

Linguagens para Especificação de Workflows

Tatiana Almeida Souza Coelho
Marco Antonio Casanova

Departamento de Informática - PUC/Rio
{tati,casanova}@inf.puc-rio.br

PUC-RioInf.MCC28/02 Novembro, 2002

Resumo

Esta monografia apresenta uma breve discussão sobre algumas linguagens para especificação de workflows encontradas na literatura: a linguagem do projeto WIDE, que visa a execução de workflows distribuídos; a linguagem BPEL4WS, projetada a partir do conceito de serviços Web; e a linguagem XRL, voltada para o tratamento de transações de comércio eletrônico. Após a análise destas linguagens, esta monografia propõe uma extensão da linguagem XRL. A extensão proposta atende aos requisitos da descrição de planos de ação de emergência para o projeto InfoPAE, uma parceira TeCGraf/Petrobras. Ela é uma linguagem baseada em XML e que prevê que um workflow pode ser composto de diversos outros subworkflows, os quais podem ser executados de modo distribuído e muitas vezes independente.

Palavras-Chave: workflow, XML, linguagem de especificação, sintaxe, semântica.

Abstract

This work presents a brief discussion about some languages for workflow specification found in the literature: the WIDE Project's language, that aims at executing distributed workflows; the BPEL4WS language, based on the concept of Web Services; and the XRL language, for the management of electronic commerce transactions. After analysing these languages, this work proposes an extension of the XRL language. The extension proposed meets the requirements of the emergency plan description for the InfoPAE Project, a partnership between TecGraf and Petrobras. The workflow specification language proposed is based on XML and assumes that a workflow can be composed of many other subworkflows, which can be executed in a distributed and, sometimes, independent manner.

Keywords: workflow, XML, specification language, syntax, semantics.

Sumário

1	Introdução	1
2	Projeto WIDE	3
2.1	Arquitetura de WIDE	4
2.2	Modelo de Organização	5
2.3	Modelo de Informação	5
2.4	Modelo de Processo	6
2.5	Tratamento de Exceções	8
2.6	Linguagem de Especificação de Workflow	8
3	A Linguagem BPEL4WS	10
3.1	Construtores Básicos	11
3.2	Atividades	13
3.3	Tratamento de Falhas e Transações em BPEL4WS	14
4	A Linguagem XRL	15
4.1	Arquitetura do XRL/Flower	15
4.2	Sintaxe de XRL	17
4.3	Semântica de XRL	21
4.4	Exemplo em XRL	23
5	Proposta de Extensão da Linguagem XRL	24
5.1	Sintaxe Proposta	24
5.2	Semântica	31
6	Conclusões	32
	Apêndices	36

Lista de Figuras

1	Arquitetura WfMC [7].	1
2	Arquitetura do sistema de gerência de workflow WIDE [3].	4
3	Associação de tarefas a agentes em WIDE [6].	5
4	Conectores em WIDE [2].	7
5	Arquitetura de XRL/Flower, a máquina de workflow que utiliza a linguagem XRL [9]. . .	16
6	Modos de integração em XRL/Flower [9].	16
7	Símbolos usados no mapeamento de um workflow em XRL para redes de Petri [9]. . . .	22
8	Construtor <i>sequence</i> de XRL [9].	22
9	Construtor <i>any_sequence</i> de XRL [9].	23
10	Construtor <i>choice</i> de XRL [9].	23
11	Construtor <i>while_do</i> de XRL [9].	24
12	Representação gráfica do exemplo descrito em XRL [9].	25
13	Construtor <i>call</i>	32
14	Construtor <i>find_executables</i>	33
15	Construtor <i>send</i>	34
16	Construtor <i>receive</i>	34

1 Introdução

Sistemas de gerência de workflow são usados para coordenar processos de negócio. Eles auxiliam empresas na especificação, na execução e no monitoramento de seus processos de maneira eficiente e muitas vezes com economia de tempo e trabalho humano. Além disso, podem ser utilizados na gerência de qualquer aplicação distribuída. Companhias de seguro, empresas de comércio eletrônico, de desenvolvimento de software e empresas do governo são alguns exemplos de empresas que utilizam sistemas de gerência de workflow. O projeto InfoPAE, parceria do laboratório TeCGraf da PUC/Rio com a Petrobras, também utiliza um sistema de gerência de workflow para especificação e execução de planos de ação de emergência (PAE) [1]. Este projeto é a grande motivação desta monografia.

Um processo de negócio normalmente é representado como um workflow, um modelo computado-rizado que especifica tarefas a serem executadas, em determinada ordem e sob determinadas condições. Além disso, um workflow especifica o fluxo de dados entre as tarefas e os usuários responsáveis pela execução das mesmas. Um workflow pode ser representado por meio de um grafo, onde os nós são tarefas ou condições que devem ser testadas para a execução de uma tarefa, e os arcos indicam a seqüência de execução.

Em 1993 foi criado o WfMC (*Workflow Management Coalition Model*) [7], um consórcio para o estudo de padronização de sistemas de workflow. Este consórcio propôs um modelo de referência que permite que diferentes sistemas de workflow sejam interoperáveis.

Esse modelo, apresentado na Figura 1, inclui cinco interfaces para padronização da comunicação entre diversas organizações.

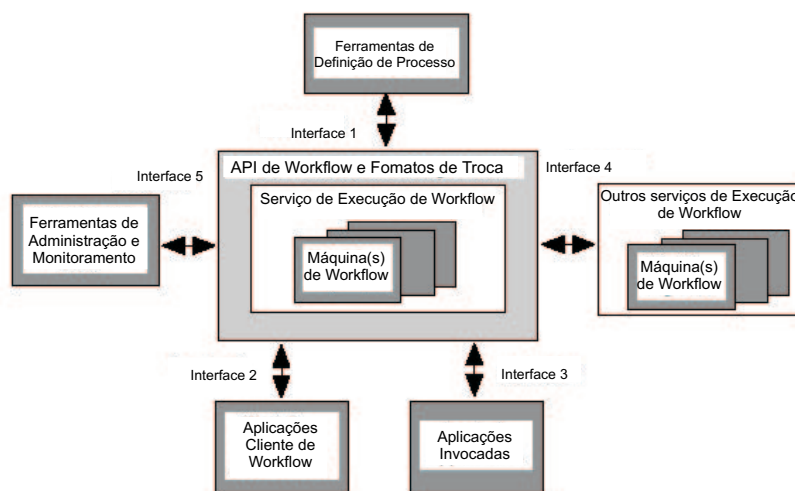


Figura 1: Arquitetura WfMC [7].

A Interface 1 define uma interface padrão entre a ferramenta de definição do processo e a própria máquina de workflow. A Interface 2 define um padrão para a máquina de workflow manter itens de trabalho, os quais são apresentados ao usuário. A Interface 3 é uma interface padrão para permitir que a máquina de workflow invoque uma variedade de aplicações distintas, as quais devem ser executadas para o cumprimento de determinadas tarefas. A Interface 4 define diversos modelos de interoperabilidade, para que organizações diversas possam cooperar entre si. Por fim, a Interface 5 define as funções de monitoramento e controle da execução de um workflow.

Recentemente este consórcio criou uma especificação em XML para a Interface 4, conhecida como Wf-XML. Esta especificação modela os requisitos de transferência de dados entre diferentes máquinas de workflow em XML, permitindo que o estado do processo e os resultados obtidos sejam trocados entre

múltiplas máquinas. No entanto, nenhuma outra interface foi especificada.

Esse modelo de referência do WfMC pode ser visto como um passo em direção à interoperabilidade. No entanto, ele modela uma arquitetura de sistema de workflow monolítica, que não é flexível ou escalável o suficiente para endereçar as necessidades de aplicações de workflow para a Internet e que está centrada na máquina de workflow. Além disso, esse modelo não define uma linguagem de especificação de workflow, a qual é nosso interesse nesse trabalho.

A proposta da linguagem de especificação que será aqui apresentada leva em consideração alguns fatores que julgamos importantes:

- diversas empresas possuem fluxos de trabalho bastante semelhantes, de modo que muitas vezes o workflow de uma empresa A pode ser aplicável ao contexto de trabalho de uma empresa B, sem necessidade de qualquer tipo de mudança
- diversas empresas se comunicam entre si através da troca de informação por meio de um workflow, de modo que este deve ser claro e sua especificação realizada em uma linguagem compreendida por ambas as partes. Padrões como XML são uma boa opção neste caso
- a construção de workflows deve ser uma tarefa fácil, através de uma linguagem simples, concisa e legível. Construtores em XML aumentam a legibilidade dos planos e simplificam o processo de criação destes planos
- como workflows muitas vezes são semelhantes, é importante que se possa facilmente reutilizar pedaços de um workflow em outros workflows similares.

Um workflow, segundo o nosso ponto de vista, pode ser considerado um conjunto de tarefas interrelacionadas, pelo ponto de vista temporal, ou pelos recursos que utilizam ou geram durante execução. Cada tarefa em um workflow tem seu relacionamento com as demais tarefas e, muitas vezes, com recursos que auxiliam os usuários na execução das mesmas ou com recursos geradas por intermédio da execução.

Considerando que um sistema de workflow possui um comportamento inerentemente distribuído, cada workflow pode ser dividido em partes, denominadas subworkflows, a serem executadas distribuídamente, por usuários distintos. Dessa forma, um workflow pode ser composto por um conjunto de subworkflows, de forma recursiva.

De acordo com essa visão, se a partir de um sistema de workflow for disparada a execução de um workflow composto de subworkflows, então deve haver um mecanismo de comunicação confiável, que torne possível o envio e o recebimento de mensagens entre os usuários executores, ou melhor, entre as partes (subworkflows) em execução, já que as partes são inter-dependentes.

Uma tarefa em um sistema de workflow é a menor unidade de execução. Ela é indivisível e, por isso, não pode ser quebrada em unidades menores a serem executadas por usuários distintos pela rede.

Cada (sub)workflow, assim como tarefas ou grupos de tarefas, pode possuir a ele associado um conjunto de um ou mais recursos. Um recurso pode ser um documento texto ou uma imagem, por exemplo.

Assim como ocorre em WIDE, os (sub)workflows podem estar armazenados separadamente dos recursos que utilizam (documentos, por exemplo) e da informação de seus executores. Dessa forma, um sistema de workflow deve possuir um banco de dados de tarefas e dependências entre tarefas, o qual descreve os workflows existentes, um banco de dados de recursos, que descreve o relacionamento entre os recursos e cada uma das tarefas e (sub)workflows, quando este for o caso, e um banco de dados de usuários, que descreve os usuários cadastrados para a execução dos workflows.

Além deste tipo de recurso, podem estar associados a cada (sub)workflow um conjunto de variáveis. Estas variáveis podem ser apenas variáveis internas de controle ou também variáveis que determinem a quais situações o (sub)workflow se aplica, ou seja, sob quais circunstâncias é válida a sua execução. A um conjunto de variáveis que determinam um conjunto de situações nas quais um determinado

(sub)workflow pode/deve ser executado, com os respectivos valores associados, chamamos condição de disparo.

Desta forma, a condição de disparo, quando existente, para um (sub)workflow, é especificada por meio de um conjunto de pares (parâmetro, valor), de forma que cada um dos parâmetros e seu valor esteja na lista das variáveis especificadas no workflow. Se um subworkflow não possui uma condição de disparo especificada, então ele pode ser executado em qualquer instante durante a execução do workflow no qual foi declarado.

Cada subworkflow declarado em um workflow pode ser chamado através dele e pode, ainda, ser tratado como uma subrotina, que recebe valores de entrada, podendo também gerar valores de saída. Estes valores, tanto de entrada quanto de saída, são valores também provenientes dos parâmetros declarados. No entanto, não são valores que determinam a execução do subworkflow.

De forma geral, um (sub)workflow pode ser descrito por uma quádrupla (T, C, P, R), onde T representa o conjunto de tarefas, C a condição em que o (sub)workflow pode ser executado (condição de disparo), P os parâmetros de entrada necessários a sua execução e R os recursos vinculados.

Essa monografia define como um sistema de workflow pode ser organizado e executado de forma distribuída, propondo uma linguagem de especificação de workflow que seja modelada de acordo com a estrutura distribuída do sistema. O foco nessa monografia não está na interoperabilidade entre diversas organizações, mas sim em sistemas de workflow que atendam às necessidades de empresas isoladamente, apesar de que a interoperabilidade, conforme será apresentado a seguir, venha de maneira natural, permitindo que organizações distintas possam se comunicar através da linguagem proposta.

Essa monografia encontra-se organizada da seguinte maneira. A seção 2 apresenta o Projeto WIDE, que define um modelo de workflow baseado em 3 módulos distintos. A seção 3 apresenta uma linguagem de especificação de processos de negócio baseada no conceito de *Web Services*. A seção 4 apresenta a XRL, uma linguagem de especificação de workflow projetada para a Internet, no intuito de que organizações distintas possam cooperar em termos de modelo de negócio. A seção 5 propõe um sistema de workflow e uma variante de XRL. Por fim, a seção 6 apresenta as conclusões deste trabalho. O Apêndice 1 apresenta um exemplo de processo de negócio descrito em BPEL. O Apêndice 2 apresenta o exemplo descrito na seção 4 de acordo com a linguagem XRL. O Apêndice 3 apresenta a DTD da linguagem XRL estendida. O Apêndice 4, por sua vez, apresenta este mesmo exemplo, escrito agora conforme a linguagem XRL estendida proposta.

2 Projeto WIDE

O projeto WIDE (*Workflow on Intelligent Distributed database Environment*) [3, 5] propõe um modelo conceitual que visa facilitar o projeto de especificações de workflow. Esse modelo, que é distribuído e possui suporte a exceção e a transações, está dividido em três modelos:

- Modelo de Organização: módulo que descreve as equipes (agentes) envolvidas na execução das atividades descritas na especificação do workflow
- Modelo de Informação: módulo que descreve as informações associadas às atividades a serem executadas. As informações podem ser, por exemplo, documentos
- Modelo de Processo: terceiro e último módulo que descreve as atividades (tarefas) a serem executadas e em que ordem a execução deve ocorrer. Esse módulo descreve ainda como os outros dois módulos são combinados a esse para a geração do sistema completo de workflow.

Sendo um modelo distribuído, WIDE permite que algumas partes do workflow sejam executadas remotamente. Neste caso, o ambiente WIDE se encarrega de sincronizar as diferentes partes dos diferentes sites [3].

O modelo de objeto distribuído em WIDE é utilizado para criar módulos de servidor de workflow e objetos de dados que podem ser transparentemente acessados por múltiplas instâncias de workflow ou mesmo por múltiplos workflows distintos. Estes workflows podem estar rodando em uma mesma máquina ou em máquinas diferentes distribuídas pela rede.

No que diz respeito ao tratamento de exceções, WIDE utiliza regras ativas, baseadas no paradigma evento-condição-ação. Atividades compensatórias são definidas em WIDE para o caso em que atividades necessitem ser desfeitas.

A primeira subseção a seguir apresenta uma descrição da arquitetura de WIDE. As subseções seguintes apresentam uma descrição mais detalhada de cada um dos modelos de WIDE. A última subseção apresenta a linguagem de descrição de workflow definida para este modelo.

2.1 Arquitetura de WIDE

A arquitetura do sistema de gerência de workflow de WIDE é apresentada na Figura 2.

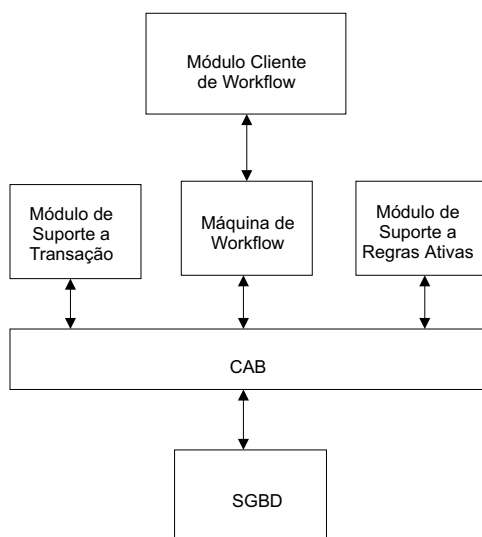


Figura 2: Arquitetura do sistema de gerência de workflow WIDE [3].

A camada inferior da arquitetura é formada pelo sistema de gerência de banco de dados (SGBD). O sistema escolhido para esse propósito foi o Oracle. A CAB, Camada de Acesso Básico, é a camada que permite que a camada do servidor, formada pelo módulo de Suporte a Transação, pela máquina de workflow e pelo módulo de Suporte a Regras Ativas, acesse o banco de dados através de uma interface orientada a objetos. Além disso, a camada CAB mapeia os acessos da camada do servidor ao banco de dados para uma interface relacional.

Os módulos de Suporte a Transação e de Suporte a Regras Ativas são módulos ortogonais. Já a máquina de workflow é a principal componente do sistema de gerência de workflow. Ela se comunica com a Camada de Acesso Básico para acesso ao banco de dados, com o módulo de Suporte a Transação para fornecer conceitos de transação avançados e com o módulo de Suporte a Regras Ativas para permitir comportamento reativo, no caso de ocorrência de falhas, por exemplo.

Na camada cliente, o módulo de Cliente de Workflow fornece uma interface interativa para usuários finais do sistema, comunicando-se apenas com a máquina de workflow. Dessa forma, o cliente do sistema comunica-se diretamente com a máquina de workflow e essa faz todas as operações necessárias para que o workflow seja executado.

A infra-estrutura de gerência de workflow de WIDE pode consistir de uma hierarquia de máquinas de workflow. Nesta hierarquia, cada máquina está vinculada a um domínio, derivado da estrutura organizacional em que o workflow foi implementado, ou da estrutura geográfica da organização.

2.2 Modelo de Organização

O propósito do Modelo de Organização dentro do modelo WIDE é descrever as pessoas envolvidas na execução de uma instância do workflow. Dessa forma, este modelo é extremamente dependente da estrutura da empresa à qual o workflow pertence. Daí a grande vantagem de WIDE em separar o processo de especificação do workflow do processo de definição de quais pessoas estão a ele vinculadas. Deste modo, qualquer alteração no corpo de funcionários da empresa não implica em alteração no fluxo de tarefas. Além disso, como o workflow em si independe da estrutura da empresa, um mesmo workflow pode ser utilizado por diferentes empresas.

De acordo com o Modelo de Organização, papéis podem ser definidos para cada tarefa. Assim, uma determinada tarefa apenas pode ser executada por um pessoa ou grupo de pessoas com aquele papel. Depois de definidos os papéis, devem ser definidas as pessoas que estão associadas a cada um deles.

A Figura 3 apresenta o relacionamento entre o Modelo de Organização e o Modelo de Processo em WIDE.

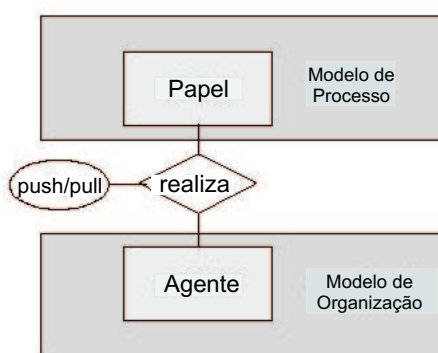


Figura 3: Associação de tarefas a agentes em WIDE [6].

Um agente em WIDE é qualquer entidade que pode realizar uma tarefa do workflow, seja esta entidade uma pessoa, um grupo de pessoas ou mesmo uma aplicação. Cada entidade é representada no sistema por um *task desk*, o qual contém todas as tarefas a serem realizadas por aquela entidade.

A associação de tarefas a agentes pode ser feita diretamente, de modo que o sistema determina o melhor agente a executá-la e associa a tarefa a seu *task desk*. Por outro lado, essa associação pode ser dirigida pelo usuário, de forma que a tarefa é vinculada a um usuário-chave, o qual pode entregá-la a outras entidades do sistema. Nestes dois casos, a estratégia pode ser uma *push strategy*, de acordo com a qual o usuário recebe do sistema ou do usuário-chave a tarefa a executar, ou pode ser uma *pull strategy*, de acordo com a qual o sistema coloca a tarefa em um *task desk* compartilhado e o usuário, caso deseje, se encarrega de trazer a tarefa do *task desk* compartilhado para o seu próprio *task desk*. Para maiores informações a respeito do processo de associação de tarefas a agentes, veja Seção 3.1.5 de [5].

2.3 Modelo de Informação

O Modelo de Informação pode ser visto como um pacote de itens de informação associados tanto à especificação do workflow quanto à sua execução. Assim como o Modelo de Organização, o Modelo de Informação procura ser o mais independente possível da especificação do workflow.

A informação associada a um workflow em WIDE pode ser uma variável ou um item de documentação. Variáveis são os elementos básicos de informação de um workflow e podem ser utilizadas tanto para armazenar informação do próprio workflow quanto para controlar sua execução. Desta forma, estas variáveis não podem ser compartilhadas entre instâncias distintas de um mesmo workflow.

Por outro lado, em WIDE existe a noção de variáveis compartilhadas, que podem ser como ponteiros para uma variável real. Estas variáveis, sendo externas ao modelo, podem ser compartilhadas entre instâncias distintas.

Um item de documentação é um documento (de texto ou imagem, por exemplo), um formulário ou folders, utilizado, criado ou modificado por um usuário durante a execução de uma determinada tarefa do workflow, e que pode ser acessado no mesmo instante de tempo por instâncias distintas de um mesmo workflow, assim como as variáveis compartilhadas. Por exemplo, um formulário pode ser modificado pelo usuário durante execução e sobre ele podem ser impostas condições que ditam o estado de execução da tarefa, ou seja, condições sobre as quais uma tarefa é considerada completada ou não.

Para controlar o acesso a estes itens e a estas variáveis, WIDE utiliza mecanismos de bloqueio. Cada bloqueio é associado a uma tarefa, de modo que um documento, por exemplo, pode estar bloqueado de forma exclusiva para uma tarefa e estar sendo utilizado em outra tarefa sem bloqueio algum.

Uma informação em WIDE também pode ser temporal, como um instante de tempo, um intervalo de tempo ou a granularidade na qual a informação será disponibilizada. Informação temporal para workflows pode ser utilizada tanto para a verificação se um determinado instante foi alcançado, quanto para a verificação se um intervalo de tempo acabou, por exemplo.

2.4 Modelo de Processo

O Modelo de Processo em WIDE descreve toda a estrutura do workflow, suas tarefas, a ordem de execução, a relação entre as tarefas e como o workflow é executado.

Nesse modelo, são três os principais elementos:

- tarefas: unidades elementares de execução
- conectores: elementos que conectam tarefas entre si
- outras unidades de execução: unidades que facilitam principalmente o reuso.

O processo de execução de uma tarefa em WIDE ocorre da seguinte maneira. A máquina de workflow recebe uma requisição para iniciar uma instância de um workflow. O Interpretador, um módulo do sistema de gerência de workflow, após alguns passos de inicialização, recebe o controle da execução e determina qual(uais) a(s) tarefa(s) a ser(em) inicialmente executada(s). Esta informação é então retornada à instância em execução.

Nesse ponto, a(s) tarefa(s) é inicialmente criada(s) no sistema. Logo após sua criação, ela é associada a um agente da organização, de acordo com o papel descrito. Esta associação é realizada por outro módulo do sistema de gerência, o Despachante de Tarefas.

Uma vez associada a um usuário, a tarefa permanece em seu *task desk* até que ele a abra para execução. Tendo aberto a tarefa para execução, o usuário tem acesso às informações a ela relacionadas e pode modificá-las. Caso o usuário não tenha recebido diretamente essa tarefa para execução, mas a escolhido a partir do *task desk* compartilhado, ele pode devolvê-la.

Para finalizar uma tarefa, o usuário pode marcá-la como completada (caso esta operação possa ser realizada) ou pode cancelá-la. Neste último caso, a instância do workflow continua em execução, sabendo que determinados resultados não estarão disponíveis pelo fato de certas tarefas terem sido canceladas.

Quando uma tarefa é terminada por um usuário, seja quando completada ou cancelada, ela é retirada do seu *task desk*. Uma tarefa pode ainda ser retirada do *task desk* de um usuário quando:

- ela é delegada a outro usuário no sistema, automaticamente pelo sistema ou manualmente pelo próprio usuário
- ela é retornada ao *task desk* compartilhado, no caso de ter sido retirada pelo usuário deste espaço (*pull strategy*)
- ela é retornada a quem a delegou.

No Modelo de Processo, WIDE utiliza os conectores definidos pela WfMC para descrever a interação entre as tarefas. Estes conectores, além de uma descrição textual, podem ser representados graficamente, conforme apresentado em [2]. Em WIDE eles são distinguidos em tipos diferentes e as vezes combinados entre si. A Figura 4 apresenta os conectores utilizados em WIDE.

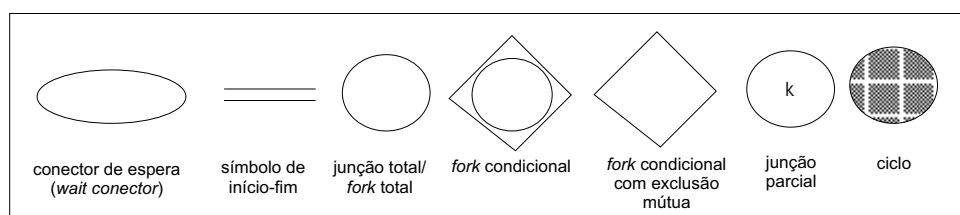


Figura 4: Conectores em WIDE [2].

O símbolo de espera tem a ele associado uma condição, que pode ser baseada, por exemplo, na disponibilidade de dados, em informação temporal e na ocorrência de um determinado evento.

Os símbolos de início-fim marcam a criação e o fim da execução de uma determinada instância de workflow. Cada workflow tem um símbolo de início e vários símbolos de término de execução. Quando um símbolo de término torna-se ativo, a instância em execução é considerada terminada.

WIDE utiliza o nome *fork* para o símbolo de *split* definido pela WfMC. Em WIDE, *forks* podem ser totais, indicando que o término da execução dos predecessores habilita os sucessores para execução; condicionais, aos quais estão associadas condições que tornam prontas para execução as tarefas cuja avaliação da condição foi verdadeira; condicionais com exclusão mútua, de modo que apenas uma condição pode ser verdadeira em determinado instante.

Conectores de junção, por sua vez, podem ser totais, parciais ou circulares. Conectores circulares indicam que a tarefa sucessora torna-se pronta para execução a cada vez que uma tarefa predecessora é terminada.

Da necessidade de se encontrar aspectos comuns entre tarefas a serem executadas e do fato de que não é sempre possível prever o número de tarefas que realizam o mesmo trabalho a serem executadas em paralelo, surgiu em WIDE a noção de multi-tarefa. Neste caso, os aspectos comuns podem ser traduzidos como parâmetros que variam entre as diversas instâncias e a quantidade de tarefas em paralelo pode ser resumida em uma mesma tarefa sendo instanciada um número necessário de vezes. Desta forma, uma multi-tarefa em WIDE tem como parâmetro de entrada o número de instâncias de tarefas que devem ser executadas.

Em WIDE também existe o conceito de subprocesso. Ele surgiu da necessidade de se modularizar a definição de um workflow, tornando as partes auto-contidas e passíveis de serem reutilizadas em diferentes conceitos. Um subprocesso pode ser visto, no que se refere à distribuição, como a unidade de distribuição, de modo que cada um deles pode ser executado por um usuário/equipe distinto. A nossa proposta a ser apresentada utiliza o mesmo conceito, como será visto adiante.

Além do conceito de multi-tarefa e subprocesso, é utilizado em WIDE o conceito de super-tarefa. Uma super-tarefa, assim como um subprocesso, é formada por um conjunto de tarefas interligadas por meio de conectores definidos pela WfMC. No entanto, uma super-tarefa não pode, ao contrário de um subprocesso, ser executada diretamente, ela não tem parâmetros e deve ser executada na mesma máquina

do subprocesso/workflow ao qual pertence. Do ponto de vista de transações, uma super-tarefa pode ser vista como uma unidade de execução com propriedades de atomicidade e isolamento.

O Modelo de Processo de WIDE possui a característica intrínseca de distribuição, de modo que cada unidade de execução de uma instância de workflow pode ser executada em uma máquina distinta.

Dessa forma, a separação em modelos de WIDE permite que um mesmo workflow seja compartilhado entre empresas distintas e que a alteração no corpo de trabalho de uma empresa, por exemplo, não afete a especificação do workflow. A separação clara de pessoas, tarefas e informação associada é bastante útil quando se pensa em reuso. No entanto, a linguagem de especificação de workflow de WIDE deixa muito a desejar, como apresentamos na subseção 2.6.

2.5 Tratamento de Exceções

Exceções em WIDE são tratadas através das chamadas regras ativas. As exceções podem estar relacionadas tanto com o Modelo de Processo quanto, por exemplo, com alterações organizacionais. Uma exceção em WIDE é especificada através do paradigma evento-ação-condição.

De acordo com esse paradigma, o evento indica quando a exceção ocorreu. Eventos podem ser: temporais, expressos em termos de intervalos de tempo ou deadlines, por exemplo; externos, quando são causados por aplicações externas; ou eventos de dados, os quais estão relacionados com atualizações nas variáveis do workflow, como variáveis de status.

A condição do paradigma evento-ação-condição é um predicado sobre o estado do banco de dados do workflow, que deve ser avaliado para se identificar a ocorrência de exceções e quais ações devem ser tomadas. Estas condições podem ser descritas em termos de valores de variáveis e de sentenças temporais.

Uma vez detectada a ocorrência de uma exceção, através da avaliação da condição, uma ação é disparada pela máquina de workflow para a execução de um serviço particular. Ações podem ser apenas informativas, consistindo de notificações a um ou mais usuários, ou podem ser corretivas, consistindo da inicialização, cancelamento ou finalização de tarefas, ou da execução de aplicações externas. Por exemplo, um sistema de workflow em uma rede móvel pode definir que se um usuário que estava executando uma tarefa se desconectar da rede e demorar mais do que um tempo para se reconectar, um outro usuário deve ser invocado para a execução daquela tarefa.

2.6 Linguagem de Especificação de Workflow

A linguagem de descrição de workflow de WIDE chamada WFDL (*WIDE Workflow Description Language*) consiste de duas partes:

- uma seção de declaração, onde os itens de informação e as tarefas são definidos
- uma seção de definição da estrutura do fluxo, na qual o workflow propriamente dito é descrito.

Em [2] é apresentado um exemplo de um processo militar. O quadro abaixo apresenta esse processo descrito em WFDL.

```
PDL definition of Military enrollment

PROCESS "Military enrollment" FILE milita.pro
DESCRIPTION 'Compulsory enrollment procedure of a military office.'
DECLARATION
  CONSTANTS
    NONE
  VARIABLES
```

```

VAR "Physical Test Result" BOOLEAN NULL
VAR "Psychological Test Result" BOOLEAN NULL
VAR "Drafted" BOOLEAN NULL
VAR "Applicant Data" STRING ""
KEYS
KEY "Applicant SSN" INTEGER 0
KEY "Applicant Name" STRING ""
INFOS
FORM "Application"
  applica.for
EXPRESSIONS
COND "@[Physical Test Result]" : NULL ENDCOND
COND "@[Psychological Test Result]" : NULL ENDCOND
TASKS
TASK "Get and record data"
  ROLE AdministratorOfficer
  SERVER Server@Case
  USER User@Case
  DESCRIPTION 'A new candidate is registered.'
  IN
    NONE
  OUT
    FORM "Application"
    MANDATORY
ENDTASK
TASK "Physical test"
  ROLE Doctor
  SERVER Server@Case
  USER User@Case
  DESCRIPTION 'The candidate is submitted to physical
    test and the result is registered in
    the "Application" form.'
  IN
    FORM "Application"
    MANDATORY
  OUT
    FORM "Application"
    MANDATORY
ENDTASK
TASK "Psychological test"
  ROLE Psychologist
  SERVER Server@Case
  USER User@Case
  DESCRIPTION 'The applicant is submitted to psychological
    test and the result is registered in the
    "Application" form.'
  IN
    FORM "Application"
    MANDATORY
  OUT
    FORM "Application"
    MANDATORY
ENDTASK
TASK "Draft"
  ROLE AdministratorOfficer
  SERVER Server@Case
  USER User@Case
  DESCRIPTION 'The candidate is enrolled and the
    "Application" form is completed.'
  IN
    FORM "Application"
    MANDATORY
  OUT
    FORM "Application"
    MANDATORY
ENDTASK
TASK "Reject"
  ROLE Officer
  SERVER Server@Case
  USER User@Case
  DESCRIPTION 'The candidate is refused and the

```

```

        "Application" form is completed.'
    IN
        FORM "Application"
        MANDATORY
    OUT
        FORM "Application"
        MANDATORY
    ENDTASK
ENDDECLARATION
TASK "Get and record data"
IF TASK "Get and record
    data" THEN
    TASK "Physical test";
IF TASK "Physical test" THEN
    COND "@[Physical Test Result]";
IF COND "@[Physical Test Result]" THEN
    TASK "Psychological test";
IF TASK "Psychological test" THEN
    COND "@[Psychological Test Result]";
IF COND "@[Psychological Test Result]" THEN
    TASK "Draft";
IF (OR (NOT COND "@[Physical Test Result]")
    (NOT COND "@[Psychological Test Result]")) THEN
    TASK "Reject";
(OR TASK "Draft" TASK "Reject")
END

```

Observe que esta linguagem é bastante informal e não segue nenhum padrão como o da linguagem XML. No entanto, WIDE apresenta um modelo extremamente rico.

3 A Linguagem BPEL4WS

Business Process Execution Language for Web Services (BPEL4WS) [4, 8, 12] é uma linguagem da IBM de especificação de processos de negócio, baseada no conceito de *Web Services*¹. O objetivo é que ela se torne a base para um padrão de composição de *Web Services* (o processo em si). A descrição de um processo de negócio em BPEL4WS por ser vista como a definição de um workflow, neste caso criado sobre o conceito de *Web Services*.

A sintaxe da linguagem BPEL4WS é descrita através de um XML Schema, definindo o comportamento dos processos, no que diz respeito às interações entre o processo em si e seus parceiros de negócio, representados através dos *Web Services*. BPEL permite a definição de processos de negócio que fazem uso de *Web Services* e processos de negócio que externalizam suas funcionalidades como *Web Services*. Um serviço Web em BPEL pode ser visto como um subprocesso.

Todas as expressões booleanas em BPEL são escritas em XPath 1.0, linguagem que dá suporte para a manipulação dos dados envolvidos nas transações. No entanto, como BPEL4WS está definida sobre o conceito de *Web Services*, ela fica dependente desta tecnologia que ainda está sendo estudada e é, por enquanto, pouco utilizada.

O Apêndice 1 apresenta um exemplo bastante simples de um processo de negócio escrito em BPEL. Neste exemplo, um agente de viagem especifica um processo de negócio chamado *ticketOrder*. O objetivo deste exemplo é permitir que o agente receba de um cliente um itinerário, peça os tickets aéreos referentes ao itinerário informado e receba estes tickets da companhia aérea, para a posteriormente entrega do ticket ao cliente.

¹ *Web Services* são aplicações de processo de negócio modulares, auto-contidas, baseadas nas tecnologias padrões da indústria: WSDL, UDDI e SOAP [8].

3.1 Construtores Básicos

Um processo de negócio especifica a ordem das operações de uma coleção de *Web Services*, os dados compartilhados entre estes *Web Services* - através dos chamados *containers* -, os gerenciadores de falhas e as atividades compensatórias a serem executadas em determinados casos, com relação a tarefas já completadas. A estrutura básica de um processo descrito de acordo com a linguagem BPEL4WS é apresentada no quadro a seguir. Esta definição básica pode ser vista como um template para a criação de instâncias de processos de negócio.

Nas sintaxes apresentadas a seguir, os símbolos '*', '+' e '?' que ocorrem depois de uma tag representam, respectivamente, que esta tag pode ocorrer zero ou mais vezes, uma ou mais vezes e nenhuma vez ou uma única vez. Esta é a nomenclatura encontrada em todos os artigos lidos sobre BPEL4WS.

```
<!-- definições de mensagens, porta, tipo de
      link de serviço, partners... -->
<process name="ncname"
  targetNamespace="uri"
  queryLanguage="anyURI"?
  expressionLanguage="anyURI"?
  suppressJoinFailure="yes|no"?
  enableInstanceCompensation="yes|no"?
  abstractProcess="yes|no"?
  xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/">
  <partners>?
    <!-- Note: At least one role must be specified. -->
    <partner name="ncname" serviceLinkType="qname"
      myRole="ncname"? partnerRole="ncname"?>+
    </partner>
  </partners>
  <containers>?
    <!-- Note: The message type may be indicated with the messageType
      attribute or with an inlined <wsdl:message> element within. -->
    <container name="ncname" messageType="qname"?>
      <wsdl:message name="ncname">?
        ...
      </wsdl:message>
    </container>
  </containers>
  <correlationSets>?
    <correlationSet name="ncname" properties="qname-list"/>+
  </correlationSets>
  <faultHandlers>?
    <!-- Note: There must be at least one fault handler or default. -->
    <catch faultName="qname"? faultContainer="ncname"?>*
      activity
    </catch>
    <catchAll>?
      activity
    </catchAll>
  </faultHandlers>
  <compensationHandler>?
    activity
  </compensationHandler>
  activity
</process>
```

A tag *partners* define as entidades envolvidas no processo de negócio e que interagem entre si. Na verdade, cada *partner* em BPEL4WS é um serviço com o qual o processo interage. Ele é descrito por um nome e tem a ele associados um papel e um tipo de link de serviço. A definição de um *partner* especifica qual papel do tipo de link de serviço o processo aceita (atributo *myRole*) e qual papel tem que ser aceito pelo *partner* (especificado no atributo *partnerRole*). Aceitar um papel em BPEL4WS significa ter a obrigação de fornecer os *Web Services* correspondentes. Desta forma, os serviços Web que são

esperados pelo processo a partir de um partner são referenciados pelo atributo *partnerRole*, enquanto que os serviços fornecidos pelo processo e que o *partner* pode contar com e usar são referenciados pelo atributo *myRole*.

Um tipo de link de serviço identifica um tipo de porta, que por sua vez define as operações possíveis. Este tipo de link caracteriza o relacionamento entre dois serviços através de seus papéis.

A informação de um processo é passada entre suas diversas atividades de um modo implícito, através do compartilhamento de um espaço de dados global, chamado de *container*. Os *containers* são as variáveis de estado de um processo, que armazenam não apenas as mensagens trocadas entre o processo e seus parceiros, mas também as mensagens persistidas pelo processo. Desta forma, o estado de um processo descrito em BPEL4WS é tratado como uma coleção de mensagens depositadas no *container*. Para que o estado de um processo seja atualizado, BPEL4WS define o construtor *assign*, que pode conter qualquer número de atribuições. Segue o trecho de XML *Schema* que define o construtor *container*.

```
<containers>
  <container name="ncname" messageType="qname"?>+
    <wsdl:message name="ncname"?>?
    ...
  </wsdl:message>
</container>
</containers>
```

Um *container* em BPEL4WS possui um nome e está sempre vinculado a um tipo de mensagem. Uma mensagem dentro do *container*, quando existente, é descrita segundo WSDL (*Web Services Description Language*).

O *assign* é uma tarefa atômica, bastante comum dentro de um processo de negócio. Se falhas ocorrerem durante o processamento de um construtor *assign*, os *containers* de destino, especificados no lugar de *to-spec*, permanecem inalterados, no mesmo estado em que se encontravam no início da operação. Esta atividade copia um valor da fonte (*from-spec*) para o destino (*to-spec*).

```
<assign standard-attributes>
  standard-elements
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>
```

Os atributos padrões, referenciados no quadro anterior como *standard-attributes*, são os seguintes:

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

A condição de junção é usada para especificar requisitos sobre caminhos paralelos alcançando uma atividade. Esta condição é uma expressão booleana escrita em XPath. O atributo *suppressJoinFailure* indica se uma falha de junção deve ser omitida quando ela ocorre. O valor default deste atributo é *no*.

Os elementos padrões, por sua vez, são os que seguem:

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

Esses elementos *source* e *target* são necessários para a sincronização entre os links. Cada link definido possui um nome, o qual é usado como valor do atributo *linkName*. Cada elemento *source* associado com uma atividade deve utilizar um *linkName* diferente. O mesmo é válido para os elementos *target*. Cada elemento *source* pode, opcionalmente, especificar um condição de transição, que é também uma expressão booleana. Atividades em BPEL4WS podem declarar elas mesmas como sendo fonte(*target*) de um ou mais link, incluindo um ou mais elementos *source(target)*.

Quando mensagens são trocadas entre parceiros de negócio, elas tipicamente carregam alguns dados que são usados para correlacionar uma mensagem com o processo de negócio apropriado. Este dado de correlação é chamado em BPEL4WS de *property*.

BPEL4WS possui a característica de tratar a ocorrência de falhas. Isso pode ser feito através do construtor *faultHandlers*. Cada *faultHandler* define a atividade que deve ser invocada no caso da ocorrência de falha na execução de uma outra atividade. Quando uma falha ocorre, o processamento normal é interrompido e o controle é transferido para o gerenciador de falhas. Cada tag *catch* intercepta uma falta específica. Se a falta não for nenhuma das faltas interceptadas pelas tags *catch* já declaradas, a falta é reconhecida pela tag *catchAll*.

Por outro lado, *compensationHandler* em BPEL é o gerenciador de atividades compensatórias. Ele pode ser invocado a partir do gerenciador de falhas, para desfazer atividades já completadas quando da ocorrência de falhas.

3.2 Atividades

Uma atividade *activity* em BPEL4WS, por sua vez, pode ser representada por qualquer um dos seguintes construtores:

- *receive*: bloqueia o processo até que a mensagem esperada chegue ao seu destino
- *reply*: este construtor, cujo significado está vinculado ao modo de interação assíncrono, permite que um processo de negócio envie uma mensagem, em resposta a uma outra mensagem recebida através de *receive*. Estes dois construtores são como uma operação de requisição-resposta
- *invoke*: permite que o processo de negócio invoque uma operação oferecida por um serviço em um tipo de porta
- *assign*: conforme apresentado, tem como objetivo permitir a atualização dos dados de um *container*
- *throw*: utilizado para sinalizar explicitamente a ocorrência de uma falha
- *terminate*: faz com que todas as atividades do processo em execução sejam abandonadas, não permitindo que sejam executadas atividades compensatórias ou de tratamento de falhas
- *wait*: força que o processo aguarde por um determinado período ou até que um certo tempo tenha se passado, de acordo com a condição especificada
- *empty*: indica a execução de nenhuma atividade. Pode ser útil em casos em que falhas devem ser percebidas, mas que nada pode se fazer para tratá-las

- *sequence*: construtor que define uma seqüência de execução de atividades. As atividades descritas neste construtor são executadas exatamente na mesma ordem em que aparecem. Uma tarefa apenas pode começar a ser executada se a anterior já completou sua execução
- *switch*: permite que se escolha pela execução de uma das atividades dentro dele declaradas
- *while*: possui a mesma semântica de um operador *while* de uma linguagem de programação
- *pick*: este tipo de atividade especifica um conjunto completo de mensagens que podem ser recebidas de um mesmo parceiro ou de diferentes parceiros de negócio. Quando uma das mensagens especificadas é recebida, esta atividade é completada e o processamento do processo de negócio onde ela está definida continua. No entanto, pode-se especificar que o processamento continue se nenhuma mensagem é recebida em um determinado tempo
- *flow*: define um fluxo paralelo de execução, implicando em execução concorrente e síncrona
- *scope*: permite a definição de uma atividade aninhada com seus próprios gerenciadores de falhas e de atividades compensatórias. Desta forma, escopo é uma atividade estruturada que permite o agrupamento de atividades, definindo um contexto de execução comum através dos gerenciadores de falhas e de atividades compensatórias. Quando uma falha ocorre dentro de um escopo, o processamento dentro do escopo é interrompido e a falha é passada para o gerenciador de falhas. A atividade aninhada dentro deste gerenciador tenta corrigir a situação, de modo que o processamento possa continuar fora do escopo ou que alternativas possam ser tomadas para que o processo possa ser completado. As tarefas já completadas dentro do escopo são então desfeitas pelo gerenciador de compensação, invocado de dentro do gerenciador de falhas. Tarefas executadas dentro de um escopo são todas completadas ou todas compensadas. Escopos em BPEL4WS podem ser aninhados dentro de outros escopos
- *compensate*: esta atividade permite que seja invocado um gerenciador de atividades compensatórias de dentro de um escopo (definido pelo construtor anterior). Ela desfaz o efeito das atividades já completadas dentro do escopo (atividades irreversíveis).

3.3 Tratamento de Falhas e Transações em BPEL4WS

Uma das vantagens de BPEL4WS é o tratamento de falhas e a possibilidade de se executar atividades compensatórias, de acordo com a semântica da aplicação. Isso resulta na característica que é chamada de *Long-Running (Business) Transactions* (LRTs).

Um conjunto de atividades definidas dentro de um escopo em BPEL pode ser visto como uma transação. Atividades dentro de um escopo são todas completadas ou todas compensadas. Não há bloqueio de recursos dentro de um escopo. Ao contrário, transações tradicionais como as de banco de dados alocam, por meio de bloqueios, os recursos necessários para a execução de cada transação. No entanto, estas transações tradicionais são de pequena duração e liberam os recursos rapidamente.

Como escopos podem ter escopos aninhados, existe em BPEL um protocolo de concordância (*agreement protocol*) entre um escopo e seu escopo-pai, cujo objetivo é determinar o resultado da transação [8]. Este protocolo em BPEL assume que um escopo e todos os seus escopos aninhados estão contidos dentro de um único processo e são hospedados por uma única máquina BPEL. Não existe, até então, um mecanismo de coordenação distribuída relativo a serviços envolvendo múltiplos participantes. Segundo a IBM, este problema está fora do escopo da linguagem.

4 A Linguagem XRL

A linguagem XRL (*XML Routing Language*) surgiu da necessidade da troca de dados entre empresas de comércio eletrônico. "O desenvolvimento de linguagens mais homogêneas para várias atividades de comércio eletrônico é uma maneira de facilitar a produtividade e aumentar a interoperabilidade" [9]. O objetivo é atender às atividades comerciais colaborativas que ocorrem via Internet.

Sendo a Internet um meio totalmente distribuído, XRL foi criada de maneira também distribuída, permitindo operações assíncronas entre empresas. Apesar de XRL possuir foco sobre transações de comércio eletrônico, ela pode ser utilizada na descrição de workflows de empresas de quaisquer ramos de negócio.

A principal característica de XRL é que ela fornece um mecanismo para descrever processos no nível de instância, de modo que um esquema XRL descreve a ordenação parcial das tarefas para uma instância de workflow específica. Além disso, ela é totalmente baseada em XML [11]. São diversas as vantagens de se utilizar a linguagem XML:

- XML é uma linguagem padrão, interoperável e que facilita a comunicação entre empresas/programas distintos
- XML é uma linguagem que está sendo cada vez mais difundida pela Web e utilizada na troca de dados
- a edição de documentos XML é bastante simples e pode ser feita a partir de qualquer editor de texto, sem que sejam necessárias ferramentas específicas para XML
- ela torna possível que um mesmo workflow seja apresentado para execução de diversas formas, através de templates distintos, sem que sejam necessárias alterações no documento XML fonte
- a elaboração de uma DTD torna o documento XML mais confiável, uma vez que ele pode ser validado.

A subseção seguinte apresenta a arquitetura de XRL/Flower, o sistema de gerência de workflow que utiliza a linguagem XRL para a especificação de workflows.

4.1 Arquitetura do XRL/Flower

A Figura 5 apresenta a arquitetura da máquina de workflow XRL/Flower.

De acordo com essa arquitetura, um esquema em XRL é enviado por e-mail para um nó da rede, aquele que simboliza a empresa com a qual a comunicação deve ser realizada. Neste nó receptor, o esquema é analisado sintaticamente e armazenado como uma estrutura de dados XML. O núcleo da máquina de workflow realiza uma leitura deste esquema e cria a representação de rede de Petri correspondente². Esta representação é então armazenada na forma de tabelas relacionais. A máquina do workflow determina qual a próxima tarefa a ser executada e a apresenta ao usuário (empresa com a qual a transação de comércio eletrônico está sendo efetuada) por meio de uma interface. O usuário, de posse da tarefa a ser realizada, notifica a máquina de workflow, que se encarrega de encontrar a tarefa seguinte. Quando todo o workflow foi executado, a máquina escreve novamente um arquivo XRL, no caso de integração forte, e o envia para o próximo nó do workflow.

Deste modo, as tarefas são executadas pelas empresas em colaboração via browser. Cada empresa a executar uma tarefa a recebe por e-mail e, através da URL recebida, pode executá-la. A submissão da tarefa realizada é recebida pela empresa que a invocou, a qual pode transformar a resposta da tarefa

²Em XRL, redes de Petri são utilizadas para a caracterização semântica do workflow.

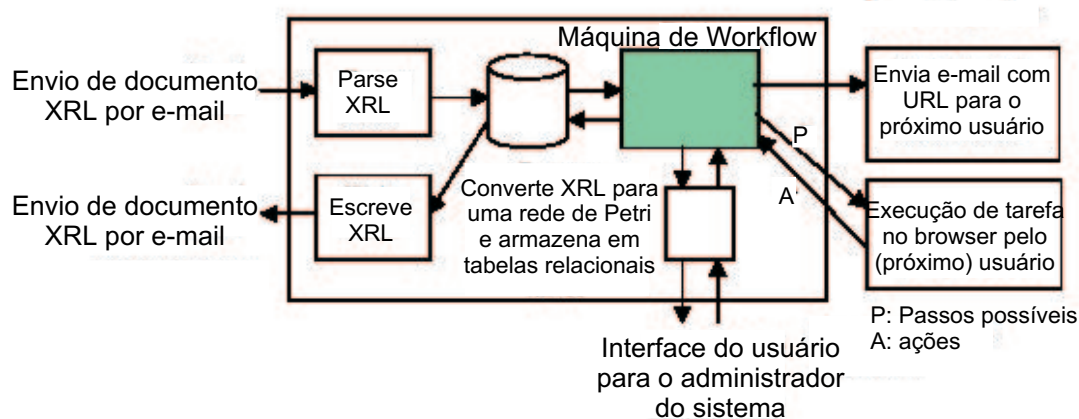


Figura 5: Arquitetura de XRL/Flower, a máquina de workflow que utiliza a linguagem XRL [9].

executada na representação de XRL. Junto à tarefa a ser executada, também podem ser enviados arquivos que serão necessários para a realização da mesma.

De acordo com o funcionamento da arquitetura descrita, são possíveis dois modos de operação distintos: o modo de forte integração e o modo de integração fraca. No primeiro modo, a empresa que inicia o processo de execução de uma instância de um workflow passa para a empresa com qual deve cooperar o controle da instância, quando esta precisa executar uma tarefa. Neste caso, o documento XML e os dados associados à execução das tarefas são enviados de um nó para o outro, de modo que o nó receptor passa a ter controle completo sobre a execução. Ao final da execução das tarefas que lhe foram delegadas, a empresa que havia recebido a tarefa e os dados envia de volta à outra empresa o arquivo XML e os dados representando o estado de sua computação.

No modo de integração fraca, a empresa que inicia o processo permanece durante todo o tempo com o controle da execução, repassando para a outra empresa apenas uma maneira desta realizar suas tarefas, via URL, por exemplo. Neste caso, não há transferência do arquivo XML que descreve o processo de interação. Através de um *submit* em um formulário da Web, por exemplo, a segunda empresa realiza sua tarefa e torna o resultado disponível para a empresa que controla a execução do workflow.

A Figura 6 é uma representação simples destes dois modos de integração existentes em XRL/Flower.

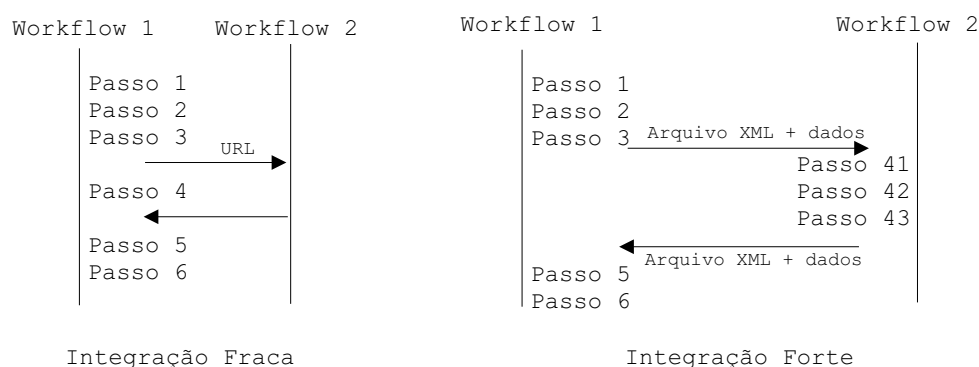


Figura 6: Modos de integração em XRL/Flower [9].

Quanto à representação semântica dos workflows, XRL utiliza redes de Petri [10]. Todo workflow em XRL é transformado internamente para a representação de redes de Petri. Porém, uma outra representação semântica pode ser utilizada.

No que se refere à aplicação de XRL, ela pode ser utilizada não apenas para a cooperação entre

empresas distintas, mas também para a cooperação entre membros de uma mesma empresa, que podem estar em pontos distintos de uma rede. De qualquer forma, o funcionamento continua sendo o mesmo, já que a transferência do arquivo XML e dos dados referentes à execução do workflow são independentes da equipe de execução, permitindo a distribuição, característica principal de sistemas de workflow atuais.

Segue a descrição da sintaxe de XRL.

4.2 Sintaxe de XRL

A sintaxe de XRL é baseada em XML. Os elementos descritos em XRL são os mais comuns em workflows para comércio eletrônico. No entanto, eles são válidos para qualquer tipo de workflow. O quadro a seguir apresenta a DTD (*Document Type Definition*) que especifica cada um dos elementos de XRL.

```
<!ENTITY % routing_element
    "task|sequence|any_sequence|choice|condition|
    parallel_sync|parallel_no_sync|parallel_part_sync|
    wait_all|wait_any|while_do|stop|terminate">

<!ELEMENT route (%routing_element;)>
<!ATTLIST route name ID #REQUIRED
    created_by CDATA #IMPLIED
    date CDATA #IMPLIED>

<!ELEMENT task (event*)>
<!ATTLIST task name ID #REQUIRED
    address CDATA #REQUIRED
    doc_read NMTOKENS #IMPLIED
    doc_update NMTOKENS #IMPLIED
    doc_create NMTOKENS #IMPLIED
    result CDATA #IMPLIED
    status (ready|running|enabled|
        disabled|aborted) #IMPLIED
    start_time NMTOKEN #IMPLIED
    end_time NMTOKEN #IMPLIED
    notify CDATA #IMPLIED
    role CDATA #IMPLIED>

<!ELEMENT event EMPTY>
<!ATTLIST event name ID #REQUIRED>

<!ELEMENT sequence ((%routing_element;|state)+)>
<!ELEMENT any_sequence ((%routing_element;)+)>
<!ELEMENT choice ((%routing_element;)+)>

<!ELEMENT condition (true|false)*>
<!ATTLIST condition condition CDATA #REQUIRED>
<!ELEMENT true (%routing_element;)>
<!ELEMENT false (%routing_element;)>

<!ELEMENT parallel_sync ((%routing_element;)+)>
<!ELEMENT parallel_no_sync ((%routing_element;)+)>
<!ELEMENT parallel_part_sync ((%routing_element;)+)>
<!ATTLIST parallel_part_sync number NMTOKEN #REQUIRED>

<!ELEMENT wait_all (event_ref|timeout)+>
<!ELEMENT wait_any (event_ref|timeout)+>

<!ELEMENT event_ref EMPTY>
<!ATTLIST event_ref name IDREF #REQUIRED>

<!ELEMENT timeout (%routing_element;)*>
<!ATTLIST timeout time CDATA #REQUIRED
    type (relative|s_relative|absolute) "absolute">
```

```

<!ELEMENT while_do (%routing_element;)>
<!ATTLIST while_do condition CDATA #REQUIRED>

<!ELEMENT stop EMPTY>
<!ELEMENT terminate EMPTY>
<!ELEMENT state (event+)>

```

O elemento raiz *route* foi definido como um conjunto de vários elementos especificados através da entidade XML *routing_element*, o elemento de roteamento. São estes os elementos: *task*, *sequence*, *any_sequence*, *choice*, *condition*, *parallel_sync*, *parallel_no_sync*, *parallel_part_sync*, *wait_all*, *wait_any*, *while_do*, *stop* e *terminate*.

Task

É a unidade atômica de processamento, qualquer tarefa a ser executada.

Sintaxe:

```

<!ELEMENT task (event*)>
<!ATTLIST task name ID #REQUIRED
               address CDATA #REQUIRED
               doc_read NMTOKENS #IMPLIED
               doc_update NMTOKENS #IMPLIED
               doc_create NMTOKENS #IMPLIED
               result CDATA #IMPLIED
               status (ready|running|enabled|disabled|aborted) #IMPLIED
               start_time NMTOKEN #IMPLIED
               end_time NMTOKEN #IMPLIED
               notify CDATA #IMPLIED
               role CDATA #IMPLIED>
<!ELEMENT event EMPTY>
<!ATTLIST event name ID #REQUIRED>

```

Cada tarefa em XRL pode descrever um evento. Este evento pode ser referenciado posteriormente à declaração da tarefa no workflow, indicando o término desta tarefa. Por exemplo, se um evento é utilizado dentro de um operador de espera, significa que o sistema deve esperar até que a tarefa relativa aquele evento tenha a sua execução finalizada.

Atributos:

- *role*: define qual o papel associado àquela tarefa, ou seja, qual o tipo de pessoa/recurso capaz de executá-la. Em XRL, a noção de papéis identifica características das entidades executoras, de modo que a cada tarefa podem estar associados um ou mais papéis distintos
- *address*: especifica a URL ou a página HTML, ASP, JSP, através da qual a tarefa será executada ou o e-mail de quem vai executá-la
- *doc_read*, *doc_update* e *doc_create*: representam documentos a serem lidos, atualizados ou criados. Estes elementos são especialmente úteis nos casos de troca de arquivos via Internet
- *result*: guarda uma string com o resultado da computação da tarefa
- *status*: indica o estado atual da tarefa, por exemplo, se ela está pronta para ser executada, em execução, abortada e assim por diante

- *start_time*: indica o momento em que a atividade foi iniciada
 - *end_time*: indica o momento em que a atividade foi finalizada. A partir deste atributo e do atributo anterior é possível conhecer em quanto tempo a tarefa foi executada
 - *notify*: indica o endereço eletrônico das pessoas/empresas que devem ser avisadas a respeito da execução da tarefa.
-

Sequence

Representa uma seqüência de tarefas, sejam elas atômicas ou complexas.

Sintaxe:

```
<!ELEMENT sequence ((%routing_element;|state)+)>
<!ELEMENT state (event+)>
```

O elemento *state* é necessário para representar o estado da execução em determinado ponto. A posição deste elemento no documento XRL que descreve o workflow determina o progresso da execução de uma parte da instância em execução. Se o processo é totalmente seqüencial, então apenas um elemento *state* aparece no documento que o representa. Caso contrário, podem haver vários destes elementos. Ele é de extrema importância para as instâncias quando o controle da execução das instâncias é repassado para outros usuários em tempo de execução, como ocorre no modo de integração forte.

Any_sequence

Uma seqüência de tarefas que podem ser executadas em qualquer ordem.

Sintaxe:

```
<!ELEMENT any_sequence ((%routing_element;)+)>
```

Choice

Representa a escolha de uma tarefa dentre um conjunto definido de tarefas a serem executadas.

Sintaxe:

```
<!ELEMENT choice ((%routing_element;)+)>
```

Condition

Testa uma condição e determina o próximo passo baseado no resultado booleano do teste.

Sintaxe:

```
<!ELEMENT condition (true|false)*>
<!ATTLIST condition condition CDATA #REQUIRED>
<!ELEMENT true (%routing_element;)>
<!ELEMENT false (%routing_element;)>
```

Parallel_sync

Um conjunto de elementos que são executados em paralelo e depois sincronizados.

Sintaxe:

```
<!ELEMENT parallel_sync ((%routing_element;)+)>
```

Parallel_no_sync

Um conjunto de elementos que são executados em paralelo e que não necessitam ser posteriormente sincronizados.

Sintaxe:

```
<!ELEMENT parallel_no_sync ((%routing_element;)+)>
```

Parallel_part_sync

Um conjunto de elementos que são executados em paralelo, com a ressalva de que a sincronização ocorre depois que um número pré-definido de tarefas foi executado.

Sintaxe:

```
<!ELEMENT parallel_part_sync ((%routing_element;)+)>  
<!ATTLIST parallel_part_sync number NMTOKEN #REQUIRED>
```

Wait_all

Quando se deseja esperar para que um grupo de eventos seja completado.

Sintaxe:

```
<!ELEMENT wait_all (event_ref|timeout)+>  
  
<!ELEMENT event_ref EMPTY>  
<!ATTLIST event_ref name IDREF #REQUIRED>  
  
<!ELEMENT timeout (%routing_element;)*>  
<!ATTLIST timeout time CDATA #REQUIRED  
                  type (relative|s_relative|absolute) "absolute">
```

Wait_any

Quando se deseja esperar para que um evento dentro de um conjunto se complete. Tanto este elemento quanto o elemento anterior podem conter especificações temporais, indicadas pelo subelemento *timeout*, ou relativas a algum evento em específico.

Sintaxe:

```
<!ELEMENT wait_any (event_ref|timeout)+>
```

Os elementos *event_ref* e *timeout* são os mesmos declarados no elemento anterior *wait_any*.

While_do

Elemento que possibilita a repetição de um ou mais eventos enquanto uma determinada condição é verdadeira.

Sintaxe:

```
<!ELEMENT while_do (%routing_element;)>
<!ATTLIST while_do condition CDATA #REQUIRED>
```

Stop

Termina a execução de uma determinada parte desta instância de workflow.

Sintaxe:

```
<!ELEMENT stop EMPTY>
```

Terminate

Finaliza a execução da instância do workflow.

Sintaxe:

```
<!ELEMENT terminate EMPTY>
```

4.3 Semântica de XRL

A arquitetura definida para XRL/Flower determina que um workflow é sempre mapeado para redes de Petri e posteriormente armazenado em banco de dados relacional. Em [10] são apontadas diversas razões para se escolher redes de Petri para a representação semântica de workflows, dentre elas o fato de redes de Petri serem uma ferramenta poderosa para a modelagem e a análise de processos de workflow. Além disso, a teoria de redes de Petri fornece técnicas de análise que podem ser utilizadas para verificar a corretude de um workflow.

Os autores de [9] argumentam que redes de Petri podem ser utilizadas para representar a lógica de negócio de uma maneira gráfica, bem parecida com linguagens de diagramação usadas para sistemas de gerência de workflow comerciais. Além disso, redes de Petri possuem técnicas poderosas de análise e resultados teóricos bastante promissores.

A Figura 7 apresenta os símbolos utilizados para o mapeamento de uma especificação de workflow em XRL para a representação gráfica de redes de Petri.

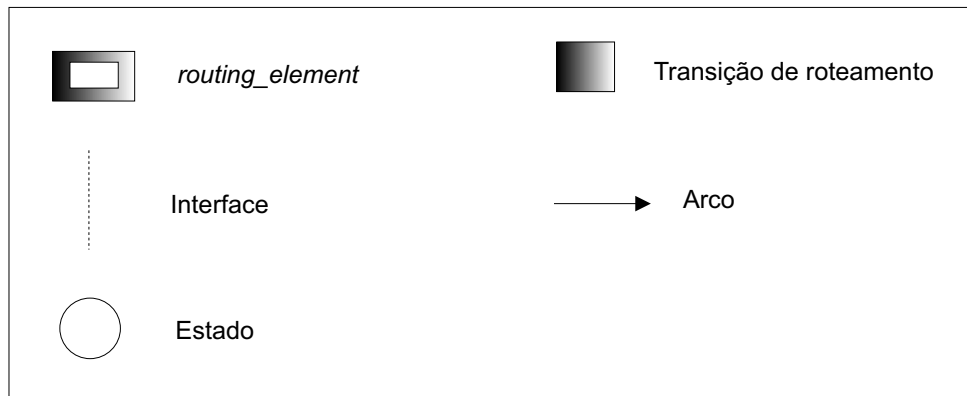


Figura 7: Símbolos usados no mapeamento de um workflow em XRL para redes de Petri [9].

As transições, representadas por quadrados, se referem às tarefas a serem executadas (representadas por retângulos). Os círculos se referem aos estados do workflow. Dizemos que, quando um token está em um estado i , fica habilitado para execução o elemento seguinte ao estado. À medida em que as tarefas vão sendo executadas, tokens vão se movendo pelos diversos estados, até que o estado final seja alcançado. Os arcos, por sua vez, representam o relacionamento entre as transições e os estados. Os exemplos que serão apresentados a seguir facilitam o entendimento de cada um destes símbolos.

Em [9] são apresentados todos os elementos da DTD de XRL mapeados para redes de Petri. Este mapeamento torna clara a função de cada um dos elementos e de como eles são interpretados internamente pela máquina de workflow.

Construtor *sequence*

O construtor XRL de seqüência é representado em redes de Petri conforme apresenta a Figura 8.

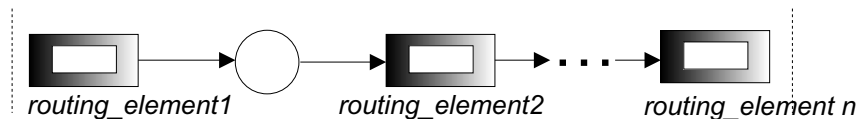


Figura 8: Construtor *sequence* de XRL [9].

Claramente, pela figura, esse construtor determina que o *routing_element2* apenas pode ser executado quando o *routing_element1* predecessor tiver sua execução completada. Desta forma, os elementos interligados são executadas em uma ordem pré-definida.

Construtor *any_sequence*

Para especificar que um conjunto de tarefas pode ser executado seqüencialmente, mas em qualquer ordem, pode ser utilizado o construtor *any_sequence*. A Figura 9 apresenta a modelagem deste construtor em redes de Petri. As tarefas apresentadas podem ser executadas em qualquer ordem, desde que as tarefas sejam executadas de forma mutuamente exclusiva. Esse controle é feito pelo *mutex*, que contém o token que identifica uma possível execução nos instantes em que não há tarefas sendo executadas. Ao final de todas as tarefas, o token é removido de *mutex*, indicando final da seqüência.

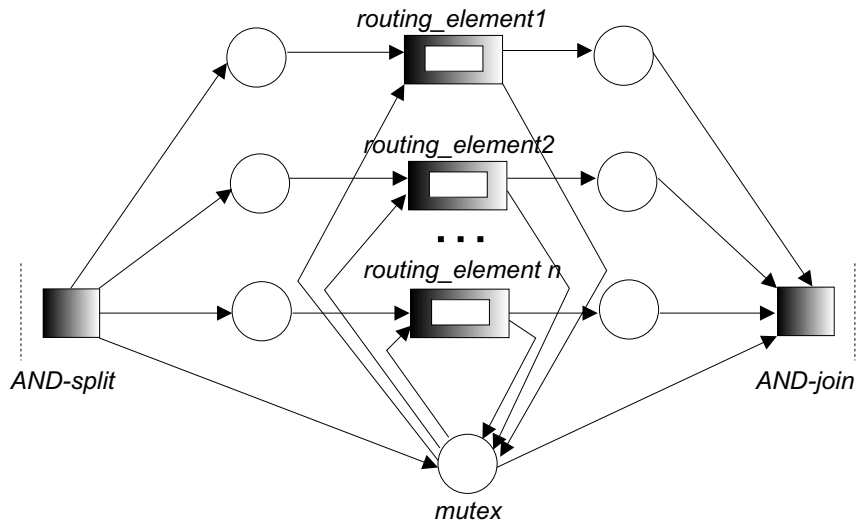


Figura 9: Construtor *any_sequence* de XRL [9].

Construtor *choice*

O construtor de escolha *choice* é mapeado para a representação de redes de Petri conforme mostra a Figura 10.

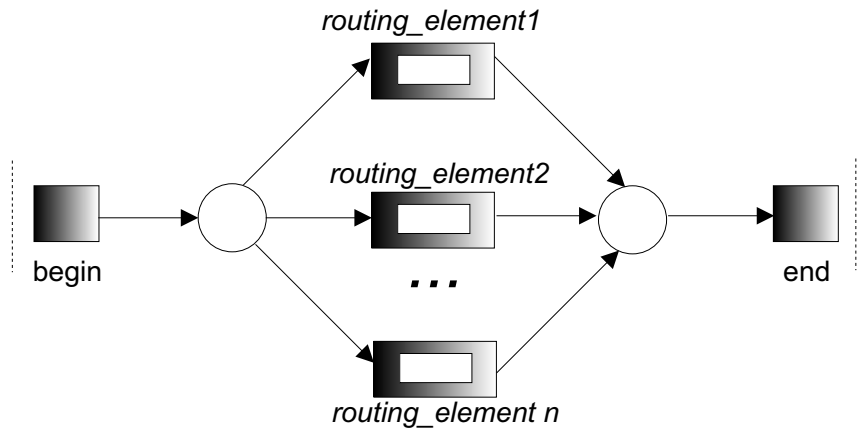


Figura 10: Construtor *choice* de XRL [9].

A semântica desse operador é bastante simples. Das n tarefas, apenas uma delas é executada.

Construtor *while_do*

Conforme apresentado na Figura 11, uma determinada tarefa é executada até que a condição de teste seja falsa. Quando o token alcança o estado *is_false* a execução do construtor *while_do* é finalizada.

4.4 Exemplo em XRL

Em [9] é apresentado um exemplo prático da utilização de XRL para o processamento de um pedido de livros. Considere que um consumidor efetua o pedido na Amazon.com de três livros, cada um de um editor diferente. A Amazon efetua três ordens de pedido, uma para cada um deles, solicitando

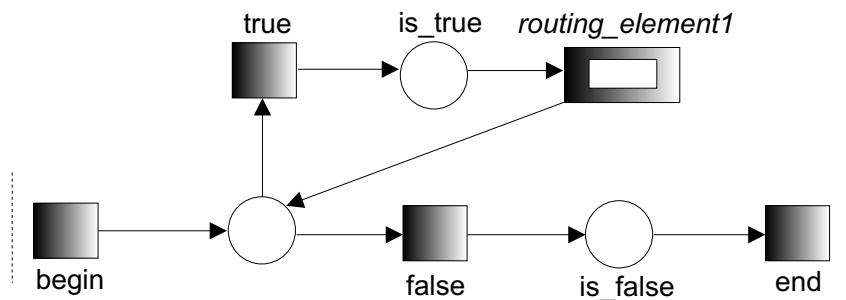


Figura 11: Construtor *while_do* de XRL [9].

o livro requerido no tempo especificado pelo consumidor. Se, por exemplo, o consumidor requisitou que se houvesse pelo menos um livro então ele gostaria de comprá-lo, a Amazon assim que receber a confirmação de um dos editores deve entrar em contato com a empresa responsável pela entrega dos livros. A Figura 12, por sua vez, apresenta a representação gráfica do documento para o exemplo descrito. O Apêndice 2 apresenta este exemplo escrito conforme a linguagem XRL.

Conforme demonstrado na figura, quando a Amazon recebe a requisição de seu cliente (consumidor), ela dispara em paralelo três instâncias de uma mesma tarefa, cada uma para um editor. Se a condição de que pelo menos dois deles confirmaram o pedido, indicando que possuem o livro e podem entregá-lo no tempo requisitado (valor do resultado "OK") for verdadeira, a Amazon passa a buscar por um serviço de entrega até que o consiga (interação representada pelo construtor *while_do*). Em seguida, a ordem é confirmada com cada um dos editores de forma assíncrona e paralela. A cada tarefa de confirmação está associado um evento, que é disparado assim que o editor correspondente envia a ordem. A semântica incluída ao final da representação gráfica mostra que esta ordem pode ser enviada em um prazo de dois dias, garantido pelo elemento *timeout*. Assim que todas as ordens foram enviadas, a execução desta instância do workflow é considerada terminada.

5 Proposta de Extensão da Linguagem XRL

Esta seção propõe uma extensão da linguagem XRL para acomodar as necessidades encontradas na especificação de planos de emergência para o projeto InfoPAE, o qual representa a grande motivação para este trabalho.

BPEL não foi escolhida por alguns motivos simples. Primeiro, BPEL é uma linguagem orientada a serviços Web, o que não é o intuito de todo processo de negócio. Em segundo lugar, a associação de recursos a partes de processos, tão comum em workflows, apenas pode ser implementada em BPEL a partir de um novo serviço. Em terceiro lugar, a semântica de BPEL ainda não está definida, ao contrário do que ocorre com a semântica de XRL.

O ponto que representa a maior diferença entre a nossa proposta de extensão e a linguagem XRL original é que a linguagem XRL é utilizada não apenas para definição de workflows, mas também para a representação de seu estado de computação durante execução. No entanto, acreditamos que a definição de um workflow e a representação de seu estado de computação devem ser definidos separadamente no sistema.

5.1 Sintaxe Proposta

Basicamente a nossa proposta de extensão da linguagem XRL envolve a inclusão de novos elementos e a alteração da sintaxe de alguns elementos. Os atributos ou elementos relativos à execução de um workflow

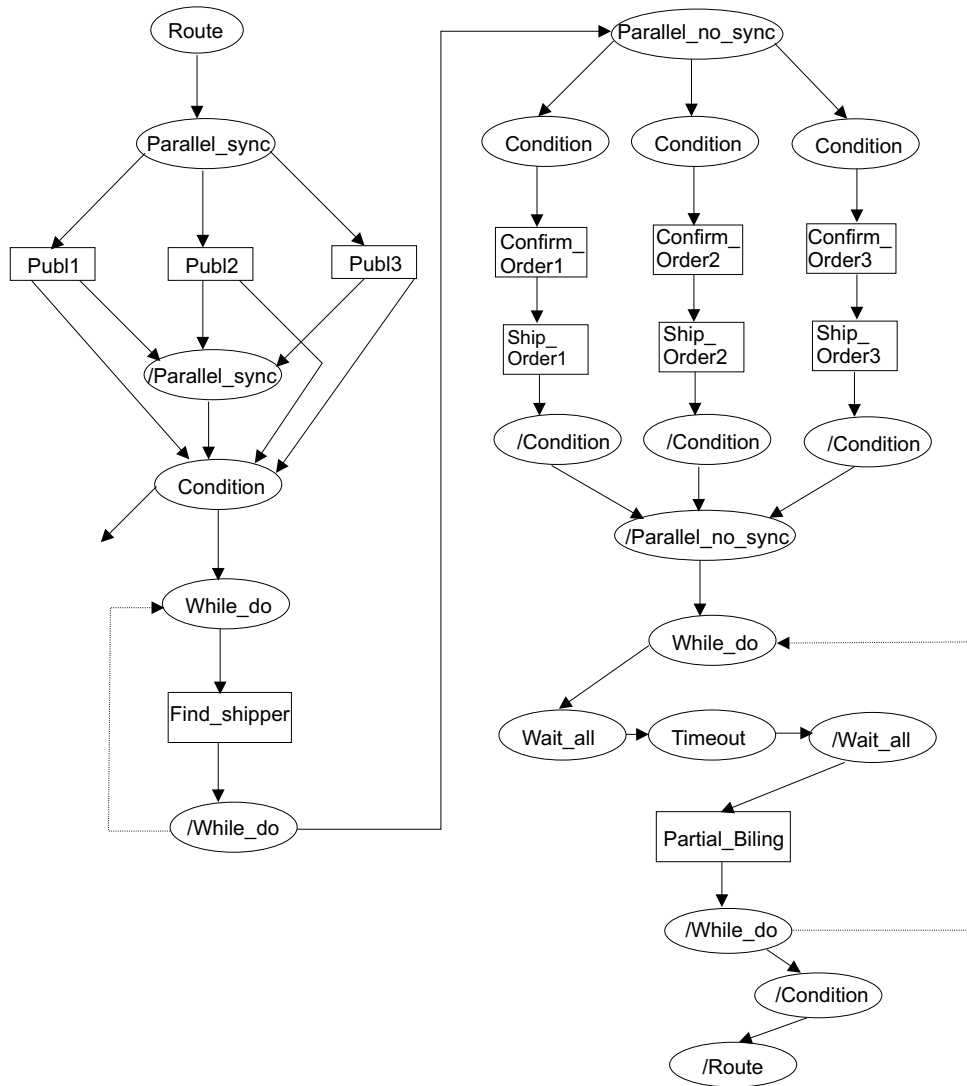


Figura 12: Representação gráfica do exemplo descrito em XRL [9].

(atributos *result*, *status*, *start_time* e *end_time*, e elementos *stop* e *terminate*) permanecem, mas como já sugerido, não são utilizados conforme a nossa visão de um sistema de workflow. A nossa proposta é que existam documentos restritos apenas à especificação de workflows e outros que guardem o estado de sua computação. No entanto, para o caso do workflow ser utilizado para o propósito de comércio eletrônico, estes atributos são bastante importantes.

Segue uma descrição das alterações realizadas sobre a DTD de XRL, segundo a nossa proposta.

O Apêndice 3 apresenta a DTD de XRL estendida com as modificações aqui propostas. O Apêndice 4, por sua vez, apresenta o exemplo descrito no Apêndice 2, modificado de acordo com as extensões de XRL propostas.

Observe que as alterações propostas não diminuem a qualidade semântica da linguagem e que os workflows que podiam antes ser descritos nesta linguagem continuam o sendo.

Entidade *term*

A entidade *term* foi acrescentada na linguagem para permitir a especificação de expressões booleanas,

sem que seja necessário um parser posterior sobre a string da condição. Esta entidade envolve todos os operadores booleanos.

O elemento *expression* foi criado no intuito de incluir uma ou mais ocorrências de qualquer expressão booleana. Portanto, este elemento é utilizado nos construtores *exec_condition*, *condition* e *while_do*.

Sintaxe:

```
<!ENTITY % term "and|or|not|lt|le|eq|ne|ge|gt">
<!ELEMENT expression ((%term;)+)>
<!ELEMENT and ((%term;), (%term;)+)>
<!ELEMENT or ((%term;), (%term;)+)>
<!ELEMENT not ((%term;) | event_conf)>

<!ELEMENT event_conf EMPTY>
<!ATTLIST event_conf name CDATA #REQUIRED>

<!ELEMENT lt EMPTY>
<!ATTLIST lt right CDATA #REQUIRED
left CDATA #REQUIRED>

<!ELEMENT le EMPTY>
<!ATTLIST le right CDATA #REQUIRED
left CDATA #REQUIRED>

<!ELEMENT eq EMPTY>
<!ATTLIST eq right CDATA #REQUIRED
left CDATA #REQUIRED>

<!ELEMENT ne EMPTY>
<!ATTLIST ne right CDATA #REQUIRED
left CDATA #REQUIRED>

<!ELEMENT ge EMPTY>
<!ATTLIST ge right CDATA #REQUIRED
left CDATA #REQUIRED>

<!ELEMENT gt EMPTY>
<!ATTLIST gt right CDATA #REQUIRED
left CDATA #REQUIRED>
```

O elemento *event_conf* representa a confirmação da ocorrência de um evento. Ele foi incluído por ser útil no caso de expressões booleanas que verificam se um evento ocorreu com sucesso.

Elemento *faultHandler*

Aproveitando de BPEL4WS a idéia do gerenciamento de falhas, incluímos na nossa proposta um construtor chamado *faultHandler*, o qual é o responsável pelo gerenciamento de falhas para as atividades atômicas do workflow (*task*, *call*, *send* e *receive*, as quais serão apresentadas a seguir). Caso ocorra uma falha em uma atividade para a qual está definido um gerenciador, então as atividades descritas por ele são executadas, a fim de que a execução do workflow possa continuar. Além disso, o gerenciador de falhas pode invocar o gerenciador de atividades compensatórias, para que este desfça o efeito de atividades já completadas, mas que não fazem sentido serem devido à falha ocorrida.

Por exemplo, um gerenciador de falhas pode ser definido para uma tarefa do tipo "Solicite entrega do produto comprado", quando o processo de compra não ocorreu com sucesso.

Sintaxe:

```
<!ELEMENT faultHandler(%routing_element;, compensationHandler?)>
```

Elemento *compensationHandler*

Este elemento, apenas invocado a partir do gerenciador de falhas, é o responsável por desfazer o efeito de atividades já completadas, mas que no entanto não fazem mais sentido devido à ocorrência de determinada falha.

Sintaxe:

```
<!ELEMENT compensationHandler(%routing_element)>
```

Elemento *route*

O elemento *route*, original de XRL, teve a sua estrutura alterada. Ele possui agora até 4 elementos:

- *variables*: define todas as variáveis utilizadas no workflow e seus possíveis valores. Estas variáveis (parâmetros) são como variáveis de programa e possuem o mesmo escopo. Cada variável é válida dentro do workflow (e seus subworkflows) no qual foi declarada
- *exec_condition*: este elemento, conforme anteriormente explicado, se refere à condição, que pode ser qualquer expressão booleana, que identifica as situações nas quais o (sub)workflow torna-se pronto para ser executado. Cada um dos parâmetros referenciados nesta condição de disparo devem estar no escopo do workflow correspondente, devidamente declarados
- *subworkflows*: são todos os subworkflow do workflow maior
- *body*: neste elemento estão todos os passos da execução do workflow. Nele está descrito o processo propriamente dito.

Sintaxe:

```
<!ELEMENT route (variables?,exec_condition?,subworkflows?,body)>
<!--ATTLIST route name CDATA #REQUIRED
created_by CDATA #IMPLIED
date CDATA #IMPLIED
id ID #REQUIRED
independent (yes | no) "no"
persistent (yes | no) "no"
cancelable (yes | no) "no"-->

<!--ELEMENT variables (parameter)+>
<!--ELEMENT parameter (value)*>
<!--ATTLIST parameter name_parameter CDATA #REQUIRED >
<!--ELEMENT value EMPTY>
<!--ATTLIST value bd (yes|no) "no"
value_parameter CDATA #REQUIRED-->

<!--ELEMENT exec_condition (expression)>

<!--ELEMENT subworkflows (route)+>

<!--ELEMENT body (%routing_element;)+>
```

Atributos inseridos ou modificados no elemento *route*:

- *name*: já que, segundo a extensão de XRL aqui proposta um workflow pode ter vários subworkflows aninhados, este atributo deixa de ser identificador, e passa a ser apenas um nome do workflow. Deste modo, workflows distintos passam a poder ter nomes iguais. Isso é essencialmente importante quando se deseja nomear igualmente dois subworkflows de workflows distintos. Todo workflow deve ser nomeado

- *id*: atributo identificador do elemento *route*. Este valor é único dentro do elemento XML no que se refere a workflow
- *independent*: indica se o workflow pode ser executado em um processo separado. Se este valor é *yes*, então o workflow também pode ser *persistent* e *cancelable*. Valor default é *no*
- *persistent*: um workflow é persistente se ele pode ser chamado para execução diversas vezes, de maneira independente. O valor default é *no*
- *cancelable*: indica se o processo que executa o workflow pode ser cancelado. Por default, um processo de workflow não é cancelável.

O elemento *variables* compreende uma lista de parâmetros. Cada parâmetro possui um lista de valores associados, de modo que cada valor pode ser indicado via string ou através de uma consulta ao banco de dados. Para que um valor seja reconhecido como uma consulta ao banco, o atributo *bd* de *value* deve ter valor *yes*.

Elemento *task*

Sintaxe:

```
<!ELEMENT task (event*, faultHandler?)>
<!ATTLIST task
    id ID #REQUIRED
    name CDATA #REQUIRED
    persistent (yes | no) "no"
    bypassed (yes | no) "no"
    postponed (yes | no) "no"
    deadline CDATA #IMPLIED
    address CDATA #IMPLIED
    doc_read NMTOKENS #IMPLIED
    doc_update NMTOKENS #IMPLIED
    doc_create NMTOKENS #IMPLIED
    result CDATA #IMPLIED
    status (ready|running|enabled|
            disabled|aborted) #IMPLIED
    start_time NMTOKEN #IMPLIED
    end_time NMTOKEN #IMPLIED
    notify CDATA #IMPLIED
    role CDATA #IMPLIED>

<!ELEMENT event EMPTY>
<!ATTLIST event name ID #REQUIRED>
```

Segue uma descrição sucinta das alterações realizadas:

- gerenciamento de falhas: para cada tarefa, podem ser definidas atividades a serem executadas no caso de ocorrência de falha durante sua execução
- atributo *name*: já que uma mesma tarefa pode ser invocada por diversos subworkflows distintos, declarados em um mesmo arquivo XML, este atributo deixa de ser identificador da tarefa
- atributo *persistent*: este atributo indica, assim como ocorre para o (sub)workflows, que a tarefa pode ser executada a qualquer instante, de forma independente dos demais processos em execução
- atributo *bypassed*: este atributo foi incluído para identificar se uma tarefa pode deixar de ser realizada, sem atrapalhar a continuação da execução do (sub)workflow do qual faz parte

- atributo *postponed*: este atributo indica que uma tarefa pode ser adiada. Por exemplo, ele é extremamente útil para o caso de tarefas do tipo "Ligar para o gerente para avisar da ocorrência". Se esta tarefa for declarada como *postponed* e o usuário não conseguir de imediato realizá-la, ele poderá continuar a execução e mais tarde tentar completá-la
- atributo *deadline*: indica o tempo máximo que o usuário deve gastar para completar a tarefa. A máquina de execução pode, por exemplo, avisá-lo de que o tempo está acabando. Caso o usuário não consiga completá-la neste tempo, o sistema de workflow deve tomar alguma providência para que a execução do processo não seja bloqueada. Para o controle do *deadline*, a máquina de execução pode utilizar o conceito de regras ativas, como em WIDE, ou o conceito de *triggers* de banco de dados. Com triggers o controle do que deve ocorrer caso este tempo acabe sem que a execução da tarefa seja completada é determinado pelo banco e, por isso, a carga de trabalho da máquina de execução é menor.

Os demais atributos de *task* foram mantidos, apenas no intuito de garantir que um workflow nesta linguagem continue podendo ser utilizado durante uma transação de comércio eletrônico. No entanto, alguns deles dizem respeito ao estado da execução e, segundo a nossa abordagem, devem ser registrados em um log de execução e não juntamente com a especificação do workflow em execução.

No entanto, seguindo a lógica de WIDE, para que a especificação do workflow seja independente das informações de cada empresa, é melhor que a empresa mantenha um repositório de dados com as informações associadas a cada tarefa, de modo que uma empresa X pode utilizar um workflow Y e distribuí-lo para empresas interessadas, sem que estas necessitem efetuar qualquer tipo de alteração. Estas empresas apenas fariam como a primeira, criando em seu banco uma base de dados com as informações pertinentes. A modularização de WIDE é bastante interessante, de modo que as tarefas são especificadas separadamente das informações a elas associadas (neste caso, não seriam utilizados os atributos que especificam recursos associados às tarefas). Uma busca no módulo de Informação pode encontrar os documentos associados a cada tarefa de um workflow. Além disso, essa modularização torna possível que um documento também seja associado a um grupo de tarefas ao invés de a uma tarefa particular.

Elemento *ask*

O elemento *ask* foi inserido para representar uma pergunta que pode ser feita ao usuário executor durante a execução de uma instância do workflow. Neste caso, a máquina de execução deve apresentar ao usuário o parâmetro para o qual espera resposta. Se o atributo *ask_value* não tiver sido identificado, isso significa que a máquina deve apresentar ao executor a lista de valores anteriormente declarados para que ele escolha um dentre eles.

Este elemento *ask* representa a condição de um (sub)workflow ser tratado com uma rotina, que pode receber valores de entrada como parâmetro. Neste caso, os valores perguntados ao usuário são como valores de entrada para o (sub)workflow.

Sintaxe:

```
<!ELEMENT ask EMPTY>
<!ATTLIST ask
    id          ID          #REQUIRED
    description CDATA      #IMPLIED
    ask_parameter CDATA      #REQUIRED
    ask_value   CDATA      #IMPLIED>
```


Elemento *parallel_part_sync*

Este elemento, original da linguagem XRL, foi alterado para permitir não só que esse conjunto de elementos de execução seja considerado como terminado se apenas alguns deles tenham sido executados, mas que seja considerado terminado se determinados elementos, especificadas através do atributo *refs* tenham sido executados. Para isso, foi necessária a inclusão do atributo identificador *id* em cada um dos elementos que fazem parte de *routing_element*.

Sintaxe:

```
<!ELEMENT parallel_part_sync ((%routing_element;)+)>
<!--ATTLIST parallel_part_sync  id ID #REQUIRED
                                number NMTOKEN #REQUIRED
                                refs IDREFS #IMPLIED-->
```

Elemento *while_do*

Para que a máquina de execução não precise fazer uma análise sobre a string de condição do elemento *while_do*, o atributo *condition* foi substituído pelo elemento *expression*, anteriormente apresentado. O subelemento *do* de *while_do* contém o conjunto de ações a serem executadas enquanto a condição em *expression* for verdadeira.

Sintaxe:

```
<!ELEMENT while_do (expression, do)>
<!--ATTLIST while_do id ID #REQUIRED-->
<!--ELEMENT do (%routing_element;)-->
```

Elemento *call*

Este elemento representa a chamada a um (sub)workflow. Da mesma forma que para uma tarefa, para este elemento podem ser definidas atividades a serem executadas quando falhas ocorrerem.

Sintaxe:

```
<!ELEMENT call (faultHandler?)>
<!--ATTLIST call id ID #REQUIRED
                sub_workflow CDATA #REQUIRED-->
```

Elemento *find_executables*

Este elemento é extremamente útil pelo fato de indicar à máquina de execução que ela deve procurar pelos (sub)workflows que podem ser disparados para execução. Neste momento, além dos (sub)workflows independentes, também podem ser disparados quaisquer (sub)workflows que tenham sua condição de disparo satisfeita. No entanto, cabe ao usuário executor decidir qual ou quais deles ele deseja executar naquele momento.

Este elemento representa a flexibilidade do workflow proposto, no sentido de que a máquina vai encontrando, em tempo de execução, quais (sub)workflows podem ser executados naquele instante.

Sintaxe:

```
<!ELEMENT find_executables EMPTY>
<!ATTLIST find_executables id ID #REQUIRED>
```

Elemento *send*

O elemento *send* foi incluído na DTD da especificação da linguagem para implementar a comunicação entre os processos, já que workflows muitas vezes podem ser executados como processos separados. A função deste elemento é representar o envio de uma mensagem, definida no campo *message*, através de canal *channel*. Para cada par de comandos *send-receive*, assumimos que exista uma fila de mensagens.

Um gerenciador de falhas para esta atividade também pode ser definido.

Sintaxe:

```
<!ELEMENT send (faultHandler?)>
<!ATTLIST send id ID #REQUIRED
               channel CDATA #REQUIRED
               message CDATA #REQUIRED>
```

Elemento *receive*

Também destinado à implementação da comunicação entre processos, o elemento *receive* trata do recebimento de uma determinada mensagem, através de um canal de comunicação específico. Pode também estar associado a um gerenciador de falhas.

Sintaxe:

```
<!ELEMENT receive (faultHandler?)>
<!ATTLIST receive id ID #REQUIRED
                  channel CDATA #REQUIRED
                  message CDATA #REQUIRED>
```

5.2 Semântica

Tendo definido a sintaxe da linguagem proposta, uma extensão de XRL, torna-se necessário definir a semântica dos novos construtores.

Construtor *call*

Conforme a nossa visão distribuída de processos/workflows, um subworkflow pode ser executado em um processo separado e independente. Desta forma, quando um subworkflow é invocado através do elemento *call*, ele pode passar a ser executado em um processo separado, desvinculando dele a execução do processo que o invocou. Desta forma, no início da execução do elemento *call*, um token habilita a execução do processo chamado e finaliza a execução do elemento. No entanto, se o subworkflow invocado não é independente, então a execução de *call* apenas é finalizada quando o subworkflow tiver sido executado.

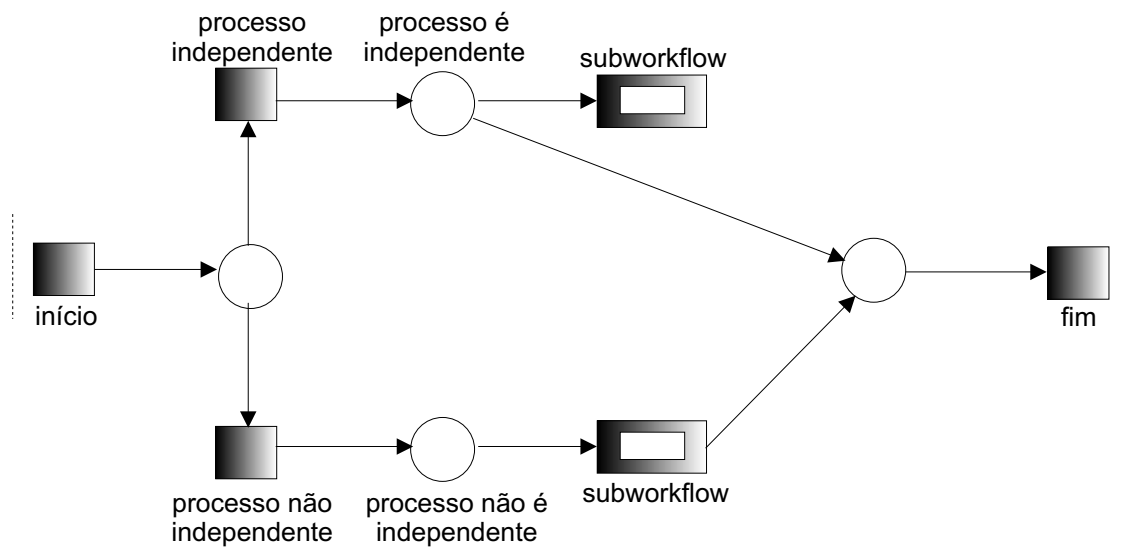


Figura 13: Construtor *call*.

Construtor *find_executables*

De acordo com a sintaxe apresentada, a máquina de execução de workflows tem por tarefa, quando de posse de um elemento *find_executables*, encontrar quais os novos workflows que se tornaram prontos para serem executados. Cada subworkflow "pronto" para ser executado pode ser disparado em um processo separado do processo que o invocou, quando este workflow é independente. Caso os workflows não sejam independentes, então o elemento *find_executables* apenas será finalizado quando todos os workflows tiverem sido executados, conforme mostra a Figura 14.

Construtor *send*

O construtor *send* pode ser representado segundo a terminologia de redes de Petri conforme a Figura 15. Quando *send* é executado, ele coloca um token no estado que representa a fila de mensagens e um token em outro estado, representando a finalização da execução do elemento.

Construtor *receive*

Segundo a semântica apresentada, este construtor apenas é considerado executado quando pelo menos uma mensagem é recebida na fila de mensagens. Enquanto a mensagem não é recebida, este construtor bloqueia a execução do processo que o contém.

6 Conclusões

Tendo em vista que diversas empresas atualmente se utilizam de workflows para a definição de seus processos de negócio, a criação de uma linguagem legível e bastante flexível para a especificação de workflows torna-se necessária.

A experiência com o projeto InfoPAE nos trouxe conhecimentos mais sólidos e práticos a respeito do problema de especificação de workflows. Além disso, o estudo de XRL nos mostrou que esta linguagem poderia atender não apenas a workflows de comércio eletrônico, mas também a qualquer outro tipo de

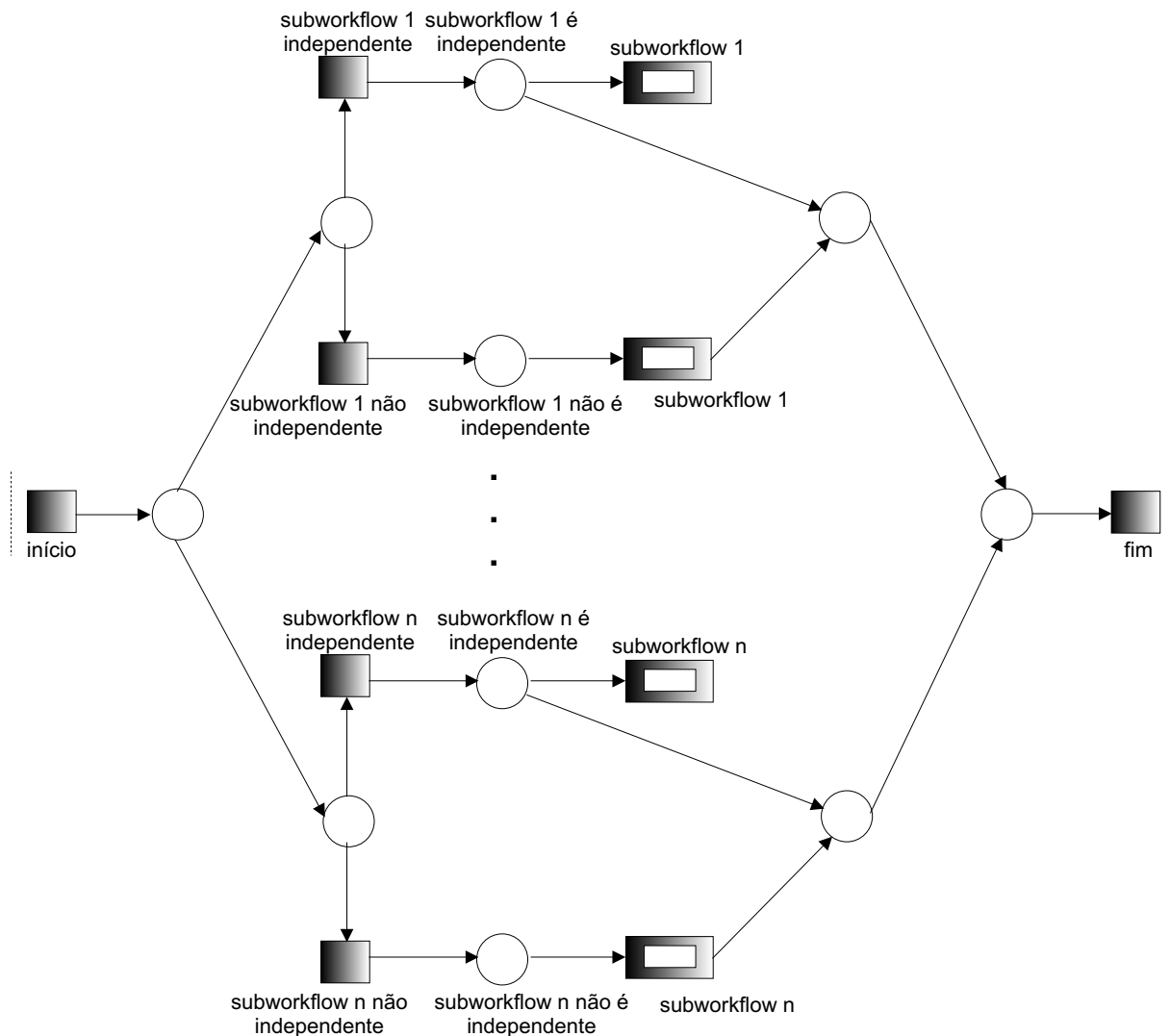


Figura 14: Construtor *find_executables*.

workflow. Para isso, estendemos esta linguagem, incorporando novos elementos que foram necessários quando da construção de planos de ação de emergência para o projeto InfoPAE.

A grande vantagem da linguagem estendida que propusemos, além do fato de estar escrita em XML, é que ela torna a execução de um workflow flexível. A partir desta linguagem, podemos decompor um workflow maior em subworkflows, alguns dos quais podem ser executados em processos separados, agilizando o processamento do workflow como um todo.

Uma outra extensão da linguagem XRL poderia ser fazer com que o estado de um processo carregasse o histórico de execução. Este histórico poderia conter, por exemplo, as tarefas executadas, os estados das tarefas pendentes, dentre outras informações. Entretanto, a inserção no histórico de informações de tarefas ainda não realizadas é bastante difícil, ou talvez impossível, dado que o workflow se comporta como uma árvore de decisão. Desta forma, é impossível prever, em determinado ponto do sistema, qual ramo da árvore será seguido durante a execução do workflow.

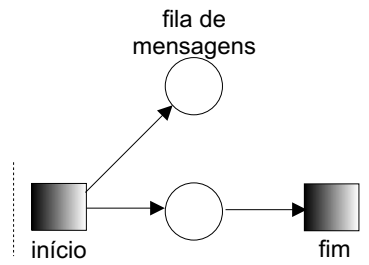


Figura 15: Construtor *send*.

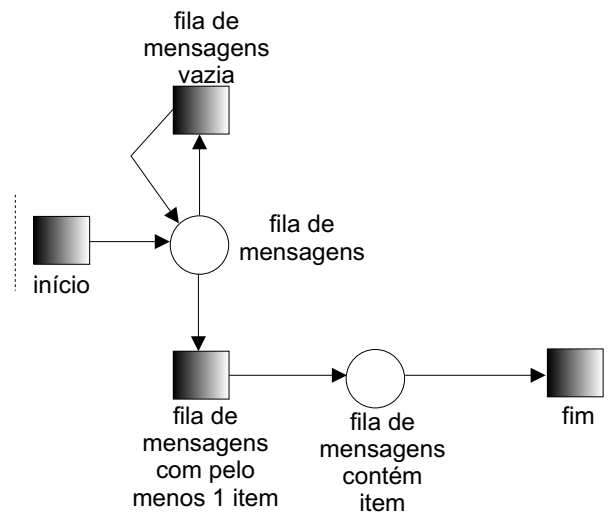


Figura 16: Construtor *receive*.

Agradecimentos

Agradecemos ao Flávio Torres e ao Ângelo Francisco dos Santos, da Petrobras, e aos outros membros do projeto InfoPAE pelas suas contribuições para as idéias expressas nesta monografia.

Referências

- [1] M. T. M. Carvalho, J. Freire, and M. A. Casanova. The Architecture of an Emergency Plan Deployment System. In *III Workshop Brasileiro de GeoInformática*, pages 19-26, Instituto Militar de Engenharia (IME) - Rio de Janeiro, RJ, Outubro 2001.
- [2] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Sánchez. WIDE Workflow Model and Architecture, April 1996.
- [3] S. Ceri, P. W. P. J. Grefen, and G. Sanchez. WIDE: A Distributed Architecture for Workflow Management. In *Seventh International Workshop on Research Issues in Data Engineering (RIDE'97)*, Birmingham, UK, 1997.
- [4] F. Curbera, Y. Golland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.0. <http://www.ibm.com/developerworks/library/ws-bpel/>, July 2002.
- [5] G. S. Gutiérrez. The WIDE Workflow Model and Language. Deliverable D5.1.2 4080-2, ESPRIT Project 20280, October 1997.
- [6] G. S. Gutiérrez. The WIDE Workflow Model and Language. Technical Report 4111-2, ESPRIT Project 20280, May 1999.
- [7] D. Hollingsworth. The Workflow Management Coalition Specification - The Workflow Reference Model. Technical Report TC00-1003, Workflow Management Coalition, Hampshire, UK, January 1995.
- [8] F. Leymann and D. Roller. Business Processes in a Web Services World - A Quick Overview of BPEL4WS. developerWorks, August 2002.
- [9] W. M. P. van der Aalst and A. Kumar. XML Based Schema Definition for Support of Inter-organizational Workflow. In *21st International Conference on Application and Theory of Petri Nets*, Aarhus, Denmark, June 2000.
- [10] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers*, 8(1):21-66, 1998.
- [11] W3C. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation. <http://www.w3.org/TR/2000/REC-xml-20001006>, October 2000.
- [12] S. Weerawarana and F. P. Curbera. Business Processes: Understanding BPEL4WS, Part 1 - Concepts in Business Processes. developerWorks, August 2002.

Apêndice 1 - Processo de Negócio em BPEL4WS

```
<process name="ticketOrder">
  <partners>
    <partner name="customer"
      serviceLinkType="agentLink"
      myRole="agentService"/>
    <partner name="airline"
      serviceLinkType="buyerLink"
      myRole="ticketRequester"
      partnerRole="ticketService"/>
  </partners>
  <containers>
    <container name="itinerary" messageType="itineraryMessage"/>
    <container name="tickets" messageType="ticketsMessage"/>
  </containers>
  <flow>
    <links>
      <link name="order-to-airline"/>
      <link name="airline-to-agent"/>
    </links>
    <receive partner="customer"
      portType="itineraryPT"
      operation="sendItinerary"
      container="itinerary">
      <source linkName="order-to-airline"/>
    </receive>
    <invoke partner="airline"
      portType="ticketOrderPT"
      operation="requestTickets"
      inputContainer="itinerary">
      <target linkName="order-to-airline"/>
      <source linkName="airline-to-agent"/>
    </invoke>
    <receive partner="airline"
      portType="itineraryPT"
      operation="sendTickets"
      container="tickets">
      <target linkName="airline-to-agent"/>
    </receive>
  </flow>
</process>
```

Apêndice 2 - Workflow de negócios segundo XRL

```
<?xml version="1.0"?>
<!DOCTYPE route SYSTEM "xrl.dtd">

<route name="example" created_by="Wil" date="120999">
  <sequence>
    <parallel_sync>
      <task name="publ1" address="www.taskserver.com/publ1"/>
      <task name="publ2" address="www.taskserver.com/publ2"/>
      <task name="publ3" address="www.taskserver.com/publ3"/>
    </parallel_sync>
    <condition condition="(publ1.result='OK' and publ2.result='OK')
      or (publ1.result='OK' and publ3.result='OK')
      or (publ2.result='OK' and publ3.result='OK')">
      <true>
        <sequence>
          <while_do condition="find_shipper.result='OK'">
            <task name="find_shipper"
              address="www.taskserver.com/find_shipper"/>
          </while_do>
          <parallel_no_sync>
            <condition condition="publ1.result='OK'">
              <true>
                <sequence>
                  <task name="confirm_order1"
                    address="www.taskserver.com/confirm1"/>
                  <task name="ship_order1"
                    address="www.taskserver.com/order1">
                    <event name="e_order1"/>
                  </task>
                </sequence>
              </true>
            </condition>
            <condition condition="publ2.result='OK'">
              <true>
                <sequence>
                  <task name="confirm_order2"
                    address="www.taskserver.com/confirm2"/>
                  <task name="ship_order2"
                    address="www.taskserver.com/order2">
                    <event name="e_order2"/>
                  </task>
                </sequence>
              </true>
            </condition>
            <condition condition="publ3.result='OK'">
              <true>
                <sequence>
                  <task name="confirm_order3"
                    address="www.taskserver.com/confirm3"/>
                  <task name="ship_order3"
                    address="www.taskserver.com/order3">
                    <event name="e_order3"/>
                  </task>
                </sequence>
              </true>
            </condition>
          </parallel_no_sync>
          <while_do condition="(publ1.result = 'OK'
            AND not_done(e_order1))
            or (publ2.result = 'OK' AND not_done(e_order2))
            or (publ3.result = 'OK' AND not_done(e_order3))">
            <sequence>
              <wait_all>
                <timeout time="2 days" type="relative"/>
              </wait_all>
              <task name="billing_partial"
                address="www.taskserver.com/billing_partial"/>
            </sequence>
          </while_do>
        </true>
      </condition>
    </parallel_sync>
  </sequence>
</route>
```



```
        </sequence>
      </while_do>
    </sequence>
  </true>
</condition>
</sequence>
</route>
```

Apêndice 3 - Linguagem XRL Estendida

```
<!ENTITY % routing_element "task|sequence|any_sequence|choice|ask|
                             condition|parallel_sync|
                             parallel_no_sync|parallel_part_sync|
                             wait_all|wait_any|while_do|call|
                             find_executables|send|receive">

<!ENTITY % term "and|or|not|lt|le|eq|ne|ge|gt">

<!ELEMENT expression ((%term;)+)>

<!ELEMENT and ((%term;), (%term;)+)>
<!ELEMENT or ((%term;), (%term;)+)>
<!ELEMENT not ((%term;) | event_conf)>

<!ELEMENT event_conf EMPTY>
<!ATTLIST event_conf name CDATA #REQUIRED>

<!ELEMENT lt EMPTY>
<!ATTLIST lt right CDATA #REQUIRED
           left CDATA #REQUIRED>

<!ELEMENT le EMPTY>
<!ATTLIST le right CDATA #REQUIRED
           left CDATA #REQUIRED>

<!ELEMENT eq EMPTY>
<!ATTLIST eq right CDATA #REQUIRED
           left CDATA #REQUIRED>

<!ELEMENT ne EMPTY>
<!ATTLIST ne right CDATA #REQUIRED
           left CDATA #REQUIRED>

<!ELEMENT ge EMPTY>
<!ATTLIST ge right CDATA #REQUIRED
           left CDATA #REQUIRED>

<!ELEMENT gt EMPTY>
<!ATTLIST gt right CDATA #REQUIRED
           left CDATA #REQUIRED>

<!ELEMENT faultHandler (%routing_element;, compensationHandler?)>

<!ELEMENT compensationHandler (%routing_element;)>

<!ELEMENT route (variables?,exec_condition?,subworkflows?,body)>
<!ATTLIST route name CDATA #REQUIRED
               created_by CDATA #IMPLIED
               date CDATA #IMPLIED
               id ID #REQUIRED
               independent (yes | no) "no"
               persistent (yes | no) "no"
               cancelable (yes | no) "no">

<!ELEMENT variables (parameter)+>
<!ELEMENT parameter (value)*>
<!ATTLIST parameter name_parameter CDATA #REQUIRED>
<!ELEMENT value EMPTY>
<!ATTLIST value bd (yes | no) "no"
               value_parameter CDATA #REQUIRED>

<!ELEMENT exec_condition (expression)>

<!ELEMENT subworkflows (route)+>

<!ELEMENT body (%routing_element;)+>
```

```

<!ELEMENT task (event*,faultHandler?)>
<!--ATTLIST task
    id ID #REQUIRED
    name CDATA #REQUIRED
    persistent (yes | no) "no"
    bypassed (yes | no) "no"
    postponed (yes | no) "no"
    deadline CDATA #IMPLIED
    address CDATA #IMPLIED
    doc_read NMTOKENS #IMPLIED
    doc_update NMTOKENS #IMPLIED
    doc_create NMTOKENS #IMPLIED
    result CDATA #IMPLIED
    status (ready|running|enabled|
            disabled|aborted) #IMPLIED
    start_time NMTOKEN #IMPLIED
    end_time NMTOKEN #IMPLIED
    notify CDATA #IMPLIED
    role CDATA #IMPLIED-->

<!ELEMENT event EMPTY>
<!--ATTLIST event name ID #REQUIRED-->

<!--ELEMENT sequence ((%routing_element; | state)+)-->
<!--ATTLIST sequence id ID #REQUIRED-->

<!--ELEMENT any_sequence ((%routing_element;)+)-->
<!--ATTLIST any_sequence id ID #REQUIRED-->

<!--ELEMENT choice ((%routing_element;)+)-->
<!--ATTLIST choice id ID #REQUIRED-->

<!--ELEMENT condition (expression, (true | false))-->
<!--ATTLIST condition id ID #REQUIRED-->
<!--ELEMENT true (%routing_element;)+-->
<!--ELEMENT false (%routing_element;)+-->

<!--ELEMENT ask EMPTY-->
<!--ATTLIST ask id ID #REQUIRED
    description CDATA #IMPLIED
    ask_parameter CDATA #REQUIRED
    ask_value CDATA #IMPLIED-->

<!--ELEMENT parallel_sync ((%routing_element;)+)-->
<!--ATTLIST parallel_sync id ID #REQUIRED-->

<!--ELEMENT parallel_no_sync ((%routing_element;)+)-->
<!--ATTLIST parallel_no_sync id ID #REQUIRED-->

<!--ELEMENT parallel_part_sync ((%routing_element;)+)-->
<!--ATTLIST parallel_part_sync id ID #REQUIRED
    number NMTOKEN #REQUIRED
    refs IDREFS #IMPLIED-->

<!--ELEMENT wait_all (event_ref | timeout)+-->
<!--ATTLIST wait_all id ID #REQUIRED-->

<!--ELEMENT wait_any (event_ref | timeout)+-->
<!--ATTLIST wait_any id ID #REQUIRED-->

<!--ELEMENT event_ref EMPTY-->
<!--ATTLIST event_ref name IDREF #REQUIRED-->

<!--ELEMENT timeout (%routing_element;)*-->
<!--ATTLIST timeout time CDATA #REQUIRED
    type (relative|s_relative|absolute) "absolute"-->

<!--ELEMENT while_do (expression, do)-->
<!--ATTLIST while_do id ID #REQUIRED-->
<!--ELEMENT do (%routing_element;)-->

<!--ELEMENT call (faultHandler?)-->
<!--ATTLIST call id ID #REQUIRED

```

```
        sub_workflow CDATA #REQUIRED>

<!ELEMENT find_executables EMPTY>
<!--ATTLIST find_executables id ID #REQUIRED-->

<!ELEMENT send (faultHandler?)>
<!--ATTLIST send id ID #REQUIRED
channel CDATA #REQUIRED
message CDATA #REQUIRED-->

<!ELEMENT receive (faultHandler?)>
<!--ATTLIST receive id ID #REQUIRED
channel CDATA #REQUIRED
message CDATA #REQUIRED-->

<!ELEMENT state (event+)>
```

Apêndice 4 - Workflow de negócios segundo a linguagem XRL estendida

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE route SYSTEM "c:\dtdXRLestendida.dtd">

<route id="r1" name="example" created_by="Tatiana" date="19082002">
  <body>
    <sequence id="s0">
      <parallel_sync id="ps1">
        <task id="t1" name="publ1"
          address="www.taskserver.com/publ1"/>
        <task id="t2" name="publ2"
          address="www.taskserver.com/publ2"/>
        <task id="t3" name="publ3"
          address="www.taskserver.com/publ3"/>
      </parallel_sync>
      <condition id="c1">
        <expression>
          <or>
            <and>
              <eq left="publ1.result" right="OK"/>
              <eq left="publ2.result" right="OK"/>
            </and>
            <and>
              <eq left="publ1.result" right="OK"/>
              <eq left="publ3.result" right="OK"/>
            </and>
            <and>
              <eq left="publ2.result" right="OK"/>
              <eq left="publ3.result" right="OK"/>
            </and>
          </or>
        </expression>
      </condition>
      <true>
        <sequence id="s1">
          <while_do id="wd1">
            <expression>
              <eq left="find_shipper.result" right="OK"/>
            </expression>
            <do>
              <task id="t4" name="find_shipper"
                address="www.taskserver.com/find_shipper"/>
            </do>
          </while_do>
          <parallel_no_sync id="pns1">
            <condition id="c2">
              <expression>
                <eq left="publ1.result" right="OK"/>
              </expression>
            </condition>
            <true>
              <sequence id="s2">
                <task id="t5" name="confirm_order1"
                  address="www.taskserver.com/confirm1"/>
                <task id="t6" name="ship_order1"
                  address="www.taskserver.com/order1">
                  <event name="e_order1"/>
                </task>
              </sequence>
            </true>
          </parallel_no_sync>
          <condition id="c3">
            <expression>
              <eq left="publ2.result" right="OK"/>
            </expression>
          </condition>
          <true>
            <sequence id="s3">
              <task id="t7" name="confirm_order2"
                address="www.taskserver.com/confirm2"/>
              <task id="t8" name="ship_order2">
```

```

        address="www.taskserver.com/order2">
        <event name="e_order2"/>
    </task>
</sequence>
</true>
</condition>
<condition id="c4">
    <expression>
        <eq left="publ3.result" right="OK"/>
    </expression>
    <true>
        <sequence id="s4">
            <task id="t9" name="confirm_order3"
                address="www.taskserver.com/confirm3"/>
            <task id="t10" name="ship_order3"
                address="www.taskserver.com/order3">
                <event name="e_order3"/>
            </task>
        </sequence>
    </true>
</condition>
</parallel_no_sync>
<while_do id="wd2">
    <expression>
        <or>
            <and>
                <eq left="publ1.result" right="OK"/>
                <not>
                    <event_conf name="e_order1"/>
                </not>
            </and>
            <and>
                <eq left="publ2.result" right="OK"/>
                <not>
                    <event_conf name="e_order2"/>
                </not>
            </and>
            <and>
                <eq left="publ3.result" right="OK"/>
                <not>
                    <event_conf name="e_order3"/>
                </not>
            </and>
        </or>
    </expression>
    <do>
        <sequence id="s5">
            <wait_all id="wal">
                <timeout time="2 days" type="relative"/>
            </wait_all>
            <task id="t11" name="billing_partial"
                address="www.taskserver.com/billing_partial"/>
        </sequence>
    </do>
</while_do>
</sequence>
</true>
</condition>
</sequence>
</body>
</route>

```