable` , as ocorrências dessas atividades. Assim, se o resultado final for uma tabela com 0 colunas, é porque sempre ocorre a reserva. A seqüência de comandos abaixo comprova a propriedade:

```
SELECT ALL FROM xtable WHERE (//ws/inv/BA/resPT/makres) NOT INCLUDED semBA
SELECT ALL FROM semBA WHERE (//ws/inv/AA/resPT/makres) NOT INCLUDED semBAAA
SELECT ALL FROM semBAAA WHERE //ws/inv/AC/resPT/makres) NOT INCLUDED
semBAUAAC
```

Outro exemplo de propriedade que pode ser verificada é: uma vez que o processo respondeu (enviou *reply*), ele não responde outra vez. Já se sabe que o processo sempre responde. O comando SELECT abaixo obtém todas colunas que têm dois comandos *reply*. Como o número de colunas retornado é 0, isso é verdadeiro.

```
SELECT ALL FROM xtable WHERE (//ws/rep/cli/cliPT/maktrvarr,
//ws/rep/cli/cliPT/maktrvarr) INCLUDED-O
```

Pode-se mostrar que, sempre que a atividade correspondente à atribuição 1 (*assign1*) é executada, então a atividade 2 (*assign2*) é também executada e, além disso, sempre é executada depois. Da seqüência de comandos abaixo, o primeiro seleciona em `assign1` todas as colunas que têm a atribuição 1. O segundo comando obtém o resultado do comando anterior e seleciona as colunas que têm as duas atribuições e estas se encontram ordenadas (modificador `-O`).

```
SELECT ALL FROM xtable WHERE (//act/assign1) IS INCLUDED assign1
SELECT ALL FROM assign1 WHERE (//act/assign1,//act/assign2) IS
INCLUDED-O
```

O resultado esperado, e obtido, é que o número de colunas do último comando seja igual ao número de colunas de `assign1`.

5.5 Um exemplo avançado

Koshkina (19) apresenta mais um exemplo em que conceitos adicionais são aplicados. O programa BPEL não é apresentado.

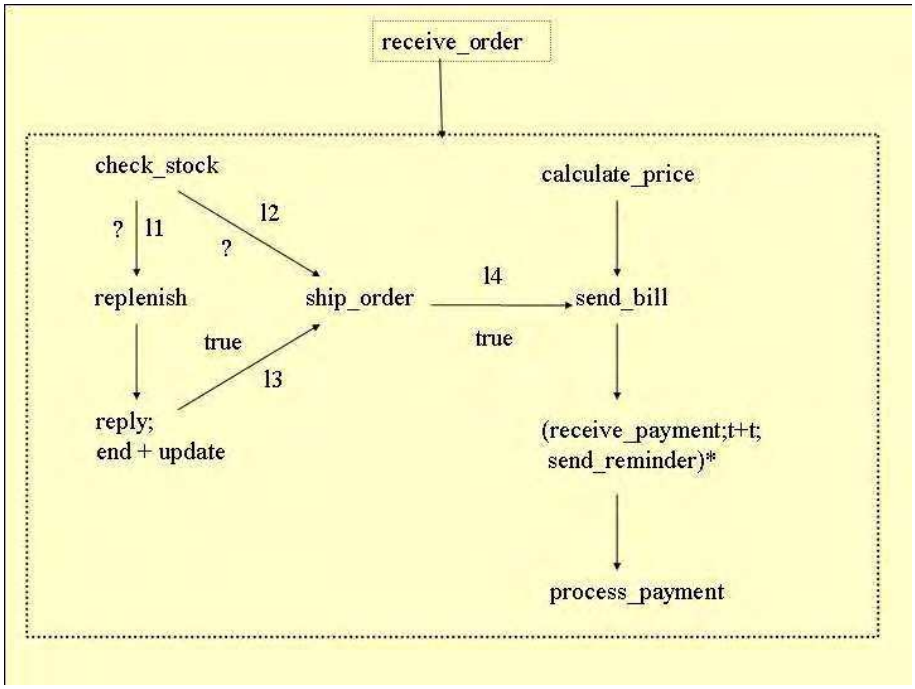

Fig. 5.6: Processo da Livraria.

O processo é iniciado com o pedido de compra do livro (*receive-order*). Paralelamente, são ativados dois fluxos, um para verificar se o livro existe no estoque (*check-stock*) e outro para o cálculo do preço (*calculate-price*).

Se o livro existe no estoque, então ele é enviado´para o cliente (*ship-order*). Após a execução do serviço de envio e do cálculo do preço, é enviada a fatura (*send-bill*). O processo espera até que o pagamento seja recebido (*receive-payment*). Se isso não acontece após um determinado tempo, é enviado um lembrete para o comprador (*send-remainder*). Após o recebimento do pagamento, este é processado (*process-payment*) e o processo termina.

Se o livro não está em estoque, então é feita uma tentativa de repor o estoque (*replenish*). Se a tentativa é bem sucedida, então o livro é enviado (*ship-order*) para o cliente. Se a tentativa falha, uma resposta é enviada notificando que o pedido não pode ser executado. O processo então termina.

As ligações estão representadas por l1, l2, l3, l4. Os links l1 e l2 geram valores que podem ser *verdadeiro* ou *falso*, após a execução das atividades correspondentes, enquanto que l3 e l4 geram sempre valores verdadeiros.

O apêndice D apresenta a expressão Pi-Calculus desse processo.

A Tabela de Execução inicial (*xtable*) neste exemplo tem 18 colunas. Através do uso do Analisador, efetua-se a análise do processo.

O propósito do comando abaixo é verificar se é possível enviar o livro depois de executar uma atividade da reposição do estoque (*replenish*):

```
SELECT ALL FROM xtable WHERE
(//ws/inv/replenish,//ws/inv/shipOrder) INCLUDED-O
```

O resultado é uma tabela com 8 colunas. Se não se usar o modificador `-O`, o resultado obtido também contém 8 colunas. Isso mostra que a ordem de envio sempre ocorre depois da reposição bem-sucedida do estoque.

O próximo passo é mostrar que o processo que foi desenhado satisfaz os requisitos mínimos, ou seja, atende às seguintes seqüências:

```
receive-order.check-order.ship-order.send-bill.receive-payment.process-payment
```

e

```
receive-order.calculate-price.ship-order.send-bill.receive-payment.process-payment
```

Utiliza-se um comando SELECT para a análise de cada seqüência:

```
SELECT ALL FROM xtable WHERE
(//ws/rec/receiveOrder,//ws/inv/checkStock,//ws/inv/shipOrder,
//ws/inv/sendBill,//ws/rec/receivePayment,//ws/inv/processPayment)
INCLUDED-O

SELECT ALL FROM xtable WHERE
```

```
(//ws/rec/receiveOrder,//ws/inv/calculatePrice,//ws/inv/shipOrder,
//ws/inv/sendBill,//ws/rec/receivePayment,//ws/inv/processPayment)
INCLUDED-O
```

No primeiro comando, são selecionadas 14 colunas. Ao executar o mesmo comando com parâmetro `NOT INCLUDED`, obtém-se uma tabela com 4 colunas, que são exatamente as que terminam com a resposta de que não há livro no estoque, mesmo depois da reposição. A figura 5.7 mostra essa tabela através da interface gráfica.

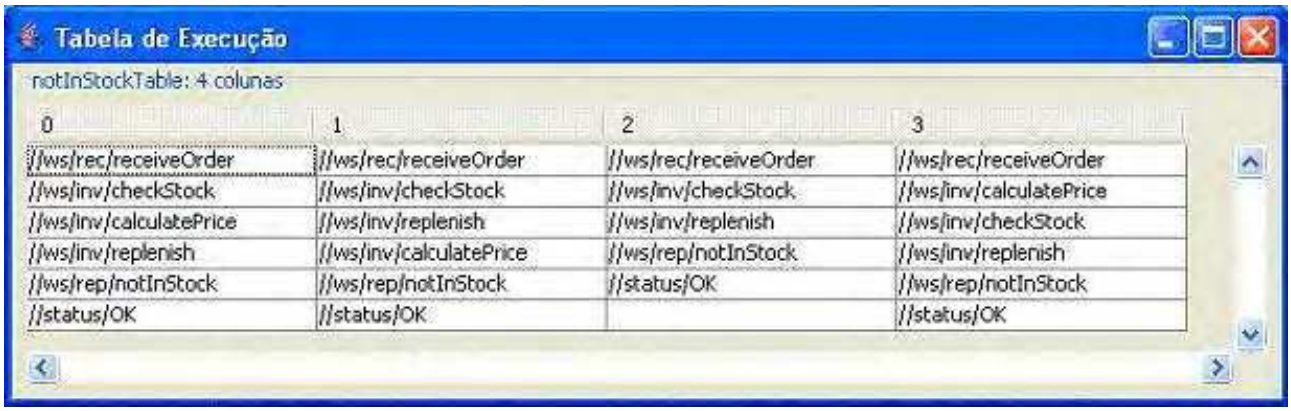

Fig. 5.7: Tabela de Execução do processo da Livraria

Do segundo comando, obtém-se apenas 10 colunas. E aí pode surgir a dúvida: por quê não são 14 colunas igualmente ao caso anterior? A análise segue com os seguintes comandos:

```
SELECT ALL FROM xtable WHERE
(//ws/rec/receiveOrder,//ws/inv/calculatePrice,//ws/inv/shipOrder,
//ws/inv/sendBill,//ws/rec/receivePayment,//ws/inv/processPayment)
NOT INCLUDED-O y1
```

O resultado, como esperado, é uma tabela com 8 colunas, das quais devem estar incluídas as 4 acima mencionadas. O comando para eliminar essas 4 colunas é:

```
SELECT ALL FROM y1 WHERE (//ws/rep/notInStock) NOT INCLUDED-O y2
```

A tabela `y2` , resultado do comando, tem 4 colunas. Listando-se essa tabela, vê-se que a ordem das chamadas `//ws/inv/calculatePrice` e `//ws/inv/shipOrder` está trocada. Ao considerar a especificação do problema, constata-se que isso pode ocorrer, pois não existe nenhum requisito quanto à seqüência de execução desses dois serviços.

Para terminar, é apresentado um exemplo em que ocorre *deadlock*. Para criar essa situação, é definida uma ligação l5, que se origina no processamento do pagamento *(process-payment)* e termina

no envio do livro (*ship-order*). A condição dessa ligação é sempre *verdadeira*. A expressão *joinCondition* da chamada ao serviço de envio do livro (*ship-order*) é a função *And* entre essa ligação (l5) e um *Or* entre as ligações l2 e l3.

O resultado obtido indica que alguns caminhos de execução terminam em *deadlock*. A *xtable* desse programa é listada a seguir:

```
[0,0] //ws/rec/receiveOrder
[0,1] //ws//inv/checkStock
[0,2] //ws/inv/calculatePrice
[0,3] //ws/inv/replenish
[0,4] //ws/rep/notInStock
[0,5] //status/OK

[1,0] //ws/rec/receiveOrder
[1,1] //ws//inv/checkStock
[1,2] //ws/inv/calculatePrice
[1,3] //ws/inv/replenish
[1,4] //status/deadlock/sendBill&trueSO#3&falseSO#4&paid&true3#5&false3#6

[2,0] //ws/rec/receiveOrder
[2,1] //ws//inv/checkStock
[2,2] //ws/inv/calculatePrice
[2,3] //status/deadlock/sendBill&trueSO#3&falseSO#4&paid&true3#5&false3#6

[3,0] //ws/rec/receiveOrder
[3,1] //ws//inv/checkStock
[3,2] //ws/inv/replenish
[3,3] //ws/inv/calculatePrice
[3,4] //ws/rep/notInStock
[3,5] //status/OK

[4,0] //ws/rec/receiveOrder
[4,1] //ws//inv/checkStock
[4,2] //ws/inv/replenish
[4,3] //ws/inv/calculatePrice
[4,4] //status/deadlock/sendBill&trueSO#3&falseSO#4&paid&true3#5&false3#6

[5,0] //ws/rec/receiveOrder
[5,1] //ws//inv/checkStock
[5,2] //ws/inv/replenish
[5,3] //ws/rep/notInStock
[5,4] //status/OK
```

```
[6,0] //ws/rec/receiveOrder
[6,1] //ws/inv/calculatePrice
[6,2] //ws//inv/checkStock
[6,3] //ws/inv/replenish
[6,4] //ws/rep/notInStock
[6,5] //status/OK

[7,0] //ws/rec/receiveOrder
[7,1] //ws/inv/calculatePrice
[7,2] //ws//inv/checkStock
[7,3] //ws/inv/replenish
[7,4] //status/deadlock/sendBill&trueSO#3&falseSO#4&paid&true3#5&false3#6

[8,0] //ws/rec/receiveOrder
[8,1] //ws/inv/calculatePrice
[8,2] //ws//inv/checkStock
[8,3] //status/deadlock/sendBill&trueSO#3&falseSO#4&paid&true3#5&false3#6
```

Observando a tabela anterior, conclui-se que fora os casos em que o processo chega ao fim pela inexistência do livro no estoque (`//ws/rep/notInStock`), as demais seqüências terminam em *deadlock*.

Também é possível verificar a situação de *deadlock* de outra maneira, através da Interface Gráfica. A figura 5.8 mostra que existe um tipo de *deadlock* e que este ocorre em vários nós da Árvore de Execução:

A figura 5.9 mostra um caminho em *deadlock* e a expressão final Pi-Calculus deste caminho.

5.6 Exemplo em Processos de Telecomunicações: Gerência de SLA

O Telemanagement Forum (TMForum) é um consórcio internacional de provedores de serviços de telecomunicações e fornecedores de equipamentos e sistemas, que visa automatizar seus processos, atuando em diferentes vertentes, que inclui a definição de um modelo de referência dos processos de negócios. Esse modelo de referência é conhecido como *enhanced Telecom Operation Map* (eTOM).

Em uma situação prática, há a necessidade de se gerenciar os processos fim-a-fim, que são específicos a cada objetivo de negócio de um provedor de serviço. Esses processos fim-a-fim são construídos através de fluxos que utilizam os processos do modelo de referência.

O apêndice F contém um exemplo do desenvolvimento e análise de um processo de Gerência de

Fig. 5.8: Lista de *deadlocks* do processo da Livraria

SLA (*Service Level Agreement*), cuja descrição foi obtida do próprio TMForum (36). Nesse documento, o processo de Gerência de SLA contém três fluxos:

- fluxo normal de execução a partir de dados de performance e de uso;
- fluxo para o caso de alerta de ultrapassagem de limite ou alarme de serviço;
- execução com violação de SLA

Os fluxos são descritos através de interações entre os processos do modelo de referência. No desenvolvimento do exemplo, esses fluxos são considerados como a especificação do problema.

Com o objetivo de simplificar a quantidade de interações entre os processos, são definidos macro-processos de gerência que encapsulam um ou mais processos do modelo de referência. Dessa forma, as interações internas a um macro-processo são eliminadas. A partir dessa simplificação, os cenários de interações entre os processos dos três fluxos são reescritos contendo somente as interações entre os macro-processos.

No passo seguinte, os *casos de uso* do processo fim-a-fim (Gerência de SLA) são construídos. Cada fluxo corresponde a um caso de uso. Foi convencionado que cada macro-processo é implementado por um sistema que disponibiliza interfaces *web services* para cada tipo de interação, e, que o processo de Gerência de SLA é implementado por um BPMS (*Business Process Management System*) que interpreta os processos escritos em BPEL.

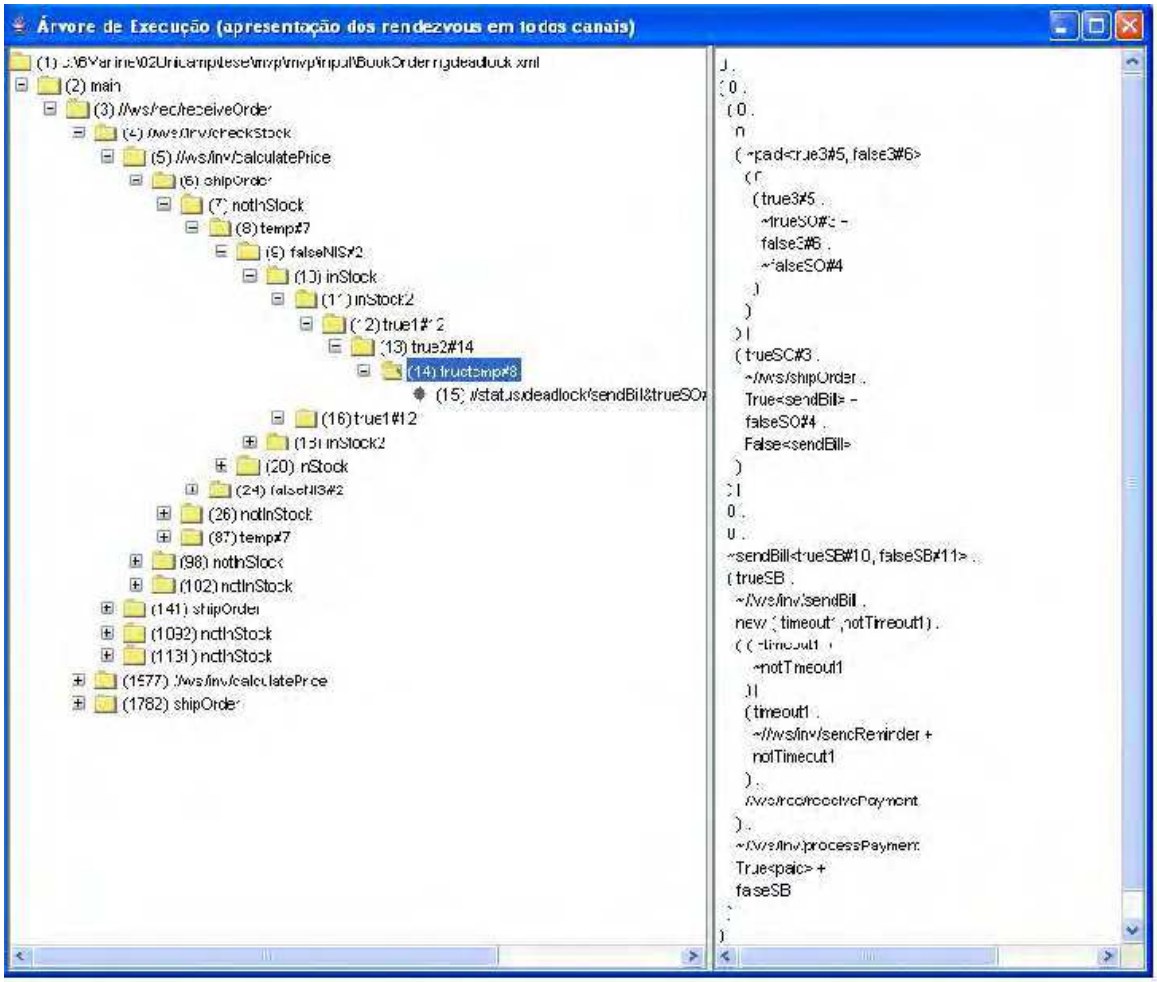

Fig. 5.9: Árvore de Execução do Processo da Livraria em *deadlock*

Usando os casos de uso e a lista de *web services* dos sistemas que implementam os macro-processos, o processo de Gerência de SLA é desenvolvido na linguagem BPEL. O passo seguinte é traduzir esse processo para Pi-Calculus e executá-lo na Máquina Virtual para gerar a Árvore de Execução e a Tabela de Execução *xtable* que são utilizadas na análise do processo e verificação de propriedades.

Para ilustrar alguns tipos de verificação que podem ser feitas pela Máquina Virtual, foram realizados três tipos de análise: análise de alto nível, em que se formula os predicados a partir das especificações do problema; análise para otimização, que procura detectar padrões de seqüência de chamadas que pode ser implemntada em uma única chamada; e por fim, o processo é modificado para gerar uma situação de *deadlock* e, a partir daí, mostrar como a Máquina Virtual detecta e apresenta essa condição.

5.7 Conclusão

Este capítulo descreveu o protótipo da Máquina Virtual Pi-Calculus desenvolvido para a análise de programas Pi-Calculus. O propósito na construção desse protótipo é validar a proposta de análise e verificação de processos BPEL através do uso de Pi-Calculus. Uma ferramenta desse tipo pode ser aplicada a um *framework* que suporta o ciclo de vida de um processo de negócio. Essa ferramenta seria utilizada na fase de testes de um processo.

O próximo capítulo apresenta as conclusões e algumas indicações de trabalhos futuros.

Capítulo 6

Conclusão e Trabalhos Futuros

Esta dissertação apresentou a proposta de um *framework* para a análise e verificação de processos BPEL através do uso de uma Máquina Virtual Pi-Calculus. Um *framework* desse tipo poderá ser agregado aos sistemas que suportam o ciclo de vida dos processos, de modo que certas propriedades dos processos de negócios sejam verificadas antes que estes sejam implantados. Para a validação da proposta, foi desenvolvido um protótipo de uma Máquina Virtual Pi-Calculus (MVP).

A análise aqui formulada ressalta o fluxo de controle e as interações com os *web services*, a partir da geração de todos os caminhos de execução que um processo pode seguir.

Nesse *framework*, um processo BPEL é traduzido para Pi-Calculus conforme as regras definidas no capítulo 4. Essas regras levam em consideração que a MVP gera todas as reações possíveis do programa Pi-Calculus. O programa é executado pela MVP e o resultado é a Árvore de Execução contendo todos os caminhos possíveis. A Interface Gráfica e a Console da MVP têm comandos para listar a Árvore de Execução permitindo analisar a execução passo-a-passo do programa através da visualização dos canais em que ocorrem o *rendez-vous* e a respectiva expressão Pi-Calculus resultante.

A partir da Árvore de Execução, cria-se a Árvore-X que contém somente os nós correspondentes aos *rendez-vous* nos canais externos. A Árvore-X é convertida para uma Tabela de Execução, denominada *xtable*, que é o ponto de partida do módulo de Análise. Esse módulo, por sua vez, possui um comando para a análise do processo, que permite verificar: se os requisitos dos processos estão sendo atendidos pelo processo BPEL, a ordenação de eventos e propriedades relacionadas a *liveness*, *safety* e *deadlock*.

Para mostrar os benefícios de se usar uma ferramenta desse tipo, foram desenvolvidos alguns exemplos, entre os quais, destaca-se o processo de Gerência de SLA do apêndice F, que possui uma aplicação e uma complexidade próximas ao mundo real. Nesse exemplo, constatou-se que a ferramenta pode ser utilizada para a verificação da implementação de propriedades da especificação do problema e propriedades de fluxos, como a verificação de existência, ou não, de *deadlocks*.

Com relação aos trabalhos futuros na linha desta tese, visualizam-se as seguintes tarefas:

- incluir escopo e os tratamentos de falha, eventos e compensação, que são as construções do BPEL que estão faltando para a cobertura de toda a linguagem;
- desenvolver um tradutor BPEL para Pi-Calculus - esse tradutor deve ser flexível de maneira a permitir que o usuário escolha as atividades que deseja levar para a fase de Análise, ou seja, traduzir essas atividades para canais externos;
- construir um simulador de execução de programa Pi-Calculus. Esse simulador pode também ser utilizado como uma ferramenta para testes. Nesse caso, as regras de mapeamento devem ser adaptadas e o bloco funcional *Gerador de Seqüências* deve ser substituído por um outro bloco que implemente o simulador (o módulo Processador, que é o mais complexo, seria mantido). As funções de manipulação de dados podem ser implementadas como uma entidade externa que se comunica através de canais externos específicos;
- modelar a Árvore de Execução e da Tabela de Execução em um banco de dados permitindo desenvolver ferramentas mais elaboradas para a análise dos programas;
- ampliar o conjunto de ferramentas de modo a fazer as traduções no sentido reverso, ou seja, dado um programa Pi-Calculus que representa a solução de um processo, traduzi-la para um esqueleto BPEL, de forma análoga à descrita por Salaün (31). Com o *framework* trabalhando nos dois sentidos, o suporte ao ciclo de vida do desenvolvimento do processo tornar-se-ia mais abrangente;
- criar uma Rede Virtual Pi-Calculus (RVP) constituída de várias Máquinas Virtuais, em que cada máquina executa um processo escrito em BPEL. Os *rendez-vous* dos canais externos correspondentes às operações *web services* passam a ser controlados por um gerenciador da rede. Algumas das regras de tradução devem ser modificadas para o tratamento diferenciado entre comunicação síncrona e assíncrona entre os *web services*. Uma ferramenta desse tipo tem como vantagem a maior abrangência das verificações estabelecidas, cobrindo o interrelacionamento entre os processos de uma organização e não simplesmente a análise de um processo de forma isolada. Em um instante seguinte, pode-se incluir também não só os *web services* descritos em BPEL mas aqueles que são implementados em outros ambientes e que têm suas coreografias descritas em WS-CDL ou outra linguagem padrão. Nesse caso, a coreografia é convertida para um programa Pi-Calculus, que é ativado em uma MVP que participa da RVP, ampliando a cobertura de verificações dos processos de negócio de uma corporação.

Referências Bibliográficas

[1] W. Aalst, A. Hofstede e M. Weske. *Business Process Management: A Survey*. Lecture Notes in Computer Science - Springer-Verlag, junho 2003.

[2] T. Andrews, F. Curbera, H. Dholakia, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic e S. Weerawarana. *Business Process Execution Language for Web Services - version 1.1*. Technical report, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems, maio 2003.

[3] J. Bowen. *comp.specification.Z and Z Forum - Frequently Asked Questions*. Documento diponível em http://www.jpbowen/pub/faq98.pdf, em 20/02/2006, 1998.

[4] T. Bray, J. Paoli, C. Sperberg-McQueen e E. Maler. *Extensible Markup Language (XML 1.0) (second edition)*. Technical report, W3C Recommendation, disponível em http://www.w3c.org/TR/REC-xml, outubro 2000.

[5] A. Brogi, C. Canal, E. Pimentel e A. Vallecillo. *Formalizing Web Service Choreographies*. Documento diponível em http://www.elsevier.nl/locate/entcs, 2003.

[6] R. Chinnici, M. Gudgin, J.-J. Moreau e S. Weerawarana. *Web Services Description Language, version 1.2*. Technical report, W3C Recommendation, disponível em http://www.w3c.org/TR/wsdl12, julho 2002.

[7] J. Clark e S. DeRose. *XML Path Language (XPath) version 1.0*. Technical report, W3C Recommendation, disponível em http://www.w3c.org/TR/xpath, 1999.

[8] R. Cleveland, J. Parrow e B. Stefeen. *The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems*. ACM Transaction on Programming Languages and Systems, 15(1):36–72, janeiro 1993.

[9] F. Curbera, R. Khalat, W. Nagy e S. Weerawarana. *Implementing BPEL4WS: The Architecture of a BPEL4WS Implementation*. Technical report, IBM T. J. Watson Research Center - documento extraído de http://www.research.ibm.com/people/r/rkhalaf/ImplementingBPEL.pdf em 14/10/2006, 2005.

[10] J. Dessel e G. Juhás. *What Is a Petri Net? - Informal Answers for the Informed Reader*. Em Hartmut Ehrig et. al., editor, Unifying Petri Nets, pp. 1–25, 2001.

[11] T. Erl. *Service-Oriented Architecture - Concepts, Technology and Design*. Prentice Hall, 2005.

[12] D.C. Fallside. *XML Schema Part 0: Primer*. Technical report, W3C Working Draft, disponível em http://www.w3c.org/TR/xmlschema-0, julho 2002.

[13] A. Ferrara. *Web Services: a process algebra approach*. Em ACM Press, editor, 2nd Internacional Conference on Service Oriented Computing, pp. 242–251, New York, NY, USA, 2004.

[14] H. Foster. *Behaviour Analysis and Verification of Web Service Composition*. Technical report, Imperial College London, University of London, Department of Computing, maio 2004.

[15] H. Foster, S. Uchitel, J. Magee e J. Kramer. *Model-based Verification of Web Service Compositions*. Em 18th IEEE Conference on Automated Software Engineering (ASE), Montreal, Canada, outubro 2003.

[16] ITU-T. *M.3010: Principle of Telecommunication Network Management, 2000*. Technical report, International Telecommunication Union - Telecommunication Standardization Sector, 2000.

[17] R. High Jr, S. Kindar e S. Graham. *IBM´s SOA Foundation - An Architectural Introduction and Overview - version 1.0*. Technical report, IBM, novembro 2005.

[18] T. Kornel. *OSS Essentials*. John Willey and Sons Inc., 2001.

[19] M. Koshkina. *Verificaton of Business Process for Web Services*. Technical report, Department of Computer Science. York University, Toronto, Ontario, outubro 2003.

[20] M. Koshkina e F. Bruegel. *Modelling and Verifying Web Service Orchestration by means of the Concurency Workbench*. Em Proceedings of the Workshop on Testing, Analysis and Verifivation of Web Services (TAV-WEB), ACM SIGSOFT. Software Engineering Notes, 29(5), setembro 2004.

[21] N. Koventas, D. Burdett, G. Ritzinger, T. Fletcher e Y. Lafon. *Web Services Coreoghraphy Desciption Language Version 1.0*. Technical report, W3C Recommendation, disponível em http://www.w3c.org/TR/2004/WD-ws-cdl-10-20041012/, 2004.

[22] P. Maciel, R. Lins e P. Cunha. *Introdução às Redes de Petri e Aplicações*. Décima Escola de Computação - Instituto de Computação da Universidade Estadual de Campinas, Campinas, São Paulo, 1996.

[23] R. Milner. *Communicating and Mobile Systems: the pi-calculus*. Cambridge University Press, 1999.

[24] D. Nickull. *An Introduction to the OASIS Reference Model for Service Oriented Architecture (SOA)*. Documento disponível em http://www.oasis-open.org/committees/tc-home.phg?wg-abbrev=soa-rm em 08/05/2006, 2006.

[25] Oracle. *Closed Loop BPM using standards based tools - An Oracle White Paper*. Technical report, Oracle, novembro 2004.

[26] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede e E. Verbeek. *Formal Semantics and Analysis of Control Flow in WS-BPEL.* Technical report, BPMCenter - documento extraído de http://www.bpmcenter.org em 08/05/2006, 2005.

[27] J. Pasley. *How BPEL and SOA are changing Web Services Development*. IEEE Internet Computing, maio-junho 2005.

[28] PMG. *Workshop Proceedings*. Technical report, Process Modeling Group - Technische Universiteit Eindhoven, disponível http://www.process-modeling-group.org, junho 2005.

[29] F. Puhlmann. *Why do we actually need the Pi-Calculus for Business Process Management?* Technical report, Business Process Technology Group. Hasso-Plattner-Institute for IT Systems at the University of Potsdam, Germany, 2005.

[30] F. Puhlmann e M. Weske. *Using the Pi-Calculus for Formalizing Workflow Patterns*. Lecture Notes in Computer Science - Springer-Verlag, 3649:153–168, 2005.

[31] G. Salaün, L. Bordeaux e M. Scherf. *Describing and Reasoning on Web Services using Process Algebra*. Technical report, DIS - Università di Roma "La Sienza", Via Salaria 113, 00198 Roma, Itália, 2004.

[32] Y. Sin, C. Wang, L.Gilbert e G. Wills. *An Overview of Service Oriented Architecture*. Technical Report ECSPR-IAM05-004, School of Electronics and Computer Science - University of Southampton, julho 2005.

[33] T. Smith e P. Fingar. *Business Process Management: the third wave*. Meghan-Kiffer Press, 2003.

[34] J. Spivey. *The Z Notation. A Reference Manual - Second Edition*. Oriel College, Oxford OX14EW, 1998.

[35] D. Sprott. *Service Oriented Architecture: An Introduction for Managers*. Technical report, CDBI, 2004.

[36] TMForum. *GB921F: Enhanced Telecom Operation Map (eTOM) The Business Process Framework, Addendum F: Process Flow Examples - Release 4.5*. Technical report, Telemanagement Forum, 2004.

[37] W.M.P. van der Aalst. *Pi Calculus versus Petri nets: Let us eat "humble pie"rather than further inflate the "Pi hype"*. Technical report, Department of Technology Management, Eindhoven University of technology, Netherlands, 2004.

[38] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski e A. Barros. *Workflow patterns*. Distributed and Parallel Databases, (14(1)):5–51, 2003.

[39] P. Wohed, W.M.P. van der Aalst, M. Dumas e A.H.M ter Hofstede. *Analysis of Web services Composition Languages: The Case of BPEL4WS*. Em I.Y. Song Liddic Ling e P. Scheuermann, editor, 22nd Internacional Conference on Conceptual Modeling (ER2003), pp. 200–215, New York, NY, EUA, 2003.

Apêndice A

Esquema da Linguagem da Máquina Virtual Pi-Calculus (MVP-L)

```
<?xml version="1.0" encoding="UTF-8"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

<!-- package --> <xs:element name="package">
  <xs:complexType>
    <xs:sequence>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="external" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element ref="process" minOccurs="1" maxOccurs="unbounded"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
</xs:element>

<!-- external --> <xs:element name="external">
  <xs:complexType>
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="uri" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>

<!-- process -->
 <xs:element name="process">
  <xs:complexType>
    <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="new"/>
          <xs:element ref="sum"/>
          <xs:element ref="parallel"/>
          <xs:element ref="replicate"/>
          <xs:element ref="exec"/>
          <xs:element ref="read"/>
          <xs:element ref="write"/>
```

```
          <xs:element ref="exit"/>
        </xs:choice>
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="required"/>
     <xs:attribute name="parm" type="listOfStrings" use="optional"/>
  </xs:complexType>
 </xs:element>

<!-- block -->
 <xs:element name="block">
  <xs:complexType>
     <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="new"/>
          <xs:element ref="sum"/>
          <xs:element ref="parallel"/>
          <xs:element ref="replicate"/>
          <xs:element ref="exec"/>
          <xs:element ref="read"/>
          <xs:element ref="write"/>
          <xs:element ref="exit"/>
        </xs:choice>
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>

<!-- sum -->
 <xs:element name="sum">
  <xs:complexType>
    <xs:sequence>
        <xs:element ref="block" minOccurs="2" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>

<!-- paralell -->
 <xs:element name="parallel">
  <xs:complexType>
     <xs:sequence>
        <xs:element ref="block" minOccurs="2" maxOccurs="unbounded"/>
     </xs:sequence>
     <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>

<!-- new --> <xs:element name="new">
  <xs:complexType>
      <xs:attribute name="var" type="listOfString" use="required"/>
      <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>
```

```
<!-- replicate --> <xs:element name="replicate">
  <xs:complexType>
   <xs:sequence>
     <xs:element ref="block" minOccurs="1" maxOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>

<!-- read -->
 <xs:element name="read">
  <xs:complexType>
     <xs:attribute name="channel" type="xs:string" use="required"/>
     <xs:attribute name="var" type="listOfStrings" use="optional"/>
     <xs:attribute name="name" type="xs:string" use="optional"/>
   </xs:complexType>
 </xs:element>

<!-- write -->
 <xs:element name="write">
  <xs:complexType>
     <xs:attribute name="channel" type="xs:string" use="required"/>
     <xs:attribute name="value" type="listOfStrings" use="optional"/>
     <xs:attribute name="name" type="xs:string" use="optional"/>
   </xs:complexType>
 </xs:element>

<!-- exec -->
 <xs:element name="exec">
  <xs:complexType>
     <xs:attribute name="process" type="xs:string" use="required"/>
     <xs:attribute name="value" type="listOfStrings" use="optional"/>
     <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>

<!-- exit --> <xs:element name="exit">
  <xs:complexType>
    <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:complexType>
 </xs:element>


<!-- listOfString --> <xs:simpleType name="listOfString">
   <xs:list itemType="xs:string"/>
</xs:simpleType>

</xs:schema>
```

Apêndice B

Programa da Agência de Viagem na Linguagem da MVP

Esse apêndice lista o programa Pi-Calculus correspondente à tradução do processo da Ag encia de Viagem. Os processos básicos, como as funções booleanas que definem os valores *True* e *False* e as operações *And*, *Or* e *Not*, foram omitidos.

```
<?xml version="1.0" ?>

<package xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="pivm.xsd"
     name="TravelAgency">
 <external name="//sys" uri="http://www.unicamp.br/vanine/mvp2/externals/sys" />
 <external name="//ws" uri="http://www.unicamp.br/vanine/mvp2/externals/webservices" />
 <external name="//act" uri="http://www.unicamp.br/vanine/mvp2/externals/bpelactivities'
 <process name="main" >
   <read channel="//ws/rec/cli/cliPT/maktrvarr"/>
   <parallel>
     <block>
       <new var="sw1 sw2 sw3" />
       <parallel>
         <block>
           <sum>
             <block>
               <write channel="sw1" />
             </block>
             <block>
               <write channel="sw2" />
             </block>
             <block>
               <write channel="sw3" />
             </block>
           </sum>
         </block>
         <block>
           <sum>
             <block>
               <read channel="sw1" />
```

```
            <write channel="//ws/inv/AC/resPT/makres" />
            <parallel>
              <block>
                <exec process="True" value="trvcan" />
              </block>
              <block>
                <exec process="False" value="trvus" />
              </block>
            </parallel>
          </block>
          <block>
            <read channel="sw2" />
            <write channel="//ws/inv/AA/resPT/makres" />
            <parallel>
              <block>
                <exec process="False" value="trvcan" />
              </block>
              <block>
                <sum>
                  <block>
                    <exec process="False" value="trvus" />
                  </block>
                  <block>
                    <exec process="True" value="trvus" />
                  </block>
                </sum>
              </block>
            </parallel>
          </block>
          <block>
            <read channel="sw3" />
            <write channel="//ws/inv/BA/resPT/makres" />
            <parallel>
              <block>
                <exec process="False" value="trvcan" />
              </block>
              <block>
                <exec process="False" value="trvus" />
              </block>
            </parallel>
          </block>
        </sum>
      </block>
    </parallel>
  </block>
  <block>
    <write channel="//ws/inv/weasrv/weaPT/getfcs" />
  </block>
  <block>
    <new var="true false out" />
    <parallel>
      <block>
        <write channel="out" value="true false" />
```

```
        </block>
        <block>
          <exec process="Or" value="trvcan trvus out" />
        </block>
        <block>
          <sum>
            <block>
              <read channel="true" />
              <write channel="//ws/inv/carren/carrenPT/rent" />
              <exec process="True" value="ren2ass" />
            </block>
            <block>
              <read channel="false" />
              <exec process="False" value="ren2ass"/>
            </block>
          </sum>
        </block>
      </parallel>
    </block>
    <block>
      <new var="true false" />
      <write channel="ren2ass" value="true false" />
      <sum>
        <block>
          <read channel="true" />
          <write channel="//act/assign1" />
        </block>
        <block>
          <read channel="false" />
        </block>
      </sum>
    </block>
   </parallel>
   <write channel="//act/assign2" />
   <write channel="//ws/rep/cli/cliPT/maktrvarr" />
 </process>
</package>
```

Apêndice C

Árvore de Execução Gerada pelo Processo da Agência de Viagem

```
0 C:\6Vanine\02Unicamp\tese\mvp\mvp\input\TravelAgent.xml
1 main
2 //ws/rec/cli/cliPT/maktrvarr
2.1 //ws/inv/weasrv/weaPT/getfcs
2.1.1 sw1#1 2.1.1.1 out#6
2.1.1.1.1 //ws/inv/AC/resPT/makres
2.1.1.1.1.1 trvcan
2.1.1.1.1.1.1 trvus
2.1.1.1.1.1.1.1 true1#9
2.1.1.1.1.1.1.2 false2#12
2.1.1.1.1.1.1.3 true#4
2.1.1.1.1.1.1.4 //ws/inv/carren/carrenPT/rent
2.1.1.1.1.1.1.5 ren2ass
2.1.1.1.1.1.1.6 true#7
2.1.1.1.1.1.1.7 //act/assign1
2.1.1.1.1.1.1.8 //act/assign2
2.1.1.1.1.1.1.9.0 //ws/rep/cli/cliPT/maktrvarr
2.1.1.1.1.1.2 true1#9
2.1.1.1.1.1.2.1 trvus <==> 2.1.1.1.1.1.1.1
2.1.1.1.1.1.2 trvus
2.1.1.1.1.2.1 trvcan <==> 2.1.1.1.1.1.1
2.1.1.2 //ws/inv/AC/resPT/makres
2.1.1.2.1 out#6 <==> 2.1.1.1.1
2.1.2 sw2#2
2.1.2.1 out#6
2.1.2.1.1 //ws/inv/AA/resPT/makres
2.1.2.1.1.1 trvcan
2.1.2.1.1.1.1 trvus
2.1.2.1.1.1.1.1 false1#10
2.1.2.1.1.1.1.2 false2#12
2.1.2.1.1.1.1.3 false#5
2.1.2.1.1.1.1.4 ren2ass
2.1.2.1.1.1.1.5 false#8 <==> 2.1.1.1.1.1.1.7
2.1.2.1.1.1.2 trvus
```

```
2.1.2.1.1.1.2.1 false1#10
2.1.2.1.1.1.2.2 true2#11 <==> 2.1.1.1.1.1.1.2
2.1.2.1.1.1.3 false1#10
2.1.2.1.1.1.3.1 trvus <==> 2.1.2.1.1.1.1.1
2.1.2.1.1.1.3.2 trvus <==> 2.1.2.1.1.1.2.1
2.1.2.1.1.2 trvus
2.1.2.1.1.2.1 trvcan <==> 2.1.2.1.1.1.1
2.1.2.1.1.3 trvus
2.1.2.1.1.3.1 trvcan <==> 2.1.2.1.1.1.2
2.1.2.2 //ws/inv/AA/resPT/makres
2.1.2.2.1 out#6 <==> 2.1.2.1.1
2.1.3 sw3#3
2.1.3.1 out#6
2.1.3.1.1 //ws/inv/BA/resPT/makres
2.1.3.1.1.1 trvcan
2.1.3.1.1.1.1 trvus
2.1.3.1.1.1.1.1 false1#10
2.1.3.1.1.1.1.2 false2#12 <==> 2.1.2.1.1.1.1.2
2.1.3.1.1.1.2 false1#10
2.1.3.1.1.1.2.1 trvus <==> 2.1.3.1.1.1.1.1
2.1.3.1.1.2 trvus
2.1.3.1.1.2.1 trvcan <==> 2.1.3.1.1.1.1
2.1.3.2 //ws/inv/BA/resPT/makres
2.1.3.2.1 out#6 <==> 2.1.3.1.1
2.1.4 out#6
2.1.4.1 sw1#1 <==> 2.1.1.1
2.1.4.2 sw2#2 <==> 2.1.2.1
2.1.4.3 sw3#3 <==> 2.1.3.1
2.2 sw1#1
2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1
2.2.2 out#6
2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.4.1
2.2.2.2 //ws/inv/AC/resPT/makres
2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.2.1
2.2.2.2.2 trvcan
2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.1
2.2.2.2.2.2 trvus
2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.2.1
2.2.2.2.2.2.2 true1#9
2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.1.2.1
2.2.2.2.2.2.2.2 false2#12
2.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1.2.2
2.2.2.2.2.2.2.2.2 true#4
2.2.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.1.1.3
2.2.2.2.2.2.2.2.2.2 //ws/inv/carren/carrenPT/rent
2.2.2.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.1.1.4
2.2.2.2.2.2.2.2.2.2.2 ren2ass
2.2.2.2.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.1.1.5
2.2.2.2.2.2.2.2.2.2.2.2 true#7
2.2.2.2.2.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.1.1.6
2.2.2.2.2.2.2.2.2.2.2.2.2 //act/assign1
2.2.2.2.2.2.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1.1.5
2.2.2.2.2.3 true1#9
```

```
2.2.2.2.2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.1.2
2.2.2.2.2.3.2 trvus <==> 2.2.2.2.2.2.2
2.2.2.2.3 trvus
2.2.2.2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.1.1.2
2.2.2.2.3.2 trvcan <==> 2.2.2.2.2.2
2.2.3 //ws/inv/AC/resPT/makres
2.2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.1.2
2.2.3.2 out#6 <==> 2.2.2.2
2.3 sw2#2
2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2
2.3.2 out#6
2.3.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.4.2
2.3.2.2 //ws/inv/AA/resPT/makres
2.3.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.2.1
2.3.2.2.2 trvcan
2.3.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1
2.3.2.2.2.2 trvus
2.3.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.2.1
2.3.2.2.2.2.2 false1#10
2.3.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1.3.1
2.3.2.2.2.2.2.2 false2#12
2.3.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.1.1.1.1.2
2.3.2.2.2.2.2.2.2 false#5
2.3.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1.1.3
2.3.2.2.2.2.2.2.2.2 ren2ass
2.3.2.2.2.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1.1.4
2.3.2.2.2.2.2.2.2.2.2 false#8 <==> 2.2.2.2.2.2.2.2.2.2.2
2.3.2.2.2.3 trvus
2.3.2.2.2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.3.1
2.3.2.2.2.3.2 false1#10
2.3.2.2.2.3.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1.3.2
2.3.2.2.2.3.2.2 true2#11 <==> 2.2.2.2.2.2.2.2
2.3.2.2.2.4 false1#10
2.3.2.2.2.4.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.1.3
2.3.2.2.2.4.2 trvus <==> 2.3.2.2.2.2.2
2.3.2.2.2.4.3 trvus <==> 2.3.2.2.2.3.2
2.3.2.2.3 trvus
2.3.2.2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.2
2.3.2.2.3.2 trvcan <==> 2.3.2.2.2.2
2.3.2.2.4 trvus
2.3.2.2.4.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.1.1.3
2.3.2.2.4.2 trvcan <==> 2.3.2.2.2.3
2.3.3 //ws/inv/AA/resPT/makres
2.3.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.2.2
2.3.3.2 out#6 <==> 2.3.2.2
2.4 sw3#3
2.4.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3
2.4.2 out#6
2.4.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.4.3
2.4.2.2 //ws/inv/BA/resPT/makres
2.4.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.2.1
2.4.2.2.2 trvcan
2.4.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.1.1.1
```

```
2.4.2.2.2.2 trvus
2.4.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.1.1.2.1
2.4.2.2.2.2.2 false1#10
2.4.2.2.2.2.2.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.1.1.1.2.1
2.4.2.2.2.2.2.2 false2#12 <==> 2.3.2.2.2.2.2.2
2.4.2.2.2.3 false1#10
2.4.2.2.2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.1.1.1.2
2.4.2.2.2.3.2 trvus <==> 2.4.2.2.2.2.2
2.4.2.2.3 trvus
2.4.2.2.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.1.1.2
2.4.2.2.3.2 trvcan <==> 2.4.2.2.2.2
2.4.3 //ws/inv/BA/resPT/makres
2.4.3.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.3.2
2.4.3.2 out#6 <==> 2.4.2.2
2.5 out#6
2.5.1 //ws/inv/weasrv/weaPT/getfcs <==> 2.1.4
2.5.2 sw1#1 <==> 2.2.2
2.5.3 sw2#2 <==> 2.3.2
2.5.4 sw3#3 <==> 2.4.2
```

Apêndice D

Expressão Pi-Calculus do Processo da Compra de Livro

O programa Pi-Calculus derivado do processo da compra de livro foi construído na linguagem XML da MVP e carregado na memória. A listagem abaixo é a expressão listada pela MVP antes do programa ser iniciado. Os comentários inseridos têm o objetivo de facilitar o entendimento do programa.

```
%% -- início do programa

main==>

%% -- recebe o pedido de compra;

//ws/rec/receiveOrder .

%% -- executa a verificaçao de estoque;

( ~//ws/inv/checkStock .

%% -- gera valores para as duas ligações: l1 (inStock) e l2(notInStock);

  ( ( True<inStock> |
      False<notInStock>
    ) +
    ( False<inStock> |
      True<notInStock>
    )
  ) |

%% -- em paralelo à geração de valores para as ligações,
%% -- espera a ligação notInStock receber valor;

  new ( trueNIS,falseNIS)) .
  ~notInStock<trueNIS,falseNIS> .

%% -- o valor recebido indica que foi executado o processo True para esse link;
```

```
  ( trueNIS .

%% ---- executa o replanejamento (replenish);

    ~//ws/inv/replenish .

%% ---- o resultado do replanejamento pode ser positivo(sw2) ou negativo(sw1);
%% ---- no primeiro caso, é definido o valor True para o link l3(inStock2);
%% ---- no segundo caso, envia uma resposta ao pedido de compra que não existe
%% ---- no estoque e termina o programa;

    new ( sw1,sw2)) .
    ( ( ~sw1 +
        ~sw2
      ) |
      ( sw1 .
        ~//ws/rep/notInStock .
        exit +
        sw2 .
        True<inStock2>
      )
    ) +

%% -- o valor recebido indica que foi executado o processo False para o link
%% -- notInStock -
%% ---- define o valor True para o link l3(inStock2) - regra de atividade
%% ---- estruturada do BPEL;

    falseNIS .
    True<inStock2>
  ) |

%% -- em paralelo à verificação do estoque, espera as ligações inStock e
%% -- inStock2 rebeberem valores e executa um processo (função lógica) Or
%% -- colocando o resultado na ligação shipOrder;

  new ( trueSO,falseSO)) .
  ( ~shipOrder<trueSO,falseSO> |
    Or<inStock, inStock2, shipOrder> |

%% ---- o resultado em shipOrder é equivalente ao processo True -
%% ---- envia a ordem de serviço e atribui o processo True para a ligação
%% ---- sendBill;

    ( trueSO .
      ~//ws/inv/shipOrder .
      True<sendBill> +

%% ---- o resultado em shipOrder é equivalente ao processo False -
%% ---- atribui o processo False para a ligação sendBill;

      falseSO .
      False<sendBill>
```

```
    )
  ) |

%% -- em paralelo À verificação de estoque, calcula o preço independente
%% -- de qq ligação;

  ~//ws/inv/calculatePrice .

%% -- em seqüência, espera a ligação sendBill receber valor;

  new ( trueSB,falseSB)) .
  ~sendBill<trueSB,falseSB> .

%% ---- o resultado em sendBill é equivalente ao processo True -
%% ---- envia a ordem de pagamento e aloca canais para representarem
%% ---- as condições de timeout e não timeout;

  ( trueSB .
    ~//ws/inv/sendBill .
    new ( timeout1,notTimeout1)) .

%% ------ gera duas condições em paralelo: a primeira representa que não
%% ------ ocorre o timeout e portanto não há nada a fazer; já a segunda
%% ------ corresponde à ocorrência do timeout - nesse caso executa o
%% ------ web service sendReminder (lembrar o pagamento); em seqüência
%% ------  a qualquer um dos casos, executa o web service processPayment;

    ( ( ~timeout1 +
        ~notTimeout1
      ) |
      ( timeout1 .
        ~//ws/inv/sendReminder +
        notTimeout1
      ) .
      //ws/rec/receivePayment
    ) .
    ~//ws/inv/processPayment +

%% ---- o resultado em sendBill é equivalente ao processo False -
%% ---- na há nada a fazer - esse caso não deve ocorrer;

    falseSB
  )
)

%% -- fim do processo.
```

Apêndice E

Verificação de Expressões Lógicas através da Máquina Virtual Pi-Calculus

Este apêndice contém a verificação das expressões lógicas básicas construídas no capítulo 3 através da Máquina Virtual Pi-Calculus (MVP). A expressão Pi-Calculus usada para a verificação da função *And* de duas variáveis é definida a seguir:

```
main==>
 new ( Aport,Bport,Zport) . new ( trueZ,falseZ) .
 ( ( ~//sys/print/A/FALSE .
    False<Aport> +
    ~//sys/print/A/TRUE .
    True<Aport>
  ) |
  ( ~//sys/print/B/FALSE .
    False<Bport> +
    ~//sys/print/B/TRUE .
    True<Bport>
  ) |
  And<Aport, Bport, Zport> |
  ~Zport<trueZ,falseZ> .
  ( trueZ .
    ~//sys/print/Z/TRUE +
    falseZ .
    ~//sys/print/Z/FALSE
  )
)
```

A construção dessa expressão considerou duas características da MVP: a primeira é que esta gera todos caminhos de execução possíveis de um programa e a segunda é o fato que a tabela de execução gerada (*xtable*) contém todas as operações nos canais externos.

O uso da primeira característica aparece na geração dos valores de entrada - para cada canal, existe uma soma em que os termos são as ativações das funções complementares `True` e `False` .

A segunda característica foi aplicada para construir a *xtable* contendo operações em canais externos que expressam os parâmetros passados nas funções de entrada e o valor do resultado obtido.

Cada processo `True` ou `False` chamado para fornecer as entradas à função lógica, é precedido da execução de uma operação em um canal externo com prefixo `//sys/print`, concatenado com o nome da porta de entrada (`/A` ou `/B`), que por sua vez é concatenado ao valor booleano `/TRUE` ou `/FALSE` . O resultado da expressão (canal Z) tem uma associação análoga, ou seja, em sequência ao *rendezvous* do canal `trueZ`, existe uma escrita no canal externo `//sys/print/Z/TRUE`; e em sequência ao rendezvous do canal `falseZ`, existe uma escrita no canal externo `//sys/print/Z/FALSE`.

O resultado esperado, após a execução do programa pela MVP, é que a *xtable* seja uma tabela verdade contendo os valores representados pelas operações nos canais externos. A figura E.1 contém essa tabela verdade.

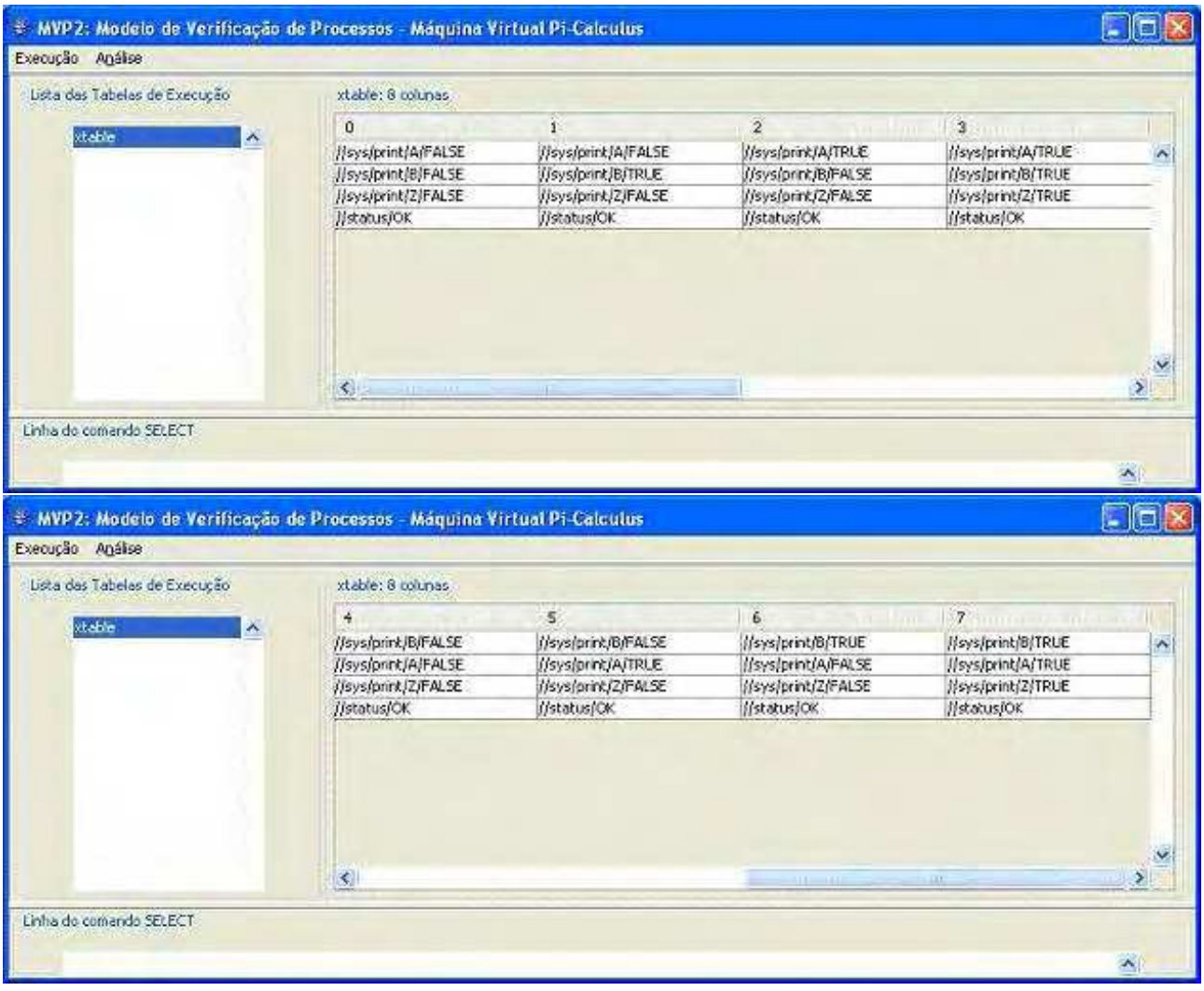

Fig. E.1: Tabela de Execução do programa de verificação da função *And* de 2 variáveis

A verificação da expressão lógica *Or* é similar à da função *And*, alterando apenas a chamada da função. O programa para a construção da tabela verdade da função *Not* é:

```
main==>
 new ( Aport,Zport) . new ( trueZ,falseZ) .
```

```
 ( ( ~//sys/print/A/FALSE .
    False<Aport> +
    ~//sys/print/A/TRUE .
    True<Aport>
  ) |
  Not<Aport, Zport> |
  ~Zport<trueZ,falseZ> .
  ( trueZ .
    ~//sys/print/Z/TRUE +
    falseZ .
    ~//sys/print/Z/FALSE
  )
)
```

Apêndice F

Desenvolvimento e Análise do Processo de Gerência de SLA

F.1 Visão Geral

O Telemanagement Forum (TMForum) é um consórcio internacional de provedores de serviços de telecomunicações e fornecedores de equipamentos e sistemas. Sua missão consiste em ajudar os provedores de serviço a automatizar seus processos, incluindo os seguintes tópicos:

- especificação dos processos de negócios;
- padronização da informação que flui entre os processos (modelo de informação);
- especificações de ambientes e arquiteturas que propiciem a integração dos Sistemas de Suporte à Operação (OSS - *Operating Support Systems*);
- apoio ao desenvolvimento de mercado e produtos para integração e automatização dos processos.

O Modelo de Referência dos processos de negócios é denominado *enhanced Telecom Operation Map* (eTOM). O eTOM caracteriza três grandes grupos de processos:

- Estratégia, Infra-estrutura e Produtos;
- Operações;
- Gerência Empresarial.

O processo de interesse no desenvolvimento deste capítulo é referente ao grupo de Operações. Nesse grupo, os processos são organizados na forma de uma matriz, como é apresentado na figura F.1.

As faixas horizontais (linhas) contêm as áreas funcionais, enquanto que as verticais (colunas) dividem os processos de operações nos seguinte grupos: *Aprovisionamento*, *Garantia da Qualidade*, *Faturamento* e *Suporte e Disponibilidade de Operações*.

Cada cruzamento de uma linha com uma coluna é um processo que é descrito em níveis de detalhes crescentes. Os níveis ora existentes vão de 1 a 3.

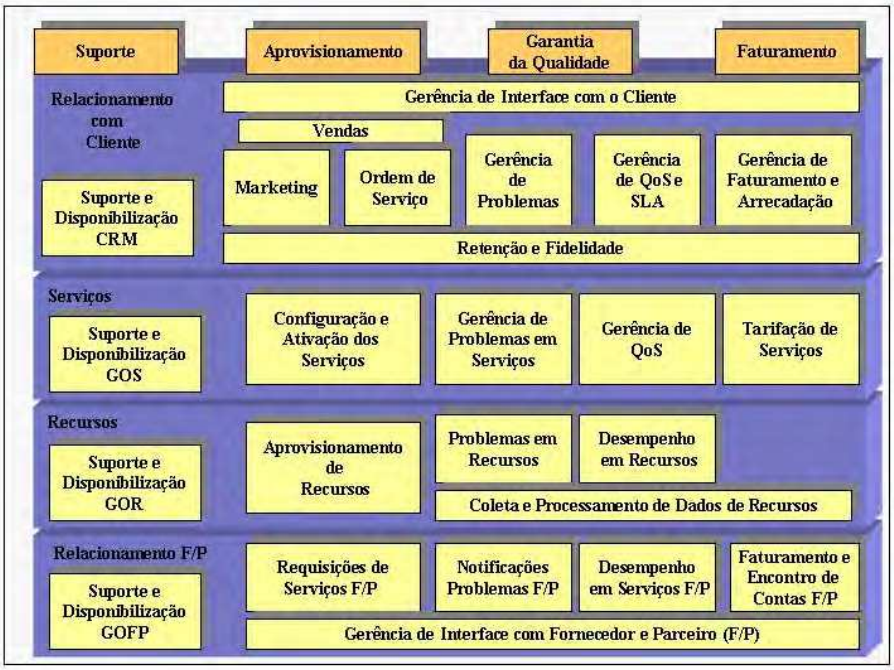

Fig. F.1: Processos do Grupo de Operações do modelo de eTOM - fonte: CPqD.

A partir da padronização dos processos nessa estrutura, os provedores de serviço constróem seus processos fim-a-fim como a integração dos processos desse modelo de referência. Em geral, um processo fim-a-fim navega em mais de uma linha e em mais de uma coluna de matriz.

O objetivo deste apêndice é construir um exemplo de processo fim-a-fim na área de telecomunicações, descrito a partir de interações com os processos de referência do eTOM. O processo escolhido é o de Gerência de SLA (*Service Level Agreement*), cujas especificações foram extraídas da referência (36).

Assim, este apêndice é composto das seguintes seções: a seção F.2 apresenta a descrição do processo; na seção F.3, o processo é especificado através dos fluxos que contêm as interações entre os processos do modelo de referência; na seção F.4 o processo é descrito através de casos de uso; a seção F.5 consiste da descrição do processo através do BPEL; a seção F.6 compreende as considerações sobre o mepeamento do processo BPEL para o programa Pi-Calculus; a seção F.7 apresenta o resultado da execução do programa pela MVP; para completar, a seção F.8 contém a análise e verificação de propriedades do processo através do módulo de Análise da MVP.

F.2 Descrição do processo de Gerência de SLA

O ciclo de vida de um contrato de SLA é composto das seguintes fases:

- desenvolvimento do produto e do serviço;
- negociação e vendas;

- implementação (aprovisionamento);
- execução (monitoração, manutenção e cobrança);
- avaliação.

O processo descrito aqui se aplica ao estágio de execução. Os fluxos que descrevem as interações entre os processos estão divididos em três classes:

- fluxo normal sem alertas e alarmes;
- fluxo com alerta e alarme;
- execução com violação de SLA.

No *fluxo normal sem alertas e alarmes*, os dados de performance são coletados dos recursos e dos serviços fornecidos por parceiros, analisados e testados com os parâmetros do SLA. Adicionalmente, o ciclo de cobrança foi colocado para mostrar como dados de recursos internos e de terceiros são agrupados para a formação de uma conta do cliente. As interações desse fluxo ocorrem nos grupos verticais Garantia da Qualidade e Faturamento, do Modelo de Referência.

No *fluxo com alerta e alarme*, ocorrem exceções instantâneas de valores de atributos dos recursos gerenciados e/ou alarmes que tenham que ser corrigidos. Todavia, após os testes dos eventos ocorridos com o que foi estabelecido no contrato, verifica-se que não há nenhuma violação de SLA. As interações ocorrem principalmente entre os processos definidos nas colunas Garantia da Qualidade e Aprovisionamento, do Modelo de Referência.

O último fluxo mostra o caso em que ocorre uma violação de SLA reportada pelo cliente. O problema é analisado e retificado, mas a violação é confirmada, produzindo um desconto na conta do cliente. Nesse fluxo, as interações ocorrem entre todos os grupos verticais.

F.3 Especificações

F.3.1 Considerações iniciais

A descrição dos fluxos do processo fim-a-fim contém vários passos, sendo que cada passo corresponde a uma interação entre os processos que são descritos no nível 2 do Modelo de Referência.

Com o objetivo de simplificar a quantidade de interações entre os processos, foram definidos macro-processos de gerência que encapsulam um ou mais processos padrão do eTOM.

Os macro-processos e os respectivos processos eTOM encapsulados são:

- **Processo de Gerência de Aprovisionamento de Serviço**: encapsula o processo Configuração e Ativação dos Serviços e todas as interações deste com os demais processos nas camadas (linhas da matriz): Serviços (exceto Tarifação de Serviços), Recursos e Relacionamento F/P (Fornecedor/Parceiro);
- **Processo de Gerência de Qualidade de Serviço**: encapsula os processos Gerência de Problemas, Gerência QoS e todas as interações destes com os demais processos nas camadas (linhas da matriz): Serviços (exceto Tarifação de Serviços), Recursos e Relacionamento F/P;

- **Processo de Gerência de Faturamento**: encapsula os processos Gerência de Faturamento e Arrecadação, Tarifação de Serviços;
- **Processo de Gerência de Relacionamento F/P**: encapsula os processos da camada de Gerência de Relacionamento F/P;
- **Processo de Gerência de Relacionamento com o Cliente**: é composto de todos os processos da linha de Gerência de Relacionamento com o Cliente (*Customer Relationship Management*), excluindo a Gerência de QoS e SLA (*Customer QoS/SLA Management*) e Gerência de Faturamento e Arrecadação (Billing and Collection Management).

O seguinte processo é considerado um macro-processo:

- **Processo de Gerência de QoS/SLA**.

A figura F.2 ilustra os macro-processos que serão usados nas especificações do processo de *Gerência de SLA*.

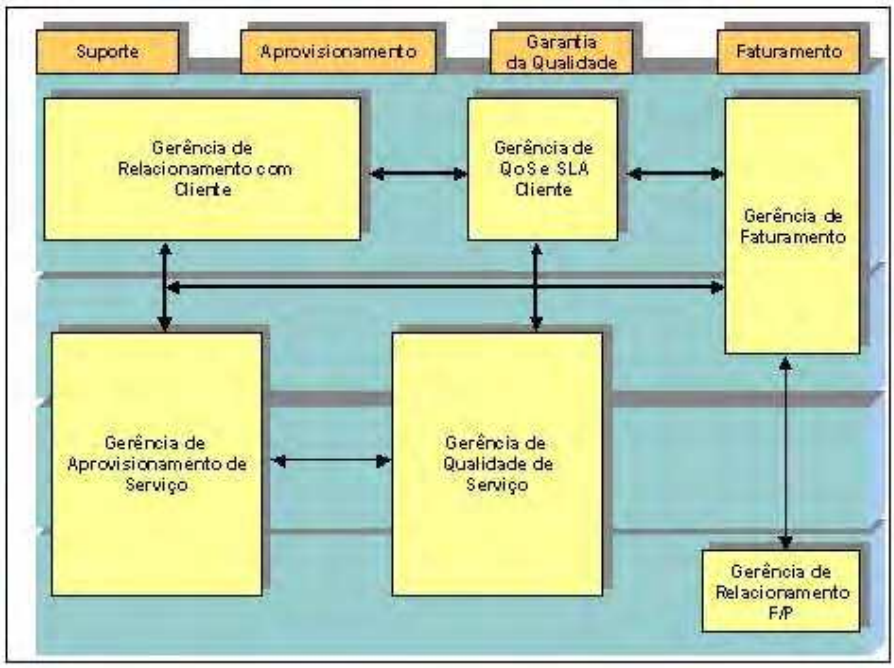

Fig. F.2: Macro-Processos usados nas especificações do processo de Gerência de SLA.

F.3.2 Cenários de interações entre os processos

Nesta seção, os cenários de interações entre os processos do documento fonte são reescritos em função das definições dos macro-processos da seção anterior. Cada passo do novo fluxo inclui a referência ao passo do documento fonte (36).

Fluxo normal de execução a partir de dados de performance e de uso

Este é o fluxo descrito como *"Normal Execution"* para o caso A:

1. o Processo de Qualidade de Serviço envia os relatórios de qualidade de serviço para o Processo de QoS/SLA (passo 9);
2. o Processo de SLA envia os relatórios de nível de serviço para o Processo de Relacionamento com o Cliente, que, por sua vez, envia para o cliente esse relatório (passo 10);
3. o Processo de Qualidade de Serviço envia os dados de uso de recursos para Processo de Faturamento (passo 6);
4. o Processo de Relacionamento F/P envia os dados de uso e cobrança dos serviços externos para Processo de Faturamento (passo 8);
5. o Processo de Faturamento envia a fatura do cliente para o Processo de Relacionamento com o Cliente, que por sua vez, envia para o cliente (passo 12).

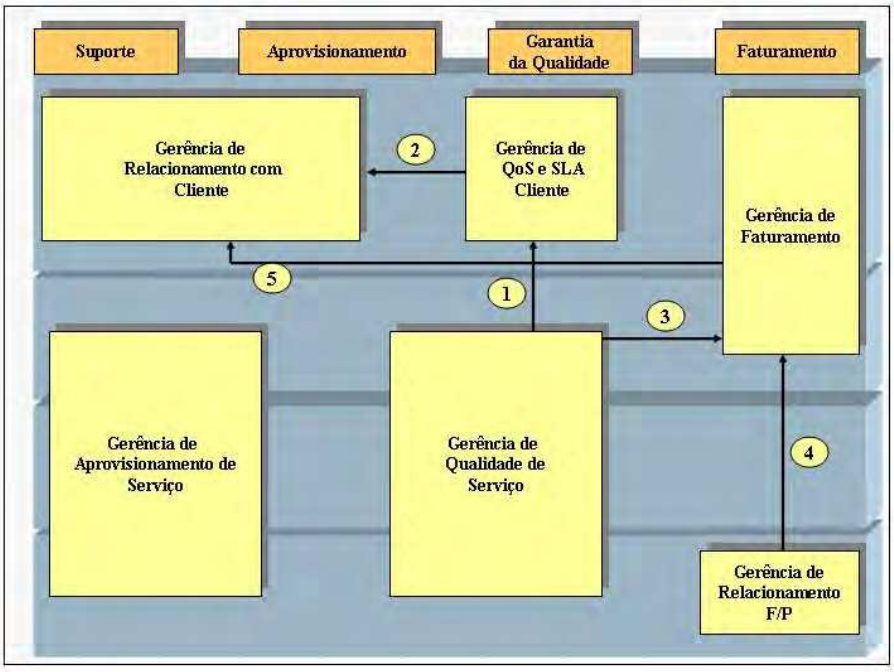

Fig. F.3: Processo de Gerência de SLA - fluxo normal.

Fluxo para o caso de alerta de ultrapassagem de limite ou alarme de serviço

Esse é o fluxo descrito como *"Normal Execution"* para os casos B e C do documento fonte:

Ocorrência do problema

1. o Processo de Qualidade de Serviço envia informações detalhadas do impacto no serviço para o Processo de QoS/SLA (passo 7). Cabe assinalar que, nos passos anteriores do fluxo da referência, ocorreram notificações de alerta de ultrapassagem de limite ou alarme que afetou o serviço;
2. o Processo de QoS/SLA obtém informação sobre o cliente através do Processo de Relacionamento com Cliente (passo 8);
3. o Processo de QoS/SLA envia a informação do impacto no contrato de SLA para o Processo de Relacionamento com o Cliente (passo 9);
4. se o Processo de Qualidade de Serviço não conseguiu fazer a restauração da condição anormal, então ele pede para o Processo de Gerência de Aprovisionamento executar as ações corretivas. O passo 5 deste fluxo é executado somente se a restauração à condição normal não ocorre (passo 10);
5. o Processo de Aprovisionamento de Serviço envia relatório com as ações executadas para o processo de Qualidade de Serviço (passo 16).

Volta à normalidade

6. o Processo de Qualidade de Serviço envia relatórios de QoS para o Processo de SLA (passo 24);
7. o Processo de QoS/SLA envia informação de serviço restaurado para o Processo de Relacionamento com o Cliente, dependendo do contrato do cliente e da extensão do problema (passo 25);
8. o Processo de QoS/SLA envia relatórios de nível de serviço para Processo de Relacionamento com Clientes (passo 26).

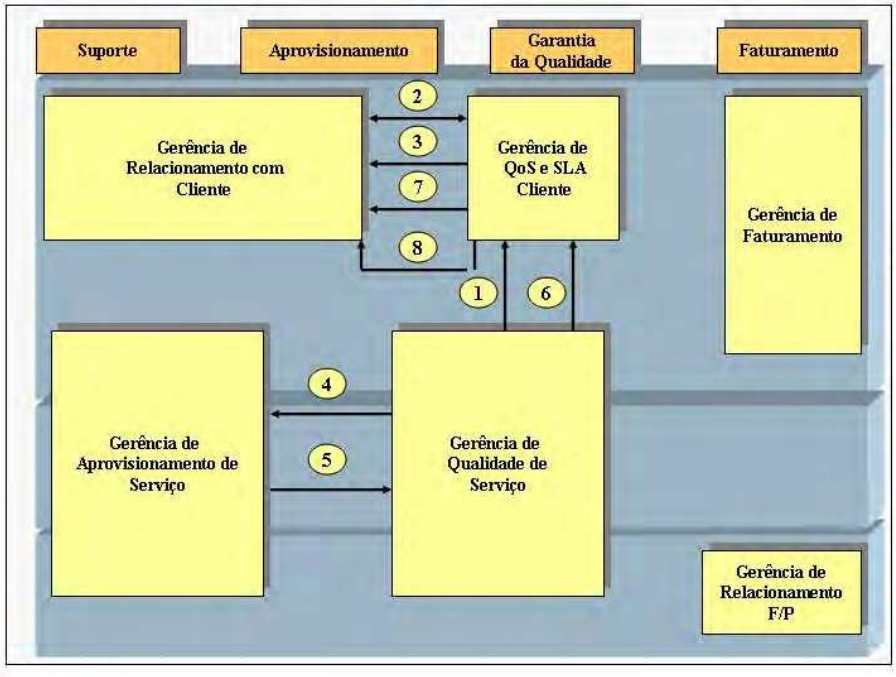

Fig. F.4: Processo de Gerência de SLA - fluxo com alerta e sem violação.

Execução com violação de SLA

Esse é o fluxo descrito como *"Execution with SLA Violation"* para os casos B e C.

Cenário

Durante a vigência dos serviços de um contrato SLA, podem ocorrer situações em que os parâmetros especificados no contrato ultrapassam os limites. Dois casos devem ser examinados, um em que o provedor de serviço detecta a falha e outro quando o cliente detecta e reporta primeiro. O fluxo descrito a seguir aplica-se ao segundo caso, que é o único descrito no documento fonte.

Fluxo

1. o Processo de Relacionamento com o Cliente envia o problema reportado para o Processo de QoS/SLA (passo 2);
2. o Processo de SLA determina prioridades e ações corretivas dependendo do contrato e envia para o Processo de Relacionamento com o Cliente (passo3);
3. o Processo de Relacionamento com o Cliente envia o problema reportado para o Processo de Qualidade de Serviço (passo 5), que, por sua vez, investiga o problema com outros processos;
4. conforme o resultado obtido do problema investigado, executa uma da alternativas A, B ou C.

Alternativa A

5. não existe problema: o Processo de Qualidade de Serviço envia o relatório de performance do serviço para o Processo de Relacionamento com o Cliente, que por sua vez reporta ao Cliente (passo 9A);
6. o Processo de Relacionamento com o Cliente envia relatório de performance de serviço para o Processo de QoS/SLA (passo 10A).

O fluxo para essa alternativa termina.

Alternativa B

7. execução de restauração automática é necessária: o próprio Processo de Qualidade de Serviço providencia a restauração automática, não havendo interações entre os macro-processos.

Esse fluxo continua no passo 10.

Alternativa C

8. execução de ações de mudanças na configuração é necessária: o Processo de Qualidade de Serviço pede para o processo de Gerência de Aprovisionamento executar as ações corretivas (passo 9c);
9. o Processo de Aprovisionamento de Serviço envia relatório com as ações executadas para o processo de Qualidade de Serviço (passo 15c).

Fluxo - continuação

10. o Processo de Qualidade de Serviço envia o relatório com o problema que foi resolvido para Processo de Relacionamento com o Cliente, que, por sua vez, envia a informação para o cliente reconhecer que o problema está resolvido (passo 21);
11. o Processo de Gerência de Qualidade Serviço envia confirmação de serviço restaurado para Processo de SLA (passo 23);
12. o Processo de Gerência de QoS/SLA envia relatório de nível de serviço para o Processo de Relacionamento com o Cliente, que envia para o Cliente (passo 25);
13. o Processo de QoS/SLA envia o valor de desconto da fatura para o Processo de Faturamento (passo 24).

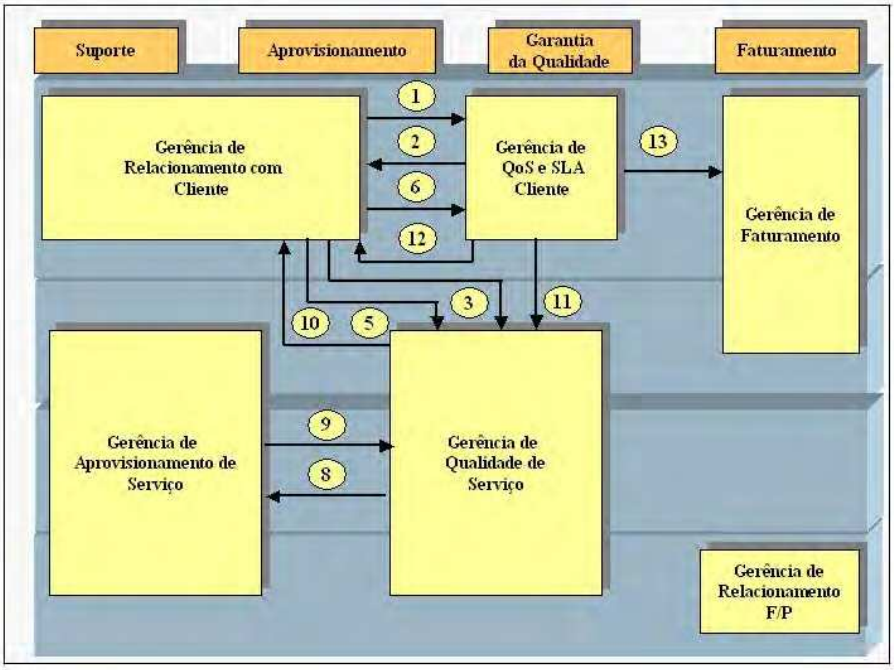

Fig. F.5: Processo de Gerência de SLA - fluxo com violação de SLA.
Nota: Os passos 3 e 7 não são interações entre processos.

F.4 Casos de uso para o processo fim-a-fim

A partir das especificações adequadas ao ambiente dos macro-processos feitas na seção anterior, descrevem-se os casos de uso do processo fim-a-fim de Gerência de SLA.

Assume-se que cada macro-processo é implementado por um sistema com o mesmo nome que provê interfaces *web services*. O processo de Gerência de SLA é implementado por um BPMS (*Business Process Management System*) que interpreta processos escritos em BPEL. Esse processo fim-a-fim, portanto, é o elemento integrador dos processos do modelo de referência do TMForum.

Algumas considerações adicionais para o processo fim-a-fim são necessárias:

- O contrato é associado a um serviço que pode ser composto de recursos internos e outros serviços externos. Os primeiros são tratados pela camada Recursos do Modelo de Referência. Os serviços externos são implementados pela camada Relacionamento F/P.
- Para melhor ilustrar a aplicação da análise do processo BPEL, foram criados alguns fluxos alternativos que não estão descritos no documento de referência.

As próximas sub-seções descrevem os fluxos do processo de forma a atender às especificações definidas nos Cenários de Interações entre Processos.

F.4.1 Fluxo normal

Fluxo A

Os fluxos A1 e A2 são cíclicos e executados a cada intervalo de tempo definido pelo contrato do serviço.

Fluxo A1

1. invoca o Sistema de Qualidade de Serviço para obtenção dos relatórios de QoS (*Quality of Service*);
2. invoca o Sistema de Gerência de SLA para obtenção dos relatórios de nível de serviço;
3. envia os relatórios de nível de serviço para o Sistema de Relacionamento com o Cliente.

Fluxo A2

4. invoca o Sistema de Qualidade de Serviço para obtenção das informações de uso de recursos;
5. invoca o Sistema de Relacionamento F/P para obtenção das informações de uso e cobrança dos serviços externos;
6. invoca o Sistema de Faturamento com os dados obtidos nos passos anteriores para obtenção da fatura do cliente;
7. envia a fatura do cliente para o Sistema de Relacionamento com o Cliente.

Fluxo Alternativo

2.a. relatórios de nível de serviço indicam anormalidade (desempenho de serviço não satisfatório ou contrato violado):

1. cria uma descrição detalhada do problema;
2. desvia para o passo 2 do fluxo C;

A similaridade entre esse caso e o do fluxo C é que existem problemas no serviço ou no contrato que ainda não foram sinalizados pelo Sistema de Qualidade.

F.4.2 Anormalidade de um recurso ou de um serviço externo

Fluxo B

1. recebe do Sistema de Qualidade de Serviços uma notificação de alerta de ultrapassagem de limite (AUL) ou um alarme (ALM) através de uma descrição detalhada dos eventos AUL e ALM e do impacto no serviço (paralelamente, o Sistema de Qualidade de Serviços tenta restaurar a condição normal);
2. desse ponto, o fluxo é desmembrado em outros dois: fluxo B1 e fluxo B2.

Fluxo B1

3. invoca o Sistema de Qualidade de Serviços para obter o resultado da restauração automática;
4. se a restauração não foi bem sucedida, então invoca o Sistema de Aprovisionamento de Serviço para executar a mudança de configuração para restaurar o serviço e obter o resultado de execução da restauração do serviço.

Fluxo B2

5. invoca o Sistema de Gerência de Relacionamento com o Cliente para obtenção dos dados do cliente do serviço afetado;
6. invoca o Sistema de Gerência de SLA com os dados obtidos anteriormente tendo como saída as informações de impacto no nível de serviço através dos relatórios de nível de serviço;
7. se o impacto do nível de serviço para o cliente for no nível tal que deve ser notificado, então envia esses dados para o Sistema de Gerência de Relacionamento.

Fluxo B - continuação

8. recebe do Sistema de Qualidade de Serviço indicação de restauração do serviço acompanhado dos relatórios de QoS;
9. invoca o Sistema de Gerência de SLA com os dados anteriores para obtenção dos relatórios de nível de serviço;
10. se no passo 7 o cliente foi notificado, então envia para o Sistema de Relacionamento com o Cliente a indicação de restauração do serviço;
11. envia para o Sistema de Relacionamento com o Cliente os relatórios de nível de serviço.

Cabe ressaltar que os passos 10 e 11 devem ser executados paralelamente.

Fluxo Alternativo

8.a. não ocorre o evento (volta à normalidade) após um determinado tempo:
   1. cria uma descrição detalhada do problema;
   2. desvia para o passo 2 do fluxo C.

F.4.3 Reclamação do cliente sobre violação de SLA

Fluxo C

1. recebe do Sistema de Relacionamento com o Cliente a descrição detalhada do problema;
2. invoca o Sistema de Gerência de QoS/SLA com a descrição detalhada do problema obtendo os passos a serem executados para que o nível de SLA não seja violado;
3. invoca o Sistema de Qualidade de Serviço com os dados acima e obtém as informações para investigação do problema;
4. conforme as informações do problema, o fluxo é desmembrado em um dos três abaixo.

Fluxo C1

5. não existe problema: executa os passos do fluxo A1;

Fluxo C2

6. execução de restauração automática do serviço é necessária: não há necessidade de nenhuma atividade aqui, pois o Sistema de Qualidade de Serviço faz automaticamente a restauração.

Fluxo C3

7. execução de ações de mudanças na configuração é necessária: envia pedido para o Sistema de Aprovisionamento de Serviço executar as mudanças de configuração para restaurar o serviço e obter os resultados da execução das mudanças de configuração;
8. envia os resultados do passo anterior para o Sistema de Qualidade de Serviços.

Fluxo C - continuação (para os fluxos C2 e C3)

9. invoca o Sistema de Qualidade de Serviços para obter os relatórios de QoS com a indicação de problema resolvido;
10. esse fluxo se desmembra em dois outros: fluxos C4 e C5.

Fluxo C4

11. envia os dados obtidos anteriormente para o Sistema de Relacionamento com o Cliente.

Fluxo C5

12. invoca o Sistema de Gerência de QoS/SLA com os dados do problema resolvido para obtenção do desconto na fatura, conforme contrato;
13. envia para o Sistema de Faturamento os dados anteriores;
14. envia para o Sistema de Relacionamento com o Cliente o desconto na fatura.

Fluxo Alternativo

9.a. Resultado da execução das mudanças indica que o problema não foi resolvido:

1. invoca o Sistema de Gerência de Falhas em Processos com indicação de falha no Sistema de Garantia de SLA;
2. invoca o Sistema de Gerência de SLA para gerar o desconto na fatura, dado que ocorreu a falha no próprio sistema;
3. envia para o Sistema de Relacionamento com o Cliente o desconto da fatura.

F.5 Construção em BPEL

A partir das considerações adicionais e dos casos de uso definidos na seção anterior, foi desenvolvido o processo BPEL que corresponde à implementação do processo fim-a-fim de Gerência de SLA.

F.5.1 Serviços dos processos

A tabela F.1 apresenta uma relação dos nomes simbólicos dos sistemas que são usados pelo processo de Gerência de SLA.

Nome do Sistema	*Descrição*
Billing	Sistema de Faturamento
CRM	Gerência de Relacionamento com o Cliente
FPM	Gerência de Relacionamento F/P (Fornecedores e Parceiros)
SLM	Gerência de QoS/SLA
PM	Gerência de Aprovisionamento
QM	Gerência de Qualidade de serviço
FM	Gerência de Falhas de Processos

Tab. F.1: Nomes dos sistemas que correspondem aos macro-processos.

A tabela F.2 contém uma lista dos *web services* exportados pelos sistemas que implementam os macro-processos do modelo de referência do TMForum. Cada linha define um *web service* e contém

Sistema	*Porta*	*Operação*	*Canal Pi-calculus*	*Casos de uso*
Billing	fatura do cliente	ler	Billing/bill/getInfo	A6
Billing	desconto na fatura	enviar	Billing/rebate/send	C13
CRM	fatura do cliente	enviar	CRM/bill/send	A7
CRM	informações de cliente	ler	CRM/customer/getInfo	B5
CRM	problema detalhado	ler	CRM/problem/receive	C1
CRM	relatórios qos	enviar	CRM/qos/send	C11
CRM	desconto na fatura	enviar	CRM/rebate/send	C14
CRM	restauração do serviço	enviar	CRM/serviceRestored/send	B10
CRM	relatórios de serviço	enviar	CRM/sla/send	A3 B7 B11
FPM	uso e cobrança dos serviços	ler	FPM/usageRating/getInfo	A5
SLM	desconto na fatura	ler	SLM/rebate/getInfo	C12
SLM	script SLA	ler	SLM/script/getInfo	C2
SLM	relatórios de SLA	ler	SLM/sla/getInfo	A2 B6 B9
PM	restaurar serviço	executar	PM/serviceRestore/exec	B4 C7
QM	resultado da restauração automática	ler	QM/automaticRestore/getInfo	B3
QM	relatórios de QoS	ler	QM/qos/getInfo	A1 C9
QM	notificação de restauração do serviço	receber	QM/RestoredEventAndQos/receive	B8
QM	investigação do problema	executar	QM/script/exec	C3
QM	resultados de restauração	enviar	QM/serviceRestoreResult/send	C8
QM	ultrapassagem de limite ou alarme	receber	QM/thresholdOrAlarm/receive	B1
QM	informações de uso	ler	QM/usage/getInfo	A4

Tab. F.2: Serviços exportados dos sistemas que correspondem aos macro-processos.

o sistema que implementa o serviço, a porta WDSL, a operação WSDL, o canal Pi-Calculus que é usado na compilação e os casos de uso que referenciam o serviço.

Foi definido mais um serviço que não está associado diretamente aos processos aqui definidos. Esse serviço está na tabela F.3

Sistema	*Porta*	*Operação*	*Canal*	*Caso de uso*
FM	notificação de alarme	enviar	FM/slaAssuranceAlarm/send	C9.a1

Tab. F.3: Serviço exportado do sistema auxiliar.

F.5.2 Descrição simplificada em BPEL

Ao ser iniciado, o processo cria dois fluxos paralelos (*flow*). O primeiro funciona como um servidor para tratar os eventos que podem disparar uma ação no processo; eventos esses que são: a chegada de notificações do Sistema de Qualidade dos Recursos (fluxo B), uma reclamação do cliente através do Sistema de Relacionamento com Clientes (fluxo C) e as temporizações para apresentação de relatórios de nível de qualidade do serviço (fluxo A1) e uso do serviço (fluxo A2).

O outro fluxo trabalha como um servidor interno para executar o fluxo C, pois este também pode ser desviado de um outro fluxo interno, como por exemplo, o fluxo alternativo A1.2.a. A ligação `servidor-fluxoC` é usada para a sincronização.

No processo BPEL, foram incluídas atividades *empty*, que, ao serem traduzidas para Pi-Calculus como atividades relevantes, auxiliam a fase de análise. Essas atividades foram colocadas para marcar as entradas nos fluxos A1, A2, B e C (atividades cujos nomes representam o fluxo) e também para indicar que houve um fim normal do processo (atividade com o nome *exit*).

A estrutura do processo BPEL para a implementação do processo de Gerência de SLA é apresentada a seguir:

```
<process>
  <flow>
    <pick>
      <onMessage ... >                    <!-- passo B1 -->
         <empty name="B" />
         fluxo B2-B11                     <!-- passos B2-B11 -->
      </onMessage>
      <onMessage>                      <!-- passo C1 -->
         <source linkName="servidor-fluxoC"
                 transitionCondition="true" />
         <empty name="C" />
      </onMessage>
      <onAlarm for="..." />
         <empty name="A1" />
         fluxo A1                 <!-- fluxo A1 -->
      </onAlarm>
      <onAlarm name="A2" for="..." />
         <source linkName="servidor-fluxoC"
                 transitionCondition="false"
         <empty name="A2" />
         fluxo A2                 <!-- fluxo A2 -->
      </onAlarm>
    </pick>
    <sequence>
      <target linkname="servidor-fluxoC" />
      fluxo C2-C14                    <!-- passos C2-C14 -->
    </sequence>
  </flow>

  <!-- essa atividade foi adicionada exclusivamente para a análise:
       ao ser executada indica que o processo
       chega ao fim normal -->
 <empty name="exit />
</process>
```

Fluxo A1

```
<sequence>
 <invoke ... />    <!-- Passo A1 -->
 <invoke ... />    <!-- Passo A2 -->
 <switch ...>
    <case "relatórios de nível de serviço indicam anormalidade" />
       <assign name="A1" />     <!-- Passo A2.a -->
       <source linkName="servidor-fluxoC"
               transitionCondition="true" />
    </case>
    <otherwise>
       <invoke ... />    <!-- Passo A3 -->
       <source linkName="servidor-fluxoC"
               transitionCondition="false" />
    </otherwise>
 </switch>
</sequence>
```

Fluxo A2

```
<sequence>
 <invoke ... />    <!-- Passo A4 -->
 <invoke ... />    <!-- Passo A5 -->
 <invoke ... />    <!-- Passo A6 -->
 <invoke ... />    <!-- Passo A7 -->
</sequence>
```

Fluxo B (passos B2 a B11)

```
<sequence>
 <flow>             <!-- passo B2 -->
   <sequence>
     <invoke ...>    <!-- passo B3 -->
     <switch ...>    <!-- passo B4 -->
       <case "restauração automática não foi bem-sucedida" />
       <invoke ... />    <!-- passo 4 (cont) -->
       </case>
     </switch>
   </sequence>
   <sequence>
     <invoke ...>    <!-- passo B5 -->
     <invoke ...>    <!-- passo B6 -->
     <switch ...>    <!-- passo B7 -->
        <case "impacto no nível de serviço deve ser notificado" />
        <invoke ... />    <!-- passo B7 (continuação) -->
        </case>
     </switch>
   </sequence>
 </flow>
 <pick>
   <onMessage ... >    <!-- passo B8 -
```

```
    <source linkName="servidor-fluxoC"
            transitionCondition="false" />
    <invoke ... />    <!-- passo B9 -->
    <switch ...>    <!-- passo B10 -->
      <case "impacto no nível de serviço deve ser notificado" />
        <invoke ... />    <!-- passo B10 (continuação) -->
      </case>
    </switch>
    <invoke ... />    <!-- passo B11 -->
  </onMessage>
  <onAlarm for= "..." >             <!-- fluxo alternativo B8.a -->
    <assign name="B8.a"... />      <!-- passo B8.a1 />
      <source linkName="servidor-fluxoC" />  <!-- passo B8.a2 />
    </assign>
  </onAlarm>
 </pick>
</sequence>
```

Fluxo C (passos C2 a C14)

```
<sequence>
 <invoke ... />    <!-- passo C2 -->
 <invoke ... />    <!-- passo C3 -->
 <invoke ... />    <!-- passo C4 -->
 <flow>
   <switch>
     <case "não existe problema" >   <!-- passo C5 -->
       <invoke ... />    <!-- Passo C5/A1 -->
       <invoke ... />    <!-- Passo C5/A2 -->
       <invoke ... />    <!-- Passo C5/A3 -->
       <source   linkName="fluxoC-continua"
                 transitionCondition = "false" />
       <source   linkName="fluxoC-alternativo"
                 transitionCondition = "false" />
     </case>
     <case "execução de restauração de serviço ..." > <!-- passo C6 -->
       <source linkName = "fluxoC-continua"
               transitionCondition = "true" />
       <source linkName = "fluxoC-alternativo"
               transitionCondition = "false" />
     </case>
     <otherwise>
 <invoke ... />             <!-- passo C7 -->
 <invoke ... />             <!-- passo C8 -->
 <source linkName = "fluxoC-continua"
               transitionCondition = "true" />
     </otherwise>
   </switch>
   <sequence>               <-- junção dos fluxos C2 e C3 -->
     <target linkName = "fluxoC-Continua" />
     <invoke ... />         <!-- passo C9  -->
```

```
      <switch ...>
         <case "resultado indica que o problema não foi resolvido" />
                              <!-- fluxo alternativo 9.a -->
           <source linkName="fluxoC-alternativo"
                   transitionCondition="true" />
         </case>
        <otherwise>
          <flow>              <!-- passo C10 -->
            <invoke ... >     <!-- passo C11 -->
            <sequence>
               <invoke ... >     <!-- passo C12 -->
               <invoke ... >     <!-- passo C13 -->
               <invoke ... >     <!-- passo C14 -->
            </sequence>
          </flow>
          <source linkName="fluxoC-alternativo"
                  transitionCondition="false" />
        </otherwise>
      </switch>
    </sequence>
    <sequence>  <-- fluxo alternativo 9.a  -->
      <target linkName = "fluxoC-alternativo" />
      <invoke ... />       <!-passo C9.a1 -->
      <invoke ... />       <!-passo C9.a2 -->
      <invoke ... />       <!-passo C9.a3 -->
    </sequence>
  </flow>
 </sequence>
```

F.6 Mapeamento do processo BPEL para o programa Pi-Calculus

O primeiro passo na utilização do *framework* proposto neste trabalho, como descrito no capítulo 1, é o mapeamento do processo BPEL para o programa Pi-Calculus, seguindo as regras estabelecidas no capítulo 4. O programa resultante é escrito na linguagem da MVP. A expressão correspondente a esse programa está listada no apêndice G.

Como o mapeamento é feito manualmente e o processo BPEL que foi traduzido tem um tamanho considerável, alguns cuidados foram tomados para evitar erros, e conseqüentemente existem alguns processos Pi-Calculus que na tradução normal não existiriam. Esses processos comportam-se como macros que, sendo expandidas ao serem invocadas por um processo, não afetam o resultado.

F.7 Execução do programa Pi-Calculus

O segundo passo no uso do *framework* é a execução do programa Pi-Calculus pela MVP para produzir a Árvore de Execução. A Árvore de Execução passa por um filtro para extrair somente os canais externos e gerar a Tabela de Execução denominada *xtable*, que é a entrada para o terceiro passo - a análise e verificação do programa.

A execução do programa do exemplo gerou uma Tabela de Execução com 323 colunas como mostra a figura F.6.

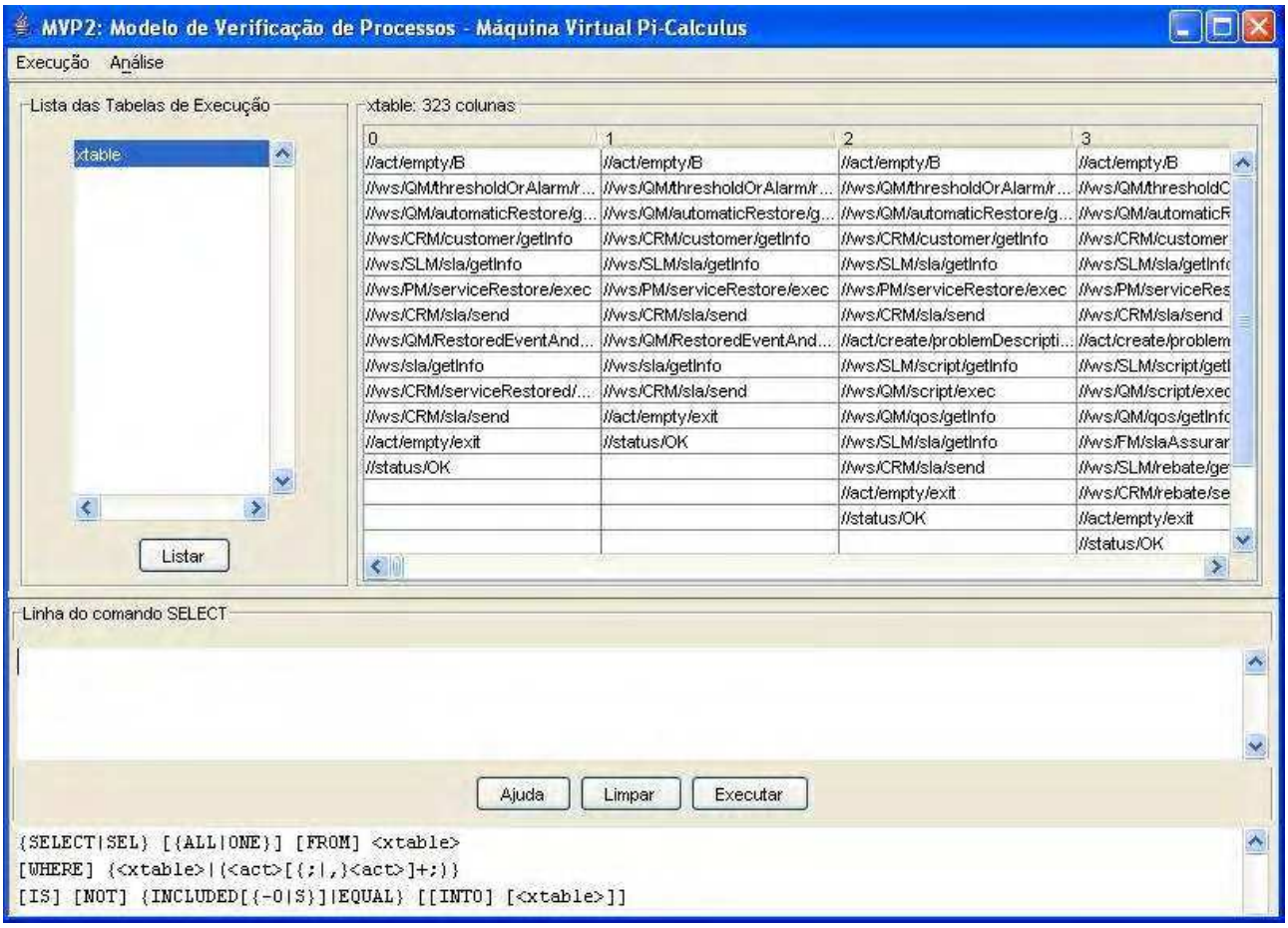

Fig. F.6: Tabela de Execução do programa Pi-Calculus resultante do processo de Gerência de SLA.

F.8 Análise e verificação do processo

O terceiro e último passo no uso do framework é a análise e verificação de propriedades do processo BPEL através do programa Pi-Calculus. Inicialmente, constata-se, através do comando de verificação de *deadlock*, que este programa é livre de *deadlock*. A seguir, serão desenvolvidas duas análises de propriedades, mais uma análise para otimização e, por último, serão introduzidas algumas mudanças no processo, para exemplificar o caso de existência de *deadlock*.

Análise 1

Esta análise consiste em determinar se, ao executar o fluxo A1 alternativo, o relatório de nível de serviço é devidamente enviado para o Sistema de Relacionamento com o Cliente.

O comando abaixo constrói uma tabela que contém os caminhos de execução do fluxo A1:

```
SELECT xtable (//ws/empty/A1) INCLUDED INTO fluxoA1
```

O número de colunas selecionadas é 12. A partir da tabela que contém o fluxo A1, seleciona-se em uma outra tabela o fluxo alternativo.

```
SELECT fluxoA1
(//act/create/problemDescription/slaReport) INCLUDED fluxoA1Alt
```

A tabela `fluxoA1Alt` contém 11 colunas. O comando abaixo seleciona todas as colunas que não contêm a execução do *web service* `//ws/CRM/sla/send`.

```
SELECT fluxoA1Alt (//ws/CRM/sla/send) NOT INCLUDED semCRM
```

Uma amostra do resultado é apresentada na figura F.7. O número de colunas selecionadas é 10, o que leva a concluir que existe um caso dentro do fluxo alternativo em que ocorre a chamada ao *web service* que envia o relatório de nível de serviço para o cliente. Para saber em qual caso isso ocorre, executa-se o comando negado, como mostrado abaixo:

```
SELECT fluxoA1Alt (//ws/CRM/sla/send) INCLUDED comCRM
```

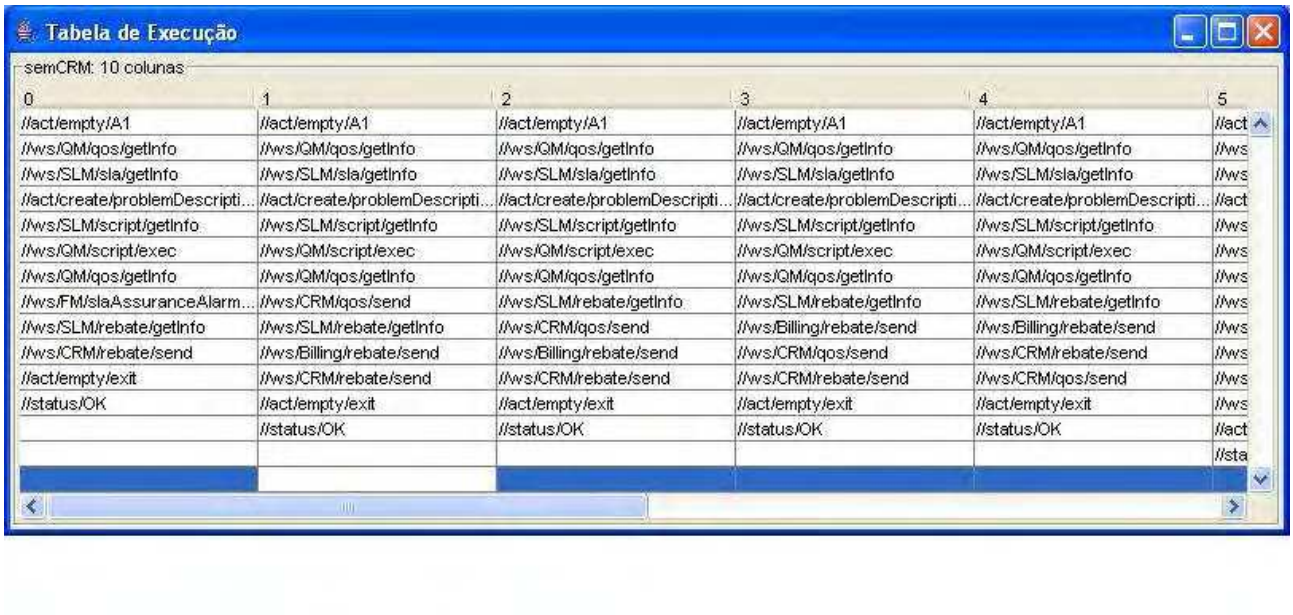

Fig. F.7: Tabela de Execução *semCRM*.

Esse comando, como era de se esperar, produz uma tabela de uma coluna. A figura F.8 mostra o único caminho de execução dessa tabela. A conclusão da análise é que existe um caminho em que o relatório é enviado para o Sistema de Relacionamento com o Cliente.

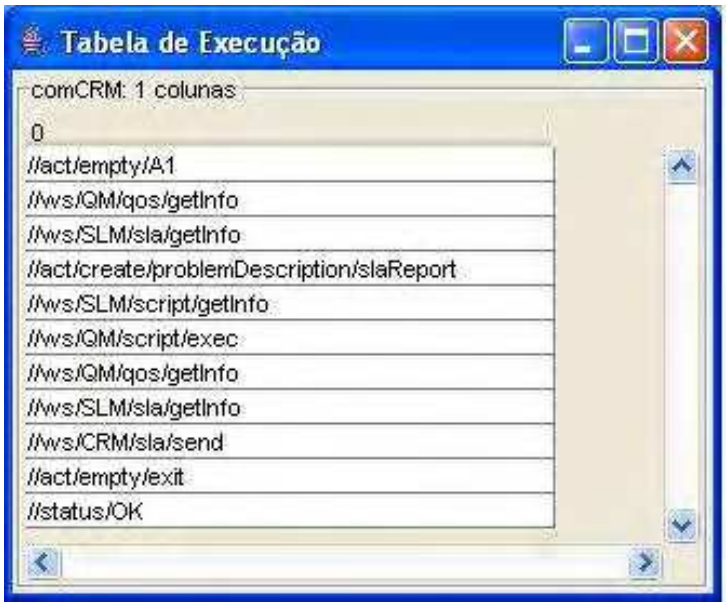

Fig. F.8: Tabela de Execução *comCRM*.

Análise 2

Tomando como referência o fluxo B, deseja-se verificar as duas propriedades que são definidas adiante.

Pré-condição: ocorre uma notificação de ultrapassagem de limite ou alarme e não acontece a indicação de restauração de serviço.

Evento 1: o cliente recebe um desconto;

Evento 2: o serviço normaliza e então os três passos do fluxo A1 são executados sem ocorrer nenhuma outra interação entre esses passos (o cliente recebe o relatório de nível de serviço).

Dado que ocorre a pré-condição estabelecida acima, deseja-se verificar as seguintes propriedades:

Propriedade 1: ou Evento 1 ou Evento 2 (mutuamente excludentes).

Propriedade 2: a Propriedade 1 cobre o universo dos casos, ou seja, não existe nenhum caso em que não ocorra um dos dois eventos.

O comando abaixo seleciona os caminhos de execução onde ocorre a primeira notificação da pré-condição, ou seja, ocorre uma notificação de ultrapassagem de limite ou um alarme.

```
SELECT xtable (//ws/QM/thresholdOrAlarm/receive) INCLUDED aux
```

O resultado obtido contém 299 colunas. O comando seguinte completa a pré-condição selecionando os caminhos onde não ocorre a restauração do serviço.

```
SELECT aux (//ws/QM/RestoredEventAndQos/receive)
NOT INCLUDED pré-condição
```

A tabela resultante do comando acima contém 253 colunas. O comando seguinte seleciona dessa tabela o Evento 1, ou seja, os caminhos de execução que contêm o *web service* em que o cliente recebe o desconto.

```
SELECT pré-condição (//ws/CRM/rebate/send) INCLUDED tabEvento1
```

O número de colunas da tabela `tabEvento1` é 230. De maneira semelhante, obtêm-se os caminhos que satisfazem a condição do Evento 2, ou seja, dado que ocorre a pré-condição, seleciona os caminhos de execução que contêm os três *web services* especificados em seqüência:

```
SELECT pré-condição (//ws/QM/qos/getInfo,//ws/SLM/sla/getInfo,//ws/CRM/sla/send)
INCLUDED-S tabEvento2
```

O número de colunas resultante do comando na tabela `tabEvento2` é 23. Para verificar que Evento 1 e Evento 2 ocorrem mutuamente excludentes, efetuam-se os comandos de seleção de uma tabela com os parâmetros da outra, ou seja, da tabela que contém os caminhos de `tabEvento1` , usa-se o parâmetro que determina o `tabEvento2`, e vice-versa.

```
SELECT tabEvento1 (//ws/QM/qos/getInfo,//ws/SLM/sla/getInfo,//ws/CRM/sla/send)
INCLUDED-S ev1-ev2

SELECT tabEvento2 (//ws/CRM/rebate/send) INCLUDED ev2-ev1
```

O resultado esperado é que as duas tabelas resultantes sejam vazias, o que mostra o atendimento da Propriedade 1. Como a soma das colunas de `evento1` e `evento2` é igual ao número de colunas de `pré-condição`, a Propriedade 2 é satisfeita.

Análise para Otimização

Para exemplificar uma análise para otimização do processo, supõe-se que, num determinado momento da análise do processo, suspeita-se que, sempre que o *web service* `//ws/CRM/qos/send` é executado, //ws/CRM/rebate/send também o seja. Isso significa dizer que um relatório de qualidade de serviço é enviado para o CRM sempre que ocorre um desconto por falha no acordo do SLA. Essa propriedade pode ser verificada através da análise do código fonte do programa BPEL. Porém, isso pode ser feito com mais certeza através da análise do programa Pi-Calculus, que é desenvolvida a seguir.

Primeiro, obtém-se as colunas na quais ocorrem a execução do serviço `//ws/CRM/qos/send`:

```
SELECT xtable (//ws/CRM/qos/send) INCLUDED CRMQosSend
```

Em seguida, extrai-se da tabela original as colunas em que o *web service* `//ws/CRM/rebate/send` é executado:

```
SELECT xtable (//ws/CRM/rebate/send) INCLUDED CRMRebateSend
```

O primeiro comando seleciona 200 colunas e o segundo 250. O comando abaixo seleciona, dos caminhos que têm o desconto, aqueles que enviam o relatório de qualidade de serviço:

```
SELECT CRMRebateSend (//ws/CRM/qos/send) INCLUDED qosFromRebate
```

Como o resultado deste comando é uma tabela de 200 colunas também, verifica-se a propriedade que havia sido suspeitada. Como conclusão desta análise, poder-se-ia propor a construção de um novo *web service* no Sistema CRM que seria equivalente à chamada dos dois *web services*.

Exemplo de Detecção de *Deadlock*

A implementação do programa BPEL não segue rigorosamente as especificações dos casos de uso. Se essas especificações fossem plenamente atendidas, haveria situações em que ocorreriam *deadlocks*. O passo C5 do caso de uso informa que é para executar os passos do fluxo A1. Porém, se isso fosse feito, a execução desses passos chamariam de volta o fluxo C provocando o *deadlock*. Para exemplificar como o *deadlock* ocorre, são introduzidas modificações no processo BPEL e no programa Pi-Calculus, seguindo o descrito no caso de uso.

Para executar o fluxo A1 tanto a partir do temporizador como de um ponto do fluxo C, o fluxo A1 é definido como um servidor interno da mesma forma que o fluxo C. O corpo principal do processo passa a incluir a seqüência:

```
<sequence>
  <target linkname="servidor-fluxoA1" />
  fluxo-A1
</sequence>
```

O trecho do programa que inicia o fluxo A1 muda para:

```
<onAlarme for="..." >
  <source linkName = "servidor-A1"
          transitionCondition="true" />
</onAlarm>
```

Nos trechos das chamadas dos fluxos A2 e B, são introduzidas as cláusulas *source* abaixo:

```
<source linkName = "servidor-A1"
        transitionCondition="false" />
```

O passo C5 do fluxo C, ao invés de invocar os três *web services*, inclui a seguinte cláusula *source*:

```
<source linkName = "servidor-A1"
        transitionCondition="true" />
```

E, finalmente, nos passos C6 e C8 acrescenta-se:

```
<source linkName = "servidor-A1"
        transitionCondition="false" />
```

As modificações descritas acima foram implementadas e traduzidas para o programa Pi-Calculus que foi executado na MVP. Como era de se esperar, o resultado contém a situação de *deadlock*, como mostra a figura F.9.

As linhas da tabela indicam linhas de execução que têm os mesmos canais em *deadlocks* com os respectivos nós na Árvore de Execução. A título de ilustração, é apresentada na figura F.10 a Árvore de Execução com um dos nós que está em *deadlock* em realce.

Observando a expressão do nó 47, verifica-se que neste caso o fluxo C foi ativado a partir do fluxo A1, que, por sua vez, está tentando ativar novamente o fluxo A1. A expressão mostra que existem três processos em paralelo, e que os dois últimos foram ativados para esperar a decisão de continuação

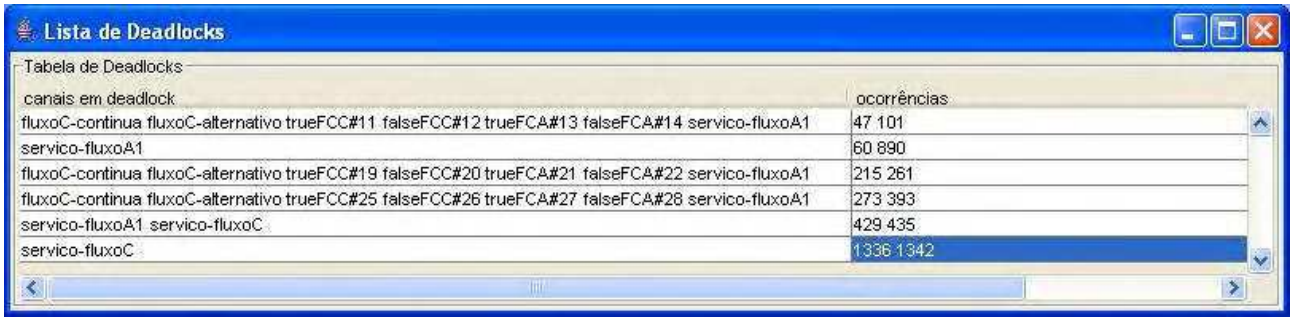

Fig. F.9: Resultado do comando para verificar *deadlock*.

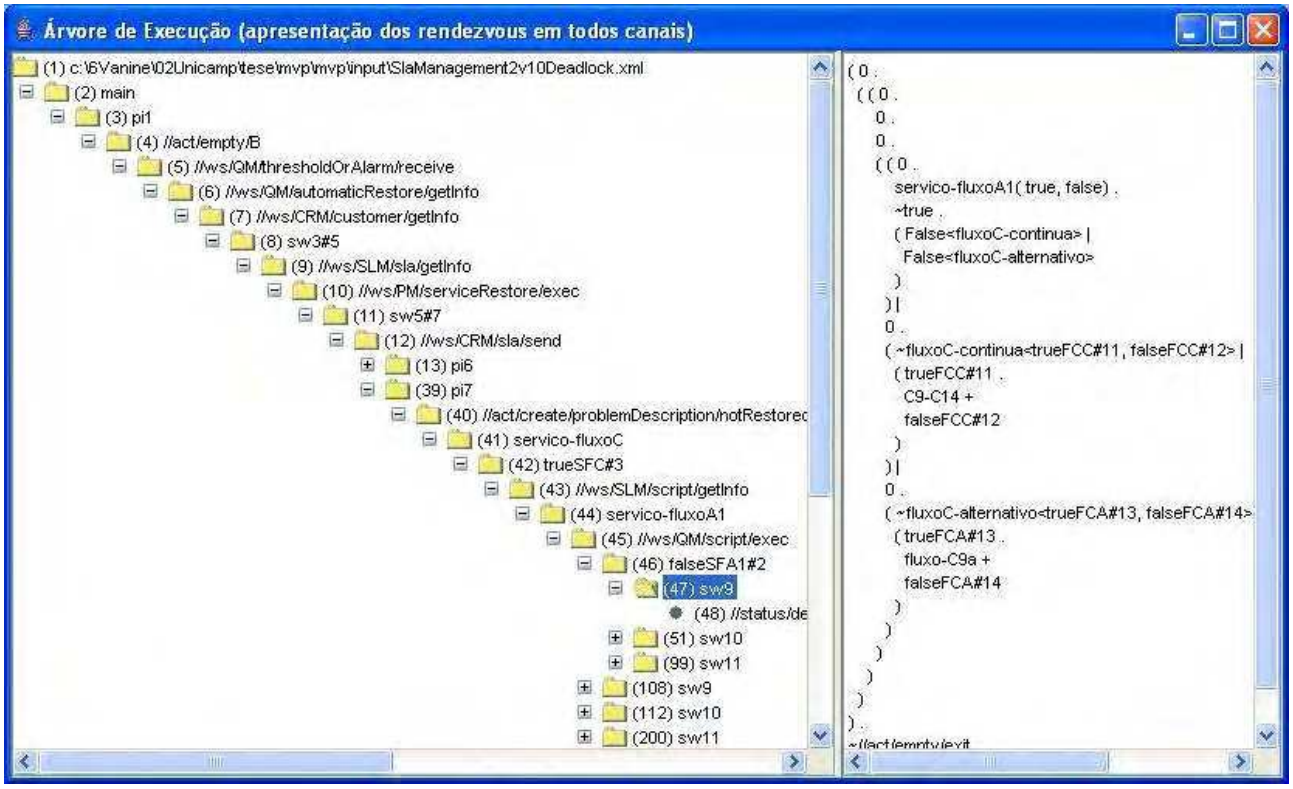

Fig. F.10: Árvore de Execução do programa de Gerência de SLA em *deadlock*.

do fluxo C ou execução do fluxo C alternativo. O primeiro processo está parado na execução da chamada ao processo `True<serviço-fluxoA1>`. Porém, não existe qualquer outro processo com uma escrita neste canal para informar os valores `true` e `false` que, por conseguinte, dê prosseguimento ao processo.

F.9 Conclusão

Este apêndice apresentou um exemplo de aplicação de BPEL para o desenvolvimento de processos fim-a-fim na área de telecomunicações. O modelo usado para a construção do exemplo foi o eTOM do TMForum. A partir das especificações das interações dos processos do modelo de referência, são descritos em (36) alguns exemplos de fluxos do ciclo de vida de um SLA. Através desses fluxos, foi criado um processo, denominado Gerência de SLA, construído como uma composição de *web services* que correspondem aos serviços dos processos do modelo de referência. Em seguida, o processo foi escrito de forma simplificada em BPEL. A partir daí, foi usado o *framework* para as análises e verificações de propriedades do processo.

Apêndice G

Expressão Pi-Calculus do Processo de Gerência de SLA

```
A1==>
 ~//ws/QM/qos/getInfo .
 ~//ws/SLM/sla/getInfo .
 new ( sw1,sw2).
 ( ( ~sw1 +
    ~sw2
  ) |
  ( sw1 .
    ~//act/create/problemDescription/slaReport .
    True<servico-fluxoC> +
    sw2 .
    ~//ws/CRM/sla/send .
    False<servico-fluxoC>
  )
)

A2==>
 ~//ws/QM/usage/getInfo .
 ~//ws/FPM/usageRating/getInfo .
 ~//ws/Billing/bill/getInfo .
 ~//ws/CRM/bill/send

B8-B11==>
 ( ( ~pi6 +
    ~pi7
```

```
   ) |
   ( pi6 .
    //ws/QM/RestoredEventAndQos/receive .
    ~//ws/sla/getInfo .
    new ( sw7,sw8) .
    ( ( ~sw7 +
        ~sw8
      ) |
      ( sw7 .
        ~//ws/CRM/serviceRestored/send +
        sw8
      )
    ) .
    ~//ws/CRM/sla/send .
    False<servico-fluxoC> +
    pi7 .
    ~//act/create/problemDescription/notRestored .
    True<servico-fluxoC>
   )
 )

B5-B7==>
 ~//ws/CRM/customer/getInfo .
 ~//ws/SLM/sla/getInfo .
 new (sw5,sw6) .
 ( ( ~sw5 +
    ~sw6
   ) |
   ( sw5 .
    ~//ws/CRM/sla/send +
    sw6
   )
 )

B2-B11==>
( ~//ws/QM/automaticRestore/getInfo .
  new ( sw3,sw4) .
  ( ( ~sw3 +
      ~sw4
    ) |
    ( sw3 .
      ~//ws/PM/serviceRestore/exec +
      sw4
```

```
    )
  ) |
  B5-B7
) . B8-B11

fluxo-C9a==>
~//ws/FM/slaAssuranceAlarm/send .
~//ws/SLM/rebate/getInfo .
~//ws/CRM/rebate/send

C9-C14==>
~//ws/QM/qos/getInfo .
new ( sw12,sw13) .
( ( ~sw12 +
    ~sw13
  ) |
  ( sw12 .
    True<fluxoC-alternativo> +
    sw13 .
    ( ~//ws/CRM/qos/send |
      ~//ws/SLM/rebate/getInfo .
      ~//ws/Billing/rebate/send .
      ~//ws/CRM/rebate/send
    ) .
    False<fluxoC-alternativo>
  )
)

C4-C14==>
( ( ~sw9 +
    ~sw10 +
    ~sw11
  ) |
  ( sw9 .
    ~//ws/QM/qos/getInfo .
    ~//ws/SLM/sla/getInfo .
    ~//ws/CRM/sla/send .
    ( False<fluxoC-continua> |
      False<fluxoC-alternativo>
    ) +
    sw10 .
    True<fluxoC-continua> +
    sw11 .
```

```
    ~//ws/PM/serviceRestore/exec .
    ~//ws/QM/serviceRestoreResult/send .
    True<fluxoC-continua>
  ) |
  new ( trueFCC,falseFCC) .
  ( ~fluxoC-continua<trueFCC,falseFCC> |
    ( trueFCC .
      C9-C14 +
      falseFCC
    )
  ) |
  new ( trueFCA,falseFCA) .
  ( ~fluxoC-alternativo<trueFCA,falseFCA> |
    ( trueFCA .
      fluxo-C9a +
      falseFCA
    )
  )
)

C2-C14==>
~//ws/SLM/script/getInfo .
~//ws/QM/script/exec .
C4-C14

main==>
( ( ( ~pi1 +
      ~pi2 +
      ~pi3 +
      ~pi4
    ) |
    ( pi1 .
      ~//act/empty/B .
      //ws/QM/thresholdOrAlarm/receive .
      B2-B11 +
      pi2 .
      ~//act/empty/C .
      //ws/CRM/problem/receive .
      True<servico-fluxoC> +
      pi3 .
      ~//act/empty/A1 .
      A1 +
      pi4 .
```

```
      ~//act/empty/A2 .
      A2 .
      False<servico-fluxoC>
    )
  ) |
  new ( trueSFC,falseSFC) .
  ( ~servico-fluxoC<trueSFC,falseSFC> |
    ( trueSFC .
      C2-C14 +
      falseSFC
    )
  )
) . ~//act/empty/exit
```