Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Modelagem e Análise de Objetos como Processos em CSP: Padrão de Projeto e Estudo de Caso

por
Renata Elaine Mesel Kaufman

Dissertação de Mestrado

Recife, fevereiro de 2003

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

Renata Elaine Mesel Kaufman

Modelagem e Análise de Objetos como Processos em CSP: Padrão de Projeto e Estudo de Caso

Este trabalho foi apresentado à Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito para a conclusão do mestrado na mesma.

ORIENTADOR:
Prof. Augusto César Alves Sampaio
CO-ORIENTADORA:
Profa. Ana Lúcia Caneca Cavalcanti

Kaufman, Renata Elaine Mesel
Modelagem e Análise de objetos como processos em CSP: padrão de projeto e estudo de caso / Renata Elaine Mesel Kaufman. – Recife : O Autor, 2003.
xii, 236 folhas : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2003.

Inclui bibliografia e apêndices.

1. Engenharia de software – Métodos Formais. 2. Verificação de modelos (Sistemas computacionais) – Estratégia em linguagem CSP-OZ. 3. Linguagem de especificação orientada a objetos – Padrão de projetos em CSP-OZ. 4. Prontuário eletrônico – Especificação do sistema GEHR. 5. Protocolos de comunicação (Saúde) – Comparação. I. Título.

004.41	CDU (2.ed.)	UFPE
005.117	CDD (21.ed.)	BC2003–139

Agradecimentos

- A minha família, especialmente aos meus pais, que nunca mediram esforços para que eu chegasse até aqui;
- A meu marido Ismar Kaufman, que sempre esteve presente nos momentos mais importantes da minha vida, torcendo e acreditando em mim, e a minha filha Sofia Kaufman, pela paciência e compreensão da ausência de sua mãe em vários finais de semana;
- Ao professor Augusto Sampaio, meu orientador e amigo, que me ensinou o que é ciência e que, antes de mais nada, foi um *orientador*;
- A professora Ana Lúcia, também minha orientadora e amiga, que esclareceu inúmeras dúvidas e revisou essa dissertação.
- Ao professor Alexandre Mota, meu amigo, que me deu muita força e apoio neste trabalho;
- Aos amigos Márcio Cornélios e Claudia Ribeiro, que sempre me ajudaram nos momentos difíceis desta jornada;
- Aos amigos Genaína Rodrigues, Klissiomara Dias, Bruno Oliveira, Sandro Ronaldo, Admilson Ribeiro pelos momentos divertidos e descontraídos que passamos juntos na universidade;
- A todos os amigos, colegas, professores e funcionários que conviveram comigo durante esse período em que estive na universidade.

Resumo

A fim de viabilizar maior interação entre profissionais da área de saúde, foram propostos protocolos de comunicação, dentre os quais destacamos: HL7, DICOM, CORBAMed e GEHR. Decidimos especificar formalmente um subconjunto de um sistema de prontuário eletrônico, baseado no modelo GEHR, pois existe um interesse real nesta formalização, pela comunidade do projeto GEHR. As vantagens da formalização são óbvias: descobrir e corrigir erros nas fases iniciais da construção de software, especificar sistemas consistentes e livres de ambigüidades, obter implementações que comprovadamente correspondem às especificações, enfim, aumentar a confiabilidade e a robustez do software e reduzir custos de desenvolvimento e evolução.

Definimos uma estratégia adequada à especificação e análise de tais sistemas. Adotamos o uso integrado de UML-RT (extensão de UML para tempo real) e a linguagem formal CSP-OZ. Estas linguagens permitem um expressão natural do modelo orientado a objetos do GEHR e o uso combinado destas notações oferece as vantagens do formalismo (CSP-OZ) com o apelo prático da notação gráfica (UML-RT). Além disso, já existe uma técnica de verificação de modelos em CSP-OZ, utilizando a ferramenta FDR, que permite a verificação automática de algumas propriedades do sistema como, por exemplo, ausência de *deadlock* e *livelock*.

Constatamos que a técnica de verificação de modelos em CSP-OZ não trata os aspectos de orientação a objetos como herança, a qual é essencial para especificação do GEHR. Para sanar este problema, definimos um padrão de projeto, em CSP, que incorpora as principais características de orientação a objetos, tais como classes, subclasses, criação e remoção dinâmica de objetos. Este padrão, juntamente com a técnica de verificação de modelos desenvolvida para CSP-OZ, permite a análise de especificação considerando herança. No padrão proposto, objetos são representados como processos. A criação de objetos é realizada através de uma função que recebe uma tupla representando o estado do objeto e retorna o processo correspondente. A remoção de um objeto é implementada por eventos que, via sincronização, coordenam a terminação do processo que representa o objeto.

A principal contribuição do trabalho é o padrão de projeto e a sua utilização na especificação formal de um subconjunto do sistema de prontuário eletrônico baseado em GEHR. Além do padrão de projeto, sugerimos duas outras abordagens em CSP, que mostram alternativas para modelar herança.

A contribuição da especificação formal do GEHR dirige-se à produção de software complexo e real com alta qualidade. Os profissionais de saúde podem ter acesso a um subconjunto do sistema de prontuário sem ambigüidades e livre de erros. Um dos objetivos futuros é a formalização completa deste sistema.

Abstract

In order to promote adequate interaction between health professionals, communication protocols in the health sector have been proposed, like HL7, CORBAMed, DICOM and GEHR, among others. We present a formal specification of a subset of a GEHR based system, motivated by the fact that the GEHR project community expressed interest in this formalization. Benefits from formalizing the protocol are obvious: uncover and correct errors early development phases, achieve consistent and un ambiguity specifications, get implementations that are proven to satisfy the specification, and finally, improve software reliability and robustness, and reduce development and evolution costs.

We introduce a strategy which we claim to be adequate to the specification and analysis of such systems. We adopt the integrated use of UML-RT (a real extension to UML) and the CSP-OZ formalism. These languages allow a natural expressiveness of GEHR's object-oriented model, and their combined use offers the advantages of formality (CSP-OZ) together with the practical appeal of a graphical notation (UML-RT). Moreover, there is a model-checking strategy for CSP-OZ specification, using the FDR tool, that allows automatic verification of some system properties, like deadlock and livelock freedom.

In the course of work, it became evident that the model-checking strategy originally developed for CSP-OZ specifications does not deal with object-oriented features, such as inheritance, which is essential for the GEHR specification. To solve this problem, we defined a design pattern, in CSP, which incorporates the main object-oriented features, such as classes, subclasses, creation and dynamic removal of objects. This pattern complements the model-checking strategy cited before to allow for inheritance in specifications.

The pattern represents objects as processes. Object creation is performed by a function which receives a tuple (representing the object state) and returns the corresponding process. Object removal is implemented by events which, through sincronization, coordinate the termination of the process that represents the object.

The main contribution of this work is the design pattern and its use in the formal specification of a subset of the GEHR based patient record system. Besides the proposed design pattern, we suggested two alternative approaches that also deal with inheritance in CSP.

The contribution of the formal specification of the GEHR case study is directed towards the production of high quality, complex and real software. Health professionals can have access to a subset of the GEHR based system without ambiguities and errors. One of the future aims is complete the formalization of this system.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este Capítulo apresenta uma visão geral do trabalho desenvolvido nesta dissertação, descrevendo a motivação, os principais objetivos e as soluções propostas.

Sistemas computacionais de suporte à área de saúde são inerentemente críticos e, portanto, candidatos naturais à utilização de métodos e técnicas modernas da Engenharia de Software - em particular, os métodos formais. Entretanto, vários aspectos mais básicos precedem a decisão de adotar ou não formalismos para a concepção, modelagem e implementação de um sistema para a saúde. Particulamente, a padronização das informações médicas é essencial para os sistemas de informação em saúde, pois ela permite a comunicação entre diferentes softwares, equipamentos e profissionais.

A comunicação entre médicos é essencial para uma boa assistência ao paciente. Como cada instituição de saúde possui os seus sistemas de informação, para viabilizar a comunicação entre médicos, é necessária a definição de um conjunto de padrões de representação e troca de informação. A padronização compreende não apenas os aspectos de hardware e software, mas, também, os aspectos de representação, transmissão, acesso e armazenamento da informação em saúde. Sem padronização é impossível compartilhar dados em meio eletrônico, processá-los em locais e equipamentos diversos, agregá-los, etc. Além disso, quando dados, recursos computacionais e recursos de telecomunicações são padronizados dentro de uma instituição ou organização, é possível reduzir o número de especializações exigidas de técnicos e outros funcionários, ficando mais fácil, desta forma, fazer rotações ou substituições dentro do quadro de pessoal.

De acordo com a definição da Organização Internacional de Padronização (International Standards Organization - ISO), padrão é um documento estabelecido por consenso e aprovado por um grupo reconhecido, que estabelece para uso geral e repetido um conjunto de regras, protocolos ou características de processos, com o objetivo de ordenar e organizar atividades em contextos específicos para o benefício de todos [45].

Software destinado a aplicações em saúde geram dados no nível do indivíduo e no nível da instituição. Esses dados são subseqüentemente filtrados, consolidados e utilizados nos níveis institucional, regional, nacional ou internacional. Para permitir e facilitar a transmissão e a agregação desses dados, é essencial que todo software que gera dados de saúde utilize padrões para captura, representação, codificação, disseminação de dados, etc.

Com relação ao suporte de métodos formais para modelagem de tais sistemas, várias linguagens de especificação formal foram desenvolvidas. Dentre elas podemos destacar Z [20], que é particularmente voltada para a descrição detalhada do estado e operações associadas de sistemas seqüenciais. Dentro do contexto de softwares complexos, sistemas paralelos e interativos tomam grande importância. CSP (Communicating Sequential Processes) é uma das linguagens que permite descrever a ordem em que as operações acontencem no tempo. Foi originalmente desenvolvida por C.A.R. Hoare como álgebra de processos em meados dos anos 70 [39] para tratar as questões de concorrência.

As linguagens Z e CSP foram projetadas para modelar diferentes e complementares aspectos dos sistemas: dados e interação, respectivamente. Explorando a complementaridade de notação, a comunidade de Métodos Formais tem dedicado grande esforço em definir linguagens unificadas. Como não existe nenhum método formal que individualmente cubra todos os aspectos dos sistemas complexos, esforços têm sido feitos na combinação das técnicas baseadas em estado como Z [20], VDM [47] ou B [1] com as linguagens comportamentais como CSP [40] ou CCS [58]. A vantatem de integrar diversas técnicas é permitir a escrita de especificações mais adequadas, escolhendo convenientemente a melhor técnica para cada aspecto do sistema. Aspectos de tempo, probabilidade

e mobilidade também podem ser integrados, de acordo com a demanda da aplicação

Como exemplo de integração entre dados e interação, temos a linguagem CSP-OZ, que é um método formal orientado a objetos que integra Object-Z e CSP. Object-Z [21] é um método orientado a estados adequado para descrever tipos de dados e operações, e CSP, como mencionado anteriormente, é adequada para descrever o comportamento dinâmico de sistemas comunicantes distribuídos. A idéia principal para a combinação destas linguagens é tratar as classes Object-Z como processos. Neste trabalho, nós propomos uma estratégia de especificação e análise, adequada à modelagem de protocolos na área de saúde, que utiliza CSP-OZ.

Embora as especificações formais sejam utilizadas para melhorar a confiabilidade dos softwares e capturar os requisitos adequadamente, elas podem conter erros. Se estes erros não forem detectados, eles poderão comprometer a qualidade do produto final. Então, para minimizar a ocorrência de erros, é necessário provar algumas propriedades do sistema, tais como ausência de *deadlock*, *livelock* e não-determinismo. Existem basicamente três técnicas utilizadas para a verificação de propriedades:

- animação: Uma especificação é convertida para uma forma executável, de forma que ela possa ser executada quase como um programa normal, pois pode gerar várias saídas a partir de uma mesma entrada. A conversão pode não ser completamente automática em todos os casos.

- prova de teorema: Uma especificação é fornecida a um provador de teorema e então as propriedades relevantes são provadas. Normalmente, esses provadores necessitam de ajuda humana para se chegar a algum resultado.

- verificação de modelos (model-checking): Para cada estado alcançado da especificação, um verificador de modelos verifica se a propriedade é verdadeira. Se ela for para todos os estados, então ela é verdadeira para o sistema. Caso contrário, um contra-exemplo é apresentado. Esta técnica só pode ser aplicada em sistemas com número finito de estados.

Dentre as três técnicas apresentadas acima, nós estamos interessados na de verificação de modelos, porque ela é totalmente automática.

Escolhemos a técnica de verificação de modelos proposta por Fischer [30] porque ela é própria para especificações em CSP-OZ. Porém, detectamos que esta técnica não trata herança, inviabilizando a análise da especificação de um sistema de prontuário eletrônico (estudo de caso), que utiliza um modelo orientado a objetos. Para resolver este problema, criamos um padrão de projeto em CSP que representa objetos como processos, permitindo, então, a modelagem e análise de sistemas orientados a objetos.

1.1 Objetivos

Os objetivos desta dissertação são sumarizados a seguir.

1. Definição de um processo, com ênfase em especificação de requisitos, arquitetura e análise (verificação de propriedades), para suporte ao desenvolvimento formal de sistemas orientados a objetos, integrando a notação gráfica UML-RT [25] e a linguagem formal CSP-OZ.

2. Desenvolvimento de uma técnica de verificação de modelos para especificações em CSP-OZ, juntamente com o padrão de projeto elaborado para viabilizar a análise das propriedades de sistemas orientadas a objetos. O desafio de encontrar este padrão de projeto resultou na contribuição central da dissertação.

3. Aplicação do processo e da técnica de verificação de modelos a um estudo de caso na área de saúde.

4. Criação de um documento formal contendo a modelagem do sistema de prontuário eletrônico GEHR servindo como ponto de partida para a formalização completa deste sistema. Em [10], o GEHR foi formalizado utilizando a notação BON [50], a qual é semi-formal porque a sua semântica não é totalmente formal, precisa. Existe um interesse em especificar o GEHR formalmente utilizando uma linguagem formal como Z++, Object-Z ou B [10], porque são adequadas à natureza do modelo, que é orientada a objetos. O motivo da comunidade não ter ainda começado a formalização do modelo GEHR é a falta de ferramentas comerciais e as dificuldades de construir softwares utilizando as mesmas. O grupo do projeto GEHR se mostrou bastante receptivo ao nosso esforço de especificar o GEHR em CSP-OZ.

1.2 Estrutura da Dissertação

Estruturamos o trabalho da seguinte forma. No Capítulo 2, apresentamos os principais protocolos na área de sáude, inclusive o GEHR, o qual faz parte do nosso estudo de caso. Além disso, exibimos um comparativo entre eles, através do qual podemos identificar pontos positivos e negativos de cada um.

No Capítulo 3, introduzimos as linguagens UML-RT e CSP-OZ, utilizando exemplos simples para ilustrar os principais aspectos sintáticos e semânticos destas linguagens. Além disso, apresentamos uma formalização, em Object-Z, dos componentes da linguagem UML-RT. Mostramos, também, uma tradução formal dos diagramas de colaboração em UML-RT para processos em CSP-OZ, e uma técnica de verificação de modelos. Todos esses elementos fazem parte da definição de uma estratégia de modelagem, a qual é descrita também neste capítulo e utilizada no estudo de caso do Capítulo 5.

Em seguida, apresentamos o Capítulo 4, que introduz um padrão de projeto implementado em CSP, o qual trata os aspectos de herança, não considerados pela técnica de verificação de modelos que adotamos [30]. Além disso, mostramos soluções alternativas para o problema de tratar herança em uma álgebra de processos.

No Capítulo 5, aplicamos a estratégia de modelagem descrita no Capítulo 3 para especificar e analisar um subconjunto de um sistema de prontuário eletrônico baseado no GEHR.

Para finalizar, no Capítulo 6, apresentamos as contribuições do nosso trabalho e as propostas de trabalhos futuros.

Capítulo 2

Protocolos

Este Capítulo apresenta o projeto GEHR*, que tem como objetivo a definição de um modelo arquitetural de registro eletrônico de paciente, e apresenta uma visão geral de outros protocolos da área de saúde.*

2.1 Introdução

Os médicos estão cada vez mais conscientes de que os sistemas de informação podem ser um apoio efetivo às práticas clínicas, como, por exemplo, no gerenciamento de doenças complexas, auditoria clínica, geração automática de relatórios, etc.

A crescente complexidade dos serviços de saúde faz com que os profissionais da área necessitem de informações sobre o paciente, como, por exemplo, o histórico das suas doenças, se tem alergia a algum medicamento, se possui alguma dieta alimentar, etc. Estas informações usualmente se encontram espalhadas em diversas instalações de saúde por onde o paciente recebeu cuidados clínicos. Reunir todas essas informações e disponibilizá-las na rede facilita a vida do paciente. Imagine alguém com uma determinada patologia, ou alérgico à penicilina, que sofre um acidente na rua. Inconsciente, é levado para o pronto-socorro, onde recebe uma dose do medicamento. Ninguém sabe que essa pessoa não pode tomar o antibiótico. Se o médico tiver o registro eletrônico desse paciente, então saberá da sua alergia. Registro eletrônico é um conjunto de documentos padronizados, ordenados e concisos, destinados ao registro dos cuidados médicos prestados ao paciente pela instituição de saúde. Infelizmente, os sistemas eletrônicos atuais, usados na maioria dos hospitais e clínicas, não têm condições de fornecer estes tipos de informação, porque não foram modelados de forma adequada.

Os médicos necessitam se comunicarem uns com outros por diversos motivos, dentre eles, solicitar uma segunda opinião com relação à doença de um paciente. Para que o objetivo da comunicação possa ser alcançado, é necessário resolver uma série de problemas, como, por exemplo, as dificuldades de interação dos profissionais da área de saúde com o computador, a padronização da terminologia médica a ser utilizada e estabelecer padrões para a comunicação entre os diferentes softwares e equipamentos computadorizados já existentes. Vamos apresentar alguns protocolos da área de saúde que viabilizam a comunicação entre sistemas.

2.2 HL7

Apresentaremos a seguir o protocolo HL7 em duas partes: a primeira descreve um breve histórico, suas principais características, e sua abrangência; a segunda foca as modificações realizadas no mesmo para atender os novos requisitos dos usuários.

2.2.1 HL7 Clássico

Health Level 7 (HL7) é o nome do padrão e da organização que desenvolve este padrão. A organização HL7 [37] foi fundada por fornecedores de saúde em 1987, para tratar dos padrões para troca de informações, tais como dados de registro de pacientes, informações sobre a estadia do paciente no hospital, informação de pedidos de medicamentos, dados de marcação de exames, e observações clínicas (relatórios e resultados). "Level Seven" se refere ao nível de aplicação, que é o nível mais alto do modelo OSI (Open System Interconnection) [84], o qual pertence a ISO (International Organization for Standardization) [45]. A organização conta com 450 membros organizacionais e possui filiações em numerosos países (Austrália, Canadá, Finlândia, Alemanha, Japão, Nova Zelândia, África do Sul, Índia, Holanda e Reino Unido, entre outros).

O HL7 trata a troca de dados dentro de mensagens. Uma mensagem pode ser uma resposta a uma consulta ou atualização não solicitada. Uma consulta pode ser tão simples quanto solicitar ao sistema de laboratório todos os resultados de exame de um paciente. Atualizações não solicitadas tipicamente contêm dados sobre a conclusão de uma ação a respeito de um determinado paciente e são realizadas sem que ele tenha feito uma solicitação de consulta. É possível consultar ou reconsultar alguma mensagem que já tenha sido enviada. Tanto as mensagens de consulta quanto as de atualização não solicitada, podem ser de visualização ou orientada a registro.

Mensagem de visualização é usada quando a informação não necessita ser capturada pelo sistema receptor; ela é meramente apresentada ou impressa e então formatada com o propósito de visualização. Uma mensagem orientada a registro é aquela em que o seu formato denota explicitamente o conteúdo do registro, e então pode ser capturada pelo sistema receptor. As duas são mensagens de texto em ASCII, as quais podem ser lidas.

Uma mensagem HL7 orientada a registro é tipicamente composta de numerosos campos de dados, tais como nome do paciente, endereço, idade, os quais são associados a um tipo de dado específico, como, por exemplo, *string*, *date* ou *numeric*, e cujo comprimento já inclui os separadores de campos. Os campos são combinados em agrupamentos lógicos chamados de segmentos, tais como identificação do paciente, alergia, e parentesco, os quais podem ser opcionais e/ou repetidos para uma mensagem particular e apresentados numa determinada ordem dentro da mensagem.

O momento da troca da mensagem pode ser governado pelos eventos *trigger* (gatilho). As mensagens são mandadas quando um evento *trigger* ocorre ou então são reunidas em lotes e enviadas de uma só vez. O padrão HL7 assume que um evento que ocorre no mundo real, na área de saúde, cria a necessidade de dados serem trocados entre duas ou mais aplicações. Por exemplo, a maioria dos hospitais usam um sistema ELT (entrada/liberação/transferência), o qual faz todo o controle da visita do paciente no hospital (quando foi internado, se foi para algum bloco cirúrgico, se foi transferido para algum outro hospital, quando foi liberado, etc.), e um sistema de contas (recebimentos), que registra todos os gastos do paciente. Então, quando o paciente é admitido no hospital, suas informações são coletadas e inseridas pelo sistema ELT. Uma mensagem de notificação (atualização não solicitada) é enviada do ELT para o sistema de contas, solicitando a criação de um registro de conta do paciente.

O padrão HL7 também segue os protocolos do modelo OSI que governam a comunicação de mensagens de erro entre dois sistemas. Quando a informação é trocada entre dois sistemas, o sistema receptor valida a mensagem e retorna uma mensagem de resposta para o sistema solicitante, indicando que a informação foi recebida com sucesso ou não pode ser interpretada pelo sistema receptor. A aplicação receptora normalmente armazena a mensagem e manda uma mensagem de reconhecimento para a aplicação solicitante.

A versão 2.3 HL7 é considerada padrão pela ANSI (American National Standards Institute) e ainda é dominante na área médica. Esta versão utiliza mecanismos clássicos de mensagem para a troca de informação textual e tem dado pouca atenção aos tipos de informação como imagem, informação altamente estruturada, ou mesmo modelo de dados. Modelos de dados estão escondidos dentro das estruturas das mensagens e não estão explicitamente definidos. As seguintes funcionalidades são cobertas:

- Clínica: laboratório, farmácia, radiologia, dieta, cuidado com o paciente, saúde

pública e outros tipos de serviços de diagnóstico.

- Administrativa: admissão de paciente, contas de paciente.

2.2.2 HL7 Moderno

Sentindo a crescente demanda por compartilhamento de várias categorias de informação entre áreas médicas de uma única organização de saúde ou de várias organizações de saúde, a HL7 está trabalhando na versão moderna 3.0. A motivação da versão 3.0 é a identificação das deficiências na versão 2, tais como: dificuldades de implementação, falta de suporte explícito às funções de segurança, ausência de um modelo de referência de informação, falta de suporte às tecnologias XML [87], Web e orientação a objetos, opcionalidade espalhada que trouxe ambigüidades na interpretação dos dados, dificuldade em medir o progresso, etc. Candidata a ser padrão no início de 2003, a versão 3.0 do HL7 resolve as deficiências citadas acima, melhorando a qualidade das mensagens.

As versões 2.x foram construídas com a mentalidade "cresça enquanto avança", de forma não estruturada (sem modelo). A versão 3.0 está sendo construída de maneira totalmente diferente, utilizando tecnologia de modelagem orientada a objetos.

O modelo de referência de informação (RIM)[1] está sendo construído há vários anos. Ele é uma visão ampla de todo o escopo de saúde, contendo mais de 100 classes e mais de 800 atributos usados para criar mensagens HL7. O RIM megulha profundamente em cada área, mapeando e definindo os relacionamentos de cada classe. Ele é uma rígida espinha dorsal sobre a qual a versão HL7 3.0 é construída. Desde que cada aspecto do RIM está bastante detalhado, pouca opcionalidade existe. Isto resolve as ambigüidades que surgiram nas versões 2.x. Além disso, a melhoria do desenvolvimento das mensagens permite mais eventos gatilhos e formatos de mensagens. Uma vez completa esta versão, o HL7 passará de um "livro de regras", que guia discussões e negociações, para um padrão de tecnologia de informação em saúde verdadeiramente testado.

A versão 3.0 oferece as seguintes funções adicionais em relação a 2.x:

- Utiliza a linguagem XML (Extensible Mark up Language) para aumentar a interoperabilidade entre sistemas.

  O HL7 tem desenvolvido o Patient Record Architecture (PRA), uma arquitetura de documentos clínicos baseados em XML, que fornece um modelo de troca para documentos de níveis de complexidade variados. Usando o PRA, a versão habilita sistemas a criar documentos XML, que são incorporados no conteúdo da mensagem HL7, e a trocar e processar mensagens e documentos entre sistemas.

- Certifica sistemas de vendedores através do MDF (Message Development Framework) [38] do HL7, que é um conjunto de testes que verifica se aplicação é compatível com o padrão HL7.

- Inclui métodos para permitir que mensagens HL7 sejam construídas sobre código e vocabulários de uma variedade de fontes. Isto fará com que sistemas que enviam e recebem mensagens entendam claramente os códigos e seus domínios de valores.

[1] RIM é o *Reference Information Model*

- Inclui IMSIG (Image Management Special Interest Group), cujo propósito é assegurar que especificações HL7 para interface entre sistemas de imagem e sistemas de informação sejam compatíveis com padrões DICOM [19], MEDICOM [53] (padrão adotado pelo comitê Europeu que adotou partes do padrão DICOM) e JIRA [46] (Japan Industries Association of Radiological Systems).

Um benefício adicional do HL7 versão 3.0 é que ele inclui novos formatos de troca de dados além do ASCII. Ele também procura suportar tecnologias baseadas em componentes como activeX [57] e CORBA [65].

2.3 CORBAMed

CORBAMed [66] é a força tarefa da OMG (Object Management Group) que tem como objetivo definir padrões para vários serviços de saúde. O principal produto da OMG é o padrão CORBA (Common Request Broker Architecture), o qual propõe uma arquitetura de software para suportar a distribuição e garantir a interoperabilidade entre as diversas plataformas de hardware e sistemas operacionais. Resumidamente, o padrão CORBA possibilita a comunicação entre aplicações independentemente de sua localização ou ambiente, através da especificação de como ORBs (Object Request Broker) de diferentes fornecedores de software podem interoperar.

ORB é o middleware que estabelece os relacionamentos cliente-servidor entre os objetos. Utilizando um ORB, um cliente pode invocar um método de um objeto no servidor, que pode estar na mesma máquina ou em qualquer ponto da rede. O ORB intercepta a chamada por parte do cliente e tem a responsabilidade de localizar o objeto que implementa a chamada, passar os parâmetros necessários a este objeto, fazer a chamada dos métodos, e retornar os resultados. O cliente, por sua vez, não precisa se preocupar com a localização do objeto que está sendo invocado, com a linguagem que o mesmo foi programado, ou com o sistema operacional utilizado. Ao realizar esta tarefa, o ORB oferece uma solução para a interoperabilidade entre aplicações em máquinas diferentes em ambientes distribuídos e heterogêneos, ao mesmo tempo que interconecta múltiplos sistemas baseados em objetos. Especificações CORBA são usadas para desenvolver aplicações distribuídas para os mercados verticais, incluindo manufatura, transporte, finanças, telecomunicações, e também saúde.

CORBAMed define tecnologias para interoperabilidade em toda comunidade de saúde. Especificamente, ela define interfaces orientadas a objetos padronizadas entre serviços relacionados à saúde, e funções para promover um alto grau de interoperabilidade entre uma variedade de plataformas, linguagens e aplicações. CORBAMed tem como objetivos:

- Melhorar a qualidade da assistência e reduzir custos, com o uso da tecnologia CORBA;
- Definir interfaces padrão orientadas a objetos entre serviços e funções relativos a saúde;
- Promover interoperabilidade entre as várias plataformas, sistemas operacionais, linguagens de programação e aplicativos;
- Utilizar o processo de padronização da OMG.

CORBAMed tem adotado vários padrões dentro do domínio de saúde. Baseado nesses padrões, vendedores têm implementado componentes em conformidade com essas interfaces. Segue um resumo dos padrões propostos e adotados pelo CORBAMed:

Resource Access Decision (RAD): Esse padrão [69], adotado pela OMG, fornece mecanismos para obter decisões de autorização. Para entender o que significa decisão de autorização, é necessário introduzir alguns conceitos relacionados a políticas de acesso (segurança) e controle de acesso. Uma política de acesso define requisitos de segurança de um sistema. Esses requisitos de segurança decidem como e quando os clientes (usuários ou sistemas) operam ou acessam recursos do sistema.

Um mecanismo de controle de acesso intermedia a concessão ou recusa dos pedidos solicitados pelos cliente para acessar recursos assegurados. Durante a intermediação, ele avalia as políticas de acesso apropriadas e os resultados dessas avaliações determinam se um pedido é aceito ou negado. A decisão de aceitar ou negar acesso a um recurso, baseado na avaliação da política de acesso, é conhecida como uma decisão de autorização.

O principal objetivo do RAD é separar a lógica de autorização da aplicação, da lógica da aplicação em si. A lógica de autorização está encapsulada em um serviço de autorização, o qual é externo à aplicação. Normalmente, encontramos a lógica da autorização como parte da aplicação. Um esquema simplificado de interações entre Aplicação Cliente, Servidor de Aplicação, e uma instância do RAD é apresentado pela Figura 2.1. Como podemos observar, a sequência de interações acontece da seguinte forma:

1. A Aplicação Cliente solicita um serviço ao Servidor de Aplicação.
2. Enquanto processa o pedido, a aplicação solicita uma decisão de autorização do RAD.
3. O RAD fornece uma decisão de autorização para a aplicação.
4. Se for dada a autorização, a aplicação realiza a operação e retorna os resultados esperados para a Aplicação Cliente. Se não for dada a autorização, ela poderá fornecer resultados parciais ou avisar que o acesso foi negado.

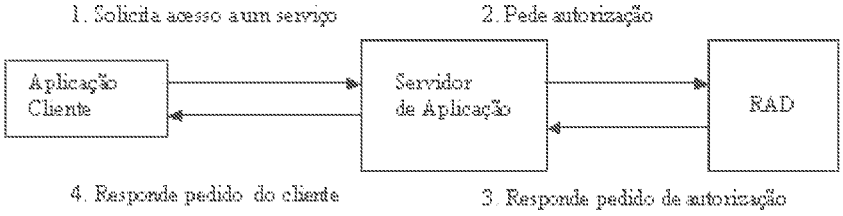


Figura 2.1: Interações entre Aplicação Cliente, Servidor de Aplicação e RAD

Person Identification Service (PIDS): Os pacientes podem receber cuidados clínicos em diversas instituições de saúde, as quais possuem o seu sistema de informação. Cada sistema tem uma maneira de identificar o paciente para obter as suas informações clínicas. Para que se torne viável a comunicação destes sistemas, é necessário um serviço de identificação de paciente que consiga identificar um mesmo paciente nos diversos sistemas. O PIDS [68] tem como objetivo permitir a identificação de uma mesma pessoa em diferentes sistemas de informação, sendo capaz de encontrar identificadores de pessoas a partir de um conjunto de características conhecidas, correlacionar identificadores, atribuir identificadores a pessoas, recuperar informações de identificação de pessoas, avisar a outros sistemas quando uma modificação numa característica de uma pessoa ocorrer, entre outras funcionalidades.

Por definir uma interface padronizada, este serviço permite que diferentes instituições, com sistemas de informação diferentes, criados em plataformas de hardware e software diferentes, troquem informações entre si de forma transparente. Embora originalmente intencionado para identificação do paciente, essa especificação tem provado ser efetiva na identificação de outros elementos dentro de um domínio designado, enquanto existe um conjunto de marcas apropriadas permitindo a busca.

Terminology Query Service (TQS): Esforços têm sido feitos para conseguir representar precisamente informações de saúde, tais como os dados de observações ou dados históricos. Essa capacidade de representar um conceito não ambíguo num formato que possa ser lido pela máquina é a chave para um melhor gerenciamento dos processos clínicos dentro de uma organização de saúde ou entre uma organização de saúde e seus parceiros de negócios. A idéia de permitir léxicos (dicionários ou vocabulários) codificados é de extrema importância dentro do segmento de saúde. É importante fazer a distinção entre conteúdo léxico (vocábulo, termo) e métodos para permitir consultas de léxicos.

A especificação TQS [70] define interfaces para características comuns do conjunto de serviços de consultas de léxicos. Ela também descreve os requisitos para os serviços suportarem léxicos (dicionários codificados) em um sistema de objetos distribuídos.

Essa especificação fornece acesso e tradução entre terminologias privadas de uma determinada instituição e informações públicas codificadas. Além disso, assegura que o significado semântico da informação é mantido, garante um apoio de decisão clínico confiável, fornece uma infra-estrutura para facilitar o desenvolvimento de aplicações e tecnologias, e melhora a comunicação entre sistemas diferentes.

Clinical Observation Access Service (COAS): Fornece a especificação de interfaces necessárias para acessar informações do domínio da saúde. Embora a especificação COAS [63] seja chamada de Serviço de Acesso a Observação Clínica, ela tem a capacidade de recuperar todos os tipos de informações do domínio da saúde, e não somente as observações clínicas. Isto é importante porque não existem apenas médicos como usuários dos sistemas de informação de saúde. Além deles, podemos citar administradores, pesquisadores, educadores, etc, que necessitam de outros tipos de informações para o seu trabalho.

Observações clínicas representam quantidades significativas de informações registradas de um paciente. Exemplos de observações clínicas incluem: resultados de exames laboratoriais, sinais vitais, observações e avaliações objetivas e subjetivas, e observações

e medições fornecidas por especialistas, tais como radiologistas ou patologistas, que interpretam imagens e outros dados multimídias. Informações que também podem ser recuperadas, mas que não são clínicas incluem: informações demográficas do paciente, faturamento, suprimentos médicos, informação de dieta, informações administrativas, tais como census, etc.

Health Care Information Locator Service (HILS): Essa especificação tem o propósito de atender a necessidade de localizar informação pertencente a um indivíduo, população, etc. Voltada a complementar o PIDS, HILS [67] leva em consideração a localização da informação e retorno dos dados resumidos sobre registros identificados que casam com algum critério especificado.

A prática de manter registros médicos no local de cuidado com o paciente, dado que pacientes e populações de pacientes podem migrar através de vários locais de cuidado, cria dificuldades de unir e disponibilizar as informações quando necessárias. As duas especificações, PIDS e HILS, procuram tornar o acesso às informações de saúde mais fácil e mais eficiente.

Clinical Image Access Service (CIAS): É uma especificação que tem o propósito de tratar as questões pertencentes a recuperação de imagens clínicas, tais como Digital Imaging and Communication in Medicine (DICOM), que será explicado mais adiante, e tipos de dados elaborados [64]. Ela também fornece um meio para transformar imagens de formatos não comuns à Internet, como DICOM, para formatos padrões de Internet, tais como jpeg e gif. CIAS só trata o acesso às imagens clínicas. Informações de tempo real ou de vídeo são excluídas dessa especificação.

O escopo da CIAS é o fornecimento de um mecanismo não ambíguo para acessar informações médicas e metadados relacionados, de forma que seja útil para profissionais em geral e especialistas clínicos, que não estão preocupados com a qualidade da imagem. Através desse serviço especializado de recuperação, considerado uma especialização do COAS, o requisitante é capaz de recuperar e manipular imagens clínicas de maneira uniforme e consistente através de aplicações e dispositivos.

2.4 DICOM (Digital Imaging and Communications in Medicine)

Com a introdução do tomógrafo computadorizado, seguido de outras modalidades de diagnóstico de imagem digital nos anos 70, e o crescente uso de computadores nas aplicações clínicas, o American College of Radiology (ACR), juntamente com a National Electrical Manufacturers Association (NEMA), reconheceram a necessidade de um método padrão para transferência de imagens e informações associadas entre dispositivos fabricados por diferentes fabricantes. Esses dispositivos produzem uma variedade de formatos de imagem.

O ACR juntamente com a NEMA formaram um comitê em 1983 para desenvolver um padrão para:

- Promover comunicação da informação da imagem digital, desconsiderando o fabricante do dispositivo.

- Facilitar o desenvolvimento de sistemas de arquivamento e comunicação de imagens (PACS) que podem se comunicar com outros sistemas de informação hospitalar.

  PACS (Picture Archiving and Communication System) fornece um sistema para o armazenamento e disponibilização de imagens em um ambiente hospitalar em geral. Ele possibilita a visualização de imagens para diagnósticos, laudos, e consultas em estações locais ou remotas. As imagens podem estar armazenadas em dispositivos óticos ou magnéticos. A comunicação e intercâmbio de imagens podem ser realizadas em redes locais ou públicas. PACS deve fornecer diversas modalidades de interfaces e acesso às facilidades dos sistemas de informação já implantados nos departamentos de um centro médico.

- Permitir a criação de bases de dados de informação de diagnóstico que podem ser interrogadas através de uma variedade de dispositivos distribuídos geograficamente.

A versão 1 foi publicada em 1985, a qual recebeu ainda duas revisões. Em 1988 foi publicada a versão 2, que inclui novo material para fornecer capacidade de comando para dispositivos de visualização, introduz um novo esquema hierárquico para identificar uma imagem, e adiciona elementos de dados para descrever uma imagem.

O padrão DICOM fornece uma capacidade precisa em relação à exibição, armazenamento, administração, e comunicação de imagens médicas, de forma que o seu propósito é compor o diagnóstico clínico. Ele é projetado para produzir uma grande variedade de serviços e descrever uma grande quantidade de classes e formatos de imagem. Esta variedade de serviços contribui para apoiar as necessidades das especialidades, tais como: radiologia, cardiologia, patologia, gastroenterologia, etc. Esse padrão especifica uma interface de hardware, um conjunto mínimo de comandos de software, e um conjunto de formatos de dados consistentes.

Atualmente o padrão DICOM se encontra na versão 3, a qual embute um maior número de melhorias em relação às versões prévias:

1. Aplicabilidade a ambiente de rede integrado.

   As versões anteriores eram aplicáveis em ambiente ponto a ponto. Para suportar operação em um ambiente de rede foi requisitado uma unidade de interface de rede (NIU). O DICOM utiliza protocolos de rede padrão tais como OSI e TCP/IP.

2. Especificação da exigência de conformidade dos dispositivos para reagir a comandos e dados sendo trocados.

   Versões anteriores eram confinadas à transferência de dados, porém a versão 3 especifica, através de conceito de classes de serviço, as semânticas dos comandos e dados associados.

3. Especificação de vários níveis de conformidade.

   Versões anteriores especificam apenas um nível conformidade.

4. Estruturação como um documento de múltiplas partes.

   Isto facilita a evolução do padrão simplificado em relação à adição de novas características. As diretivas da ISO que definem como estruturar documentos de múltiplas partes têm sido seguidas na construção do padrão DICOM.

5. Introdução de objetos de informação explícitos, não somente para imagens e gráficos, mas também para estudos de caso, relatórios, etc.

6. Especificação de uma técnica para identificação única de algum objeto de informação.

   Isto facilita definições não ambíguas de relacionamentos entre objetos de informação, quando eles são representados através da rede.

As versões 1 e 2 contaram com um modelo de informação implícito que é usado nos departamentos de radiologia. Os elementos de dados foram agrupados baseados na experiência dos projetistas, e embora o mapeamento fosse imperfeito, a estrutura da mensagem permitia que a informação necessária pudesse ser transmitida. Em contraste, o DICOM v3 possui modelos explícitos e detalhados de como pacientes, imagens, relatórios, etc. envolvidos nas operações de radiologia são descritos e como eles estão relacionados.

A importância da modelagem cresce ainda mais quando existe a necessidade de conhecer o contexto da informação considerando comunicações de rede. No ambiente ponto–a–ponto, o usuário conhece exatamente quais dispositivos estão conectados e o que eles fazem. Centenas de dispositivos podem estar ligados através de uma rede e alguns dispositivos podem ser reconfigurados dinamicamente para manipular diferentes tarefas e carga de dados. Isto significa que pode não ser possível conhecer que dispositivos podem fazer o que. Os dispositivos podem ter que negociar para estabelecer o nível comum sob o qual se constroi as comunicações necessárias para realizar a tarefa que o usuário solicitou.

O DICOM trabalha com modelos orientados a objetos. Os objetos no seu modelo são chamados de objetos de informação (IO) e seus atributos, definições de objetos de informação (IOD). Ele define um conjunto de operações e notificações genéricas que são chamadas de elementos de serviço de mensagem do DICOM (DIMSE) e a combinação do objeto de informação (IO) e tais serviços é chamada de SOP (senice–object pair). A classe SOP representa a unidade elementar de funcionalidade definida pelo DICOM. Especificando uma classe SOP para a qual uma implementação deve estar em conformidade, e a regra que um dispositivo em conformidade deve suportar, é possível definir um subconjunto preciso não ambíguo de funcionalidades do DICOM, incluindo os tipos de mensagens para serem trocadas, os dados transferidos dentro dessas mensagens, e o contexto semântico em que os dados devem ser entendidos.

O DICOM define formatos de objetos de informação (imagens, formas de onda, relatórios estruturados, etc.) e mecanismos para troca e armazenamento de informações. O domínio da aplicação do padrão DICOM é principalmente radiologia. Outras disciplinas que produzem imagens também utilizam DICOM: cardiologia, ultrasom, e medicina nuclear.

O padrão tem sido adotado pelos fornecedores de produtos de imagem: estações de trabalho de diagnóstico, PACS (Picture Archive and Communication Systems), arquivos e modalidades de imagem. O padrão tem evoluido, constantemente, para fornecer um conjunto abrangente de serviços para dar suporte completo ao fluxo de trabalho em radiologia, incluindo capacidade para integrar sistemas de imagem e modalidades de imagem com sistema de informação de radiologia (RIS): troca de informação do registro do paciente, dados de exame e resultados de diagnóstico.

2.5 Projeto GEHR (Good Eletronic Health Record)

O projeto GEHR foi oficializado pelo Programa de Desenvolvimento Tecnológico e de Pesquisas Telemáticas da União Européia, tendo como objetivo desenvolver uma estrutura de dados comum, abrangente e amplamente aplicável, que permita o compartilhamento de registros eletrônicos de saúde (Eletronic Health Record) dentro da Europa. A arquitetura do registro eletrônico foi resultado de um projeto de pesquisa que envolveu 21 organizações de 8 países entre 1992 e 1995.

O início do projeto foi dedicado à exploração e identificação dos requisitos dos usuários da área de saúde, classificados nas áreas de abrangência clínica, portabilidade, comunicação, aceitabilidade ética e legal, e educação. A questão chave do projeto GEHR era determinar o nível apropriado de padronização da estrutura da informação contida no registro e da troca da informação para uso clínico.

O grupo do projeto GEHR acredita que o principal propósito do registro eletrônico do paciente (EHR) é beneficiar o paciente, dando suporte aos cuidados clínicos presentes e futuros. Em contraste, a maioria dos sistemas da área de saúde se preocupam em otimizar a administração do serviço de saúde, o planejamento, a epidemologia, a pesquisa, não dando a devida importância ao bem estar do paciente. Além disso, o grupo concluiu que o que deve ser padronizado não é a prática clínica por si só, mas sim a semântica da informação documentada dentro do processo de fornecer cuidados clínicos.

Para a arquitetura ser útil, deve facilitar tanto a evolução dos sistemas existentes quanto a construção de novos. As idéias principais para arquitetura do GEHR, que visam atender os requisitos gerais dos usuários, foram desenvolvidas por médicos e cientistas da computação. A arquitetura tem sido refinada progressivamente através de testes de protótipos.

A arquitetura do GEHR fornece uma estrutura que pode suportar armazenamento e recuperação de uma grande diversidade de dados clínicos e promover comunicação requisitada pelos médicos. Além disso, o GEHR permite que sistemas possam ser construídos já suportando requisitos legais e éticos de uma boa prática médica. A arquitetura do GEHR tem sido formulada para conter os requisitos de saúde primária (serviços para prevenção de doenças) e secundária (cuidados com especialistas) para que possa ser usada pelos médicos, enfermeiras e profissionais da área de saúde. A necessidade de se trabalhar com uma grande variedade de objetos e memos, como, por exemplo, raio-x e imagens fotográficas, bio–sinais, desenhos clínicos, informações textuais, aumenta ainda mais a sua importância clínica porque ela permite utilizar qualquer tipo de dado.

O registro clínico deve evoluir gradualmente durante a vida do paciente e deve ser usado por profissionais de diversas disciplinas, trabalhando em diversos estabelecimentos e usando diferentes linguagens. Além disso, deve acomodar as necessidades de desenvolvimento de codificação e classificação de padrões e servir como guia clínico dentro do estabelecimento de saúde.

O projeto GEHR produziu um grande volume de documentação que está em domínio público [81], além de um modelo de informação orientado a objetos da arquitetura do GEHR chamado de GOM (GEHR Object Model).

Tentativas de desenvolver sistemas de prontuário eletrônico baseado na primeira versão do GEHR não tiveram sucesso por conta da sua complexidade e falta de roteiros de implementação. Em 1997, o GEHR foi recomendado pelo governo australiano, mas

como não havia uma implementação, o projeto também não obteve êxito. Isto levou à formação de um grupo chamado Ocean Informatics, o qual estendeu o modelo dando ênfase à implementação. Este trabalho foi feito com a colaboração de um grupo da universidade UCL (University College London).

Atualmente, existem dois grupos que trabalham com o GEHR, um do Reino Unido e o outro australiano. Os dois estenderam e implementaram o GEHR original. Porém, os australianos decidiram trabalhar com dois modelos: o GOM, que é o modelo de informação fundamental, e o modelo de arquétipos, o qual modela o conteúdo clínico. Nas Seções 2.5.3.1 e 2.5.4 estes modelos e o porquê de se trabalhar com os dois serão explicados.

Desde o início da década de 90, empresas de saúde têm mostrado interesse no GEHR. Dentre elas se encontram CPRI, OMG Health Domain Taskforce (CORBAMed), internet2 saúde, HL7, etc. Além disso, já existem aplicações baseadas no GEHR, como por exemplo, Black Sea TeleDiab (sistema de informação para diabéticos) [72], HDMP HEALTH.one (implementação comercial de uma aplicação GEHR) [71] e I-Med Clinical Networks (infra-estrutura de comunicação para sistemas de informação em saúde) [62].

2.5.1 Conceitos e Terminologias

Antes de apresentar o conceito de registro, é necessário compreender a visão de arquitetura. Para isso, introduzimos alguns conceitos básicos.

De acordo com Modell [59], dados são palavras e/ou números que possuem um significado, e informação são dados organizados para transmitir conhecimento. A estrutura que organiza esses dados para promover o uso da informação é chamada de arquitetura. Por exemplo, podemos dizer que a arquitetura de um documento é representada da seguinte forma:

```
Documento
    Capítulos
       Parágrafos
          Sentenças
             Palavras
```

Figura 2.2: Arquitetura de documentos

Registro de Saúde

O registro de saúde não é apenas o registro da anamnese[2] do paciente, mas todo acervo documental padronizado, organizado e conciso, referente ao registro dos cuidados médicos prestados, assim como aos documentos pertinentes a essa assistência. O registro inclui o exame clínico do paciente, suas fichas de ocorrências e de prescrição terapêutica, os relatórios da enfermagem, da anestesia e da cirurgia, a ficha do registro dos resultados de exames complementares e, até mesmo, cópias de solicitação e de resultado de exames complementares.

[2]Anamnese - entrevista que o médico ou terapeuta faz para saber mais sobre o paciente, podendo incluir ainda dados sobre gestação, parto, ambiente em que vive e, dependendo do médico e da linha seguida, pode durar uma hora. Esta anamnese servirá de base para o médico.

Cada paciente tem um único registro de saúde, o qual contém a história da sua saúde. As informações encontradas no registro podem ser usadas para auxiliar os médicos durante os cuidados clínicos do paciente, responsabilizar os mesmos com relação às questões éticas e legais, pesquisar e fornecer dados estatísticos, ensinar, etc. Um registro de saúde é dito ser eletrônico quando é manipulado e armazenado por meio eletrônico. As principais operações feitas com o registro eletrônico de saúde são entrada e armazenamento de dados, acesso e recuperação de dados, edição e transferência de dados.

Existe uma questão bem interessante: A quem pertence o registro? Antes pensava-se que ele pertencia ao médico assistente ou à instituição para a qual ele prestava seus serviços. Mesmo sendo o médico, indubitavelmente, o autor intelectual do dossiê por ele recolhido, é claro que este documento é de propriedade do paciente. Mas, o médico e a instituição têm o direito de guarda.

Os médicos normalmente trabalham em cooperação, então o registro tem um papel fundamental na comunicação do conhecimento. Além disso, é uma importante ferramenta que proporciona qualidade no cuidado do paciente, porque fornece informações essenciais para diagnóstico e tratamentos mais eficientes.

Em suma, o registro do paciente é um depósito de informações documentadas que tem como principal propósito fornecer suporte aos cuidados do paciente e a comunicação entre os profissionais da área de saúde.

Arquitetura do Registro Eletrônico de Saúde

Uma arquitetura de registro eletrônico é um modelo de informação ou *framework* para construção de registros eletrônicos. A arquitetura do GEHR foi definida pelo comitê de padronização europeu CEN (Comunité Européen de Normalization) como um modelo de informação conceitual de um registro de saúde eletrônico [52].

A arquitetura não estabelece ou dita o que deve ser armazenado dentro dos registros de saúde, e nem como os sistemas de prontuário eletrônico devem ser implementados. Ela é um modelo de características genéricas necessárias para que o registro de saúde possa ser comunicável, completo, ético-legal e útil, e para que ainda possa manter a integridade entre sistemas.

A arquitetura do registro de saúde se preocupa não somente com os dados, mas com a maneira com que os dados estão organizados para transmitir seu significado completo. Ela não faz nenhuma restrição com relação aos tipos de dados que podem aparecer dentro do registro. A restrição pode ser imposta pelos bancos de dados, que podem não trabalhar com algum tipo de dado específico. Devemos deixar claro que os termos arquitetura e *framework* utilizados acima não têm o mesmo significado na Engenharia de Software.

2.5.2 Requisitos dos Usuários

Sistemas clínicos podem ser projetados para determinados usuários de saúde, mas uma arquitetura de registro eletrônico deve atender as necessidades de *todos* os profissionais da área de saúde. O formalismo da arquitetura do GEHR foi desenvolvido seguindo uma investigação extensiva dos requisitos dos usuários. Médicos, enfermeiras e outros

profissionais de saúde foram envolvidos na definição dos requisitos que estão organizados em várias áreas chaves, conforme [42]:

1. Requisitos para um registro amplo de consultas realizadas por profissionais da área de saúde primária e secundária.

   Incluem necessidades de se trabalhar com uma variedade de tipos de dados multimídia, de acessar um conjunto de termos codificados e de acomodar entradas estruturadas e texto livre. O registro é uma coleção de diversas observações e estruturas de dados, construídas por várias pessoas de vários lugares durante a toda a vida do paciente.

2. Requisitos para a portabilidade de registros de saúde entre sistemas institucionais diferentes, independentemente de configurações de hardware, de sistemas operacionais ou de aplicações daqueles específicos, e independente da língua origem e do conjunto de termos usados.

   Um registro eletrônico precisa estar disponível em diversas plataformas, tanto as atuais quanto as do futuro. O projeto GEHR em vez de propor um formato de troca específico, propõe um modelo formal de informação, através do qual se pode construir a camada de negócio dos sistemas de prontuário eletrônico.

3. Requisitos para comunicação dos registros de saúde entre médicos envolvidos no cuidado do paciente, podendo ser através de redes de telecomunicação ou via dispositivos conectados intermitentemente tais como *smart cards*.

   Pacientes mudam-se de um lugar para o outro e consultam-se com diversos médicos. Resultados de testes laboratoriais vindo dos laboratórios, raio-x dos radiologistas e outros dados, todos necessitam estar contidos no registro. Médicos e pesquisadores necessitam de informações anônimas sobre pacientes e estudos baseados em população. O registro do paciente deve permitir e incentivar a comunicação.

4. Requisitos relacionados à questões de ética, segurança e médico–legal.

   Estes aspectos assumem maior importância quando utiliza-se registros eletrônicos como meio para registrar e armazenar informações relativas ao paciente. A informação dentro do registro deve ser fielmente registrada como pretendida pelo médico. Deve ser à prova de alterações e emendas, mas deve permitir alguma forma de alterar erros inocentes. Para que exista a responsabilidade médica pela entrada de informações, é necessário que se tenha rastreabilidade dessas informações.

5. Requisitos relacionados às necessidades educacionais no nível de pré–registro e pós–registro para habilitar os profissionais no uso das novas tecnologias.

   As informações que são inseridas no registro do paciente pelos estudantes devem ser diferenciadas daquelas inseridas pelos médicos. Quando o estudante inclui uma informação, esta é considerada pré-registro. O médico após avaliá-la, pode passá-la para pós-registro, tornando-a válida para o acompanhamento clínico do paciente, ou pode deixá-la como pré-registro e servir apenas como forma de avaliar o aluno. O médico se responsabiliza pela informação quando passa de pré para pós-registro.

Não é possível atender todos os requisitos dos usuários apenas com a padronização do formato dos dados porque eles podem ser trocados facilmente. Uma maneira de atender esses requisitos é através da padronização da arquitetura do registro eletrônico (EHR).

2.5.3 Descrição da Arquitetura do GEHR

O grupo do projeto GEHR se preocupou em propor uma arquitetura genérica, flexível e tão não prescritiva quanto possível. Porém, onde os médicos identificaram a necessidade, a arquitetura incorpora características que permitem ser utilizadas para prescrição. Os principais componentes arquiteturais do GEHR, de acordo com [42], são:

- Registro Eletrônico (EHCR): Fornece um depósito para todos os dados de um paciente. Cada paciente só pode ter um registro em cada sistema de prontuário. O registro é formado por uma coleção de transações agrupadas de acordo com determinados critérios.

- Transação (Transaction): "informação registrada sobre um paciente por um autor em uma única instituição em um determinado momento" [41]. Ela representa a entrada de dados de uma sessão interativa com o registro do paciente. Uma sessão interativa pode ser uma consulta, um preenchimento de resultado de exames ou de uma carta de segunda opinião, etc.

  As transações são autorizadas pelos médicos que se responsabilizam pela exatidão da informação. Elas devem estar sempre disponíveis para minuciosas avaliações médico-legal, mesmo que tenham sofrido alterações. Toda informação dentro do registro de saúde deve ser atribuída a transações, cada uma identificada unicamente pela data e hora, instituição e médico responsável pela sua criação.

  Uma transação é formada por cabeçalhos, que contêm coleções de itens de registros, que por sua vez, contêm itens de registros e coleções de itens de registros.

- Item de Registro (Health Record Item): Como dados inseridos no registro podem ter diversos formatos, é importante definir uma unidade elementar de entrada de dados. Esse conceito de menor unidade de informação que se pode entrar dentro do registro é conhecida como item de registro de saúde. Ele fornece o mecanismo para expressar o valor do conteúdo das entradas realizadas no registro do paciente. É obtido como resultado de uma medida específica, observação, etc. O item é composto, principalmente, de um identificador (nome) e um valor de conteúdo principal. Porém, ele pode ter atributos. Por exemplo:

```
nome:peso - (identificador)
valor:60 - (valor do conteúdo)
unidade:Kg - (atributo)
```

Figura 2.3: Item de registro

- Coleção de Itens de Registro (HRI Collection): Também conhecida como observação, pode conter outras coleções de itens e itens de registro. Permite a construção de agregações de dados complexas. É composta por um identificador (nome) e observações (itens de registro ou coleções de itens de registro). Por exemplo:

```
coleção:  distenção muscular
   nome:  localização     valor:  abdômen
   nome:  tempo           valor:  22/02/2002
```

Figura 2.4: Coleção de itens de registro

- Cabeçalho (Heading): Fornece o significado do agrupamento das coleções de itens e de itens.

Esses componentes são compatíveis com as estruturas que aparecem nos registros de saúde existentes e preenchem os requisitos dos usuários identificados através do projeto GEHR (Seção 2.5.2).

A Figura 2.5 é um exemplo de utilização dos componentes arquiteturais Cabeçalho (Heading), Coleções de Itens de Registro de Saúde (HRI Collection) e Itens de Registro de Saúde (HRI).

```
cabeçalho:  motivo Consulta
   coleção:  dor
     nome :localização     valor:cabeça
     nome :duração         valor:2     unidade:  dias
   coleção:vômito
     nome :tempo           valor:02/10/2002
     nome :freqüência      valor:3X
```

Figura 2.5: Exemplo de cabeçalho e coleção de itens de registro

Os construtores básicos da arquitetura do GEHR, HRIs e HRI Collections, permitem que a documentação dos conceitos clínicos possa ser organizada utilizando uma hierarquia. Além disso, permitem acomodar tipos de dados multimídia, termos codificados, texto, quantidade, etc. O cabeçalho fornece um construtor adicional para comentários dessas hierarquias de conceitos dentro do registro.

Todas as entradas no registro de saúde são gerenciadas pelo componente *Transaction*. Existem vários tipos de transação e cada uma delas encapsula informações da língua falada e de auditoria (médico-legal) relacionadas ao conjunto de entradas. O nome do item de registro, o nome da coleção de itens de registro e o nome do cabeçalho permitem que entradas clínicas possam ser identificadas para pesquisa e análise através de um sistema de prontuário eletrônico e para troca de informações entre sistemas.

2.5.3.1 GOM - GEHR Object Model

A partir dos componentes arquiteturais do projeto GEHR original apresentados na Seção 2.5.3, foi criado, inicialmente, um modelo GOM versão 1.0 para a arquitetura do registro do paciente. Esse modelo foi definido utilizando a notação gráfica BON (Business Object Notation) [50], a qual é usada para especificar e descrever sistemas orientados a objetos. BON é um método de análise e projeto que oferece um mapeamento direto para uma implementação Eiffel. A linguagem Eiffel [56] foi escolhida para implementação porque atendia diversos requisitos que outras linguagens não possuiam, como por exemplo, pré e pós condições, invariante de classe, etc.

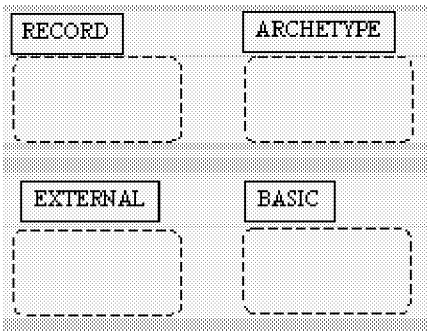


Figura 2.6: Pacotes de nível topo do GOM

Posteriormente, a arquitetura do GEHR foi estendida e é utilizada como estudo de caso nesta dissertação. O GOM da arquitetura consiste de 4 pacotes como mostra a Figura 2.6. A seguir, estes pacotes são descritos.

RECORD

Este pacote modela o registro eletrônico de saúde. Ele é divido em cinco níveis: EHR, TRANSACTION, ORGANISER, CONTENT e DATA (Figura 2.7). Os primeiros dois níveis podem ser considerados níveis de gerenciamento de informação, porque são responsáveis pela criação e recuperação de informação no registro de saúde e transferência de informação entre registros de saúde. A classe EHR_CONTENT, a qual se encontra dentro do pacote TRANSACTION, é o ponto inicial do que se pode dizer "conteúdo" do registro. Sua estrutura é definida pelos pacotes ORGANISER e CONTENT.

Como podemos observar, através da Figura 2.7, o RECORD é constuído dos seguintes pacotes:

- EHR: Modela os conceitos relacionados ao próprio registro eletrônico. Cada registro de saúde eletrônico (EHCR) representa os cuidados da saúde de um paciente no formato eletrônico. É possível encontrar, simultaneamente, várias instâncias do registro de saúde eletrônico em locais distintos, porque o paciente pode receber cuidados em diversas instalações de saúde. O registro lógico, também conhecido como registro virtual de um paciente, é a junção de todas as instâncias do seu registro de saúde (EHCR). A classe EHR_EXTRACT tem a mesma estrutura que o EHR, porém possui apenas as transações que se deseja importar ou exportar de ou para outro local.

  A classe EHR_BASE agrupa as características comuns encontradas nas classes EHR e EHR_EXTRACT. Ela consiste de 4 grupos de transações versionadas (VERSIONED_TRANSACTION): *subject_transaction*, *demographic_entities*, *persistent_clinical_transactions*, *event_clinical_transactions*, como mostra a Figura 2.8. O primeiro grupo possui uma única transação, que contém informação de identificação do paciente. O segundo grupo possui transações de entidades demográficas, representando médicos, instalações de saúde, etc. O terceiro grupo possui transações persistentes, válidas durante toda a vida do paciente. O último grupo possui transações eventuais, as quais são válidas em um determinado período.

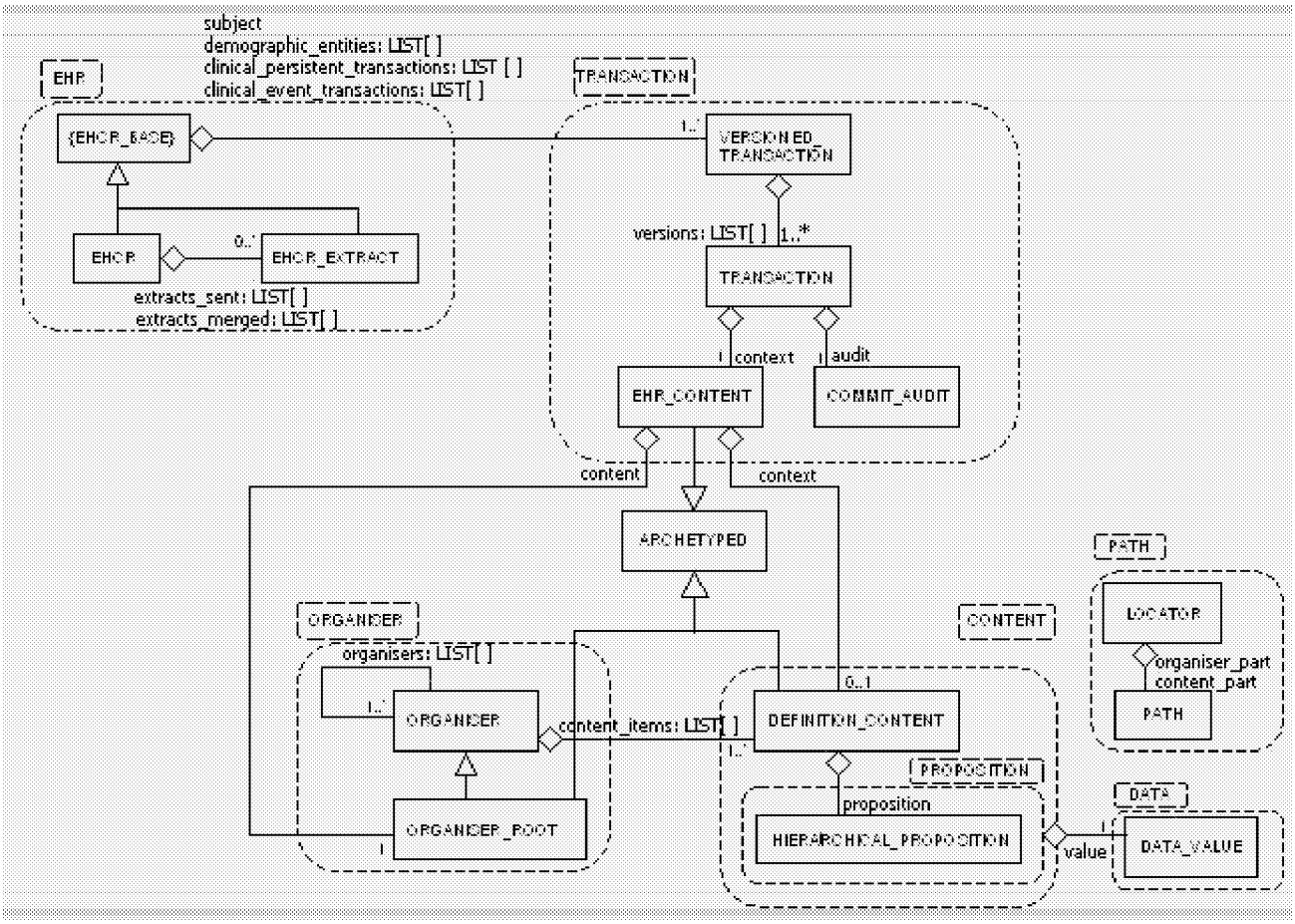


Figura 2.7: Pacote RECORD

- TRANSACTION: Modela as transações que ocorrem no registro. Uma transação versionada é definida pelas classes VERSIONED_TRANSACTION e TRANSACTION. A classe TRANSACTION contém o atributo *content* do tipo EHR_CONTENT, o qual possui a informação do paciente, e o atributo *context* do tipo DEFINITION_CONTENT que possui o contexto da transação. Uma interação do médico com o registro de saúde (EHR) num determinado tempo corresponde a uma VERSIONED_TRANSACTION. Mesmo tendo vários médicos envolvidos na criação do registro, apenas um é o responsável pela informação inserida nele.

  A classe COMMIT_AUDIT modela a responsabilidade do médico. Existem dois tipos de alteração no registro do paciente, a primeira é quando de fato se cria a transação que acontece através de um contato do médico com o paciente, entrada de resultado de exames, etc. A segunda forma é através da criação de uma nova versão para a transação existente como forma de corrigir erros ou omissão de informação.

- ORGANISER: É composto pelas classes ORGANISER_ROOT e ORGANISER. O seu propósito é fornecer uma estrutura navegacional dentro do registro de saúde, similar ao conceito de "diretórios" nos sistemas de arquivo, e aos cabeçalhos no registro de papel. Os organizadores podem conter recursivamente outros organizadores, os quais finalmente contêm os conteúdos, permitindo uma extensa estrutura de na-

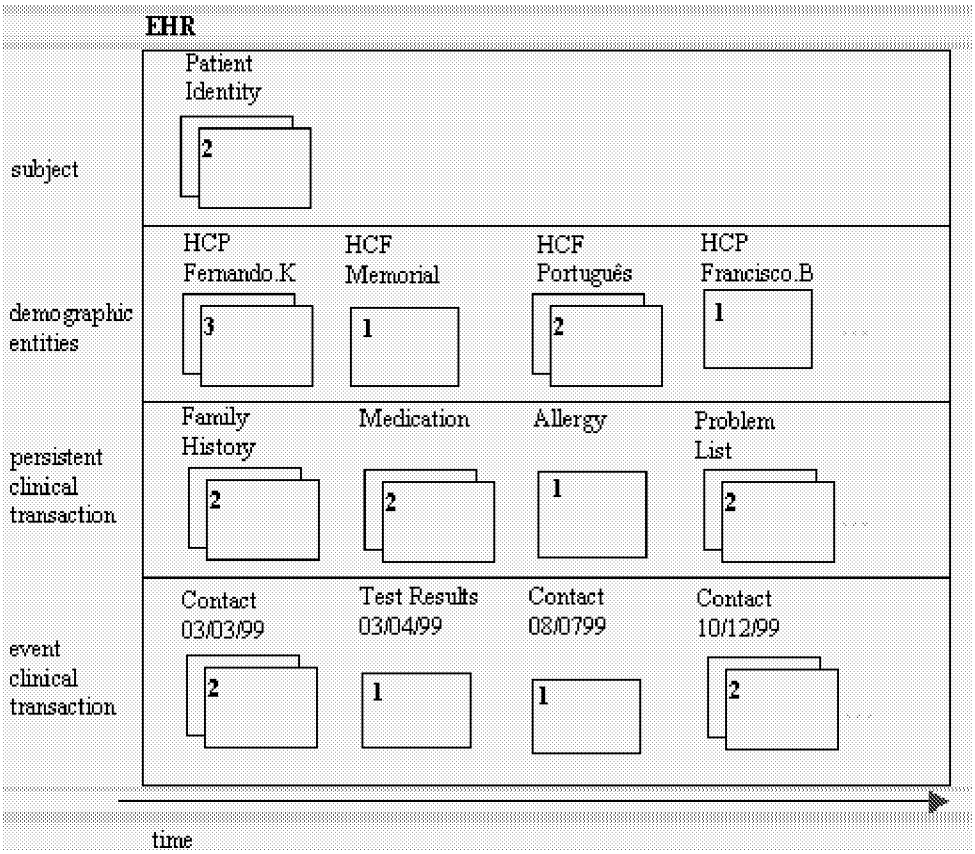


Figura 2.8: Registro do paciente

vegação.

- CONTENT: É formado pela classe DEFINITION_CONTENT e pelo pacote PROPOSITION. Ele é responsável por modelar os tipos de conhecimento, como por exemplo, uma observação (resultado de algum exame), uma segunda opinião, alguma instrução (procedimento que o paciente deve seguir), etc. Além disso, define o formato da estrutura do conteúdo do conhecimento, o qual pode ser tabela, lista, valor simples, séries de tempo, matriz, etc. A Figura 2.9 [10] ilustra apenas alguns dos possíveis formatos de representação do conteúdo do conhecimento.

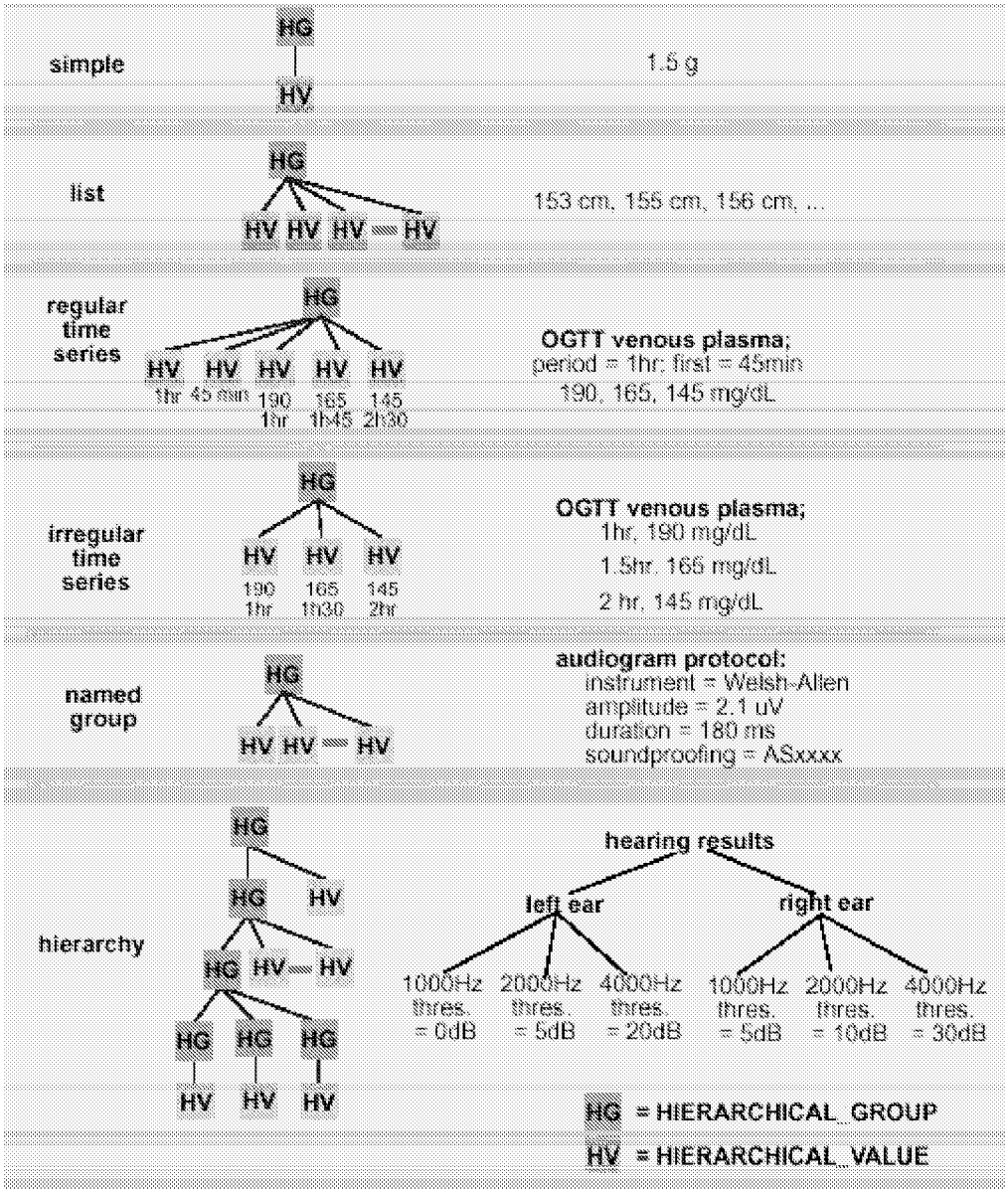


Figura 2.9: Exemplos de representação de dados

As estruturas de representação do conteúdo do conhecimento são formadas pelos elementos arquiteturais coleções de itens de registros (HIERARCHICAL_GROUP) e itens de registros (HIERARCHICAL_VALUE). Por exemplo, o protocolo de audiograma está estruturado da seguinte forma:

- Nome da coleção de itens de registro (HG) é protocolo de audiograma
- Nomes e valores dos Itens de registro (HV) são:
  (nome = instrumento, valor = Welsh-Allen),
  (nome = amplitude, valor = 2.1 uV),
  (nome = duração, valor = 180ms),
  (nome = teste de som, valor = ASxxxx)

A classe dentro do pacote PROPOSITION é responsável pela definição da estrutura do conteúdo. Esse pacote tem como objetivo fornecer uma diversidade de interfaces para cada formato de representação do conteúdo, como, por exemplo, para listas, tabelas, matrizes, valores simples, etc.

- DATA: Define tipos de dados através dos quais valores são codificados. Além dos tipos usuais, como por exemplo INTEGER, STRING, etc, há tipos como TERM_TEXT, MULTIMEDIA, QUANTITY, etc. que são próprios do modelo GEHR.

- PATH: Um aspecto importante do modelo GEHR é o *locator* ou *path*. Este é um conceito similar ao "caminho" ou URL de sistemas de arquivo. Ele permite que qualquer item do registro seja facilmente encontrado a partir de um texto fornecido.

Como a especificação está em um nível abstrato, as questões de busca de um item no registro via *path* ou *locator* não estão sendo tratadas na especificação.

ARCHETYPE

Este pacote é responsável pela definição do conteúdo clínico. A Seção 2.5.4 apresenta detalhadamente este assunto.

BASIC

Este pacote é responsável pela definição dos tipos básicos, incluindo tipos enumerados.

EXTERNAL

Algumas classes são definidas para permitir que o conteúdo do registro possa referenciar entidades fora do registro. O principal exemplo é a URI (Unique Resource Identifier) definida pelo consórcio W3C (World Wide Web Consortium) [86].

2.5.4 Arquétipos

Um conceito pode ser descrito de várias maneiras. Por exemplo, pressão sanguínea, pressão arterial, pressão sistólica e diastólica são variações do mesmo conceito. Uma questão importante é saber qual a variedade permitida, antes que duas descrições de um mesmo conceito sejam interpretadas como conceitos distintos.

Na arquitetura GEHR, este problema é resolvido através de um arquétipo para cada conceito. Todo arquétipo possui um conjunto de restrições estruturais, que combinadas nas especializações do arquétipo, definem um conjunto de variações do mesmo conceito. Cada especialização é uma variação do conceito principal, o qual é definido pelo arquétipo pai. A Figura 2.10 ilustra as variações do conceito principal.

Para exemplificar o mecanismo de especializações no modelo de arquétipos, considere a Figura 2.11 [9], a qual mostra o arquétipo que define o conceito "pressão sanguínea". O item protocolo é opcional e, no máximo, só pode haver um protocolo. Dentro da

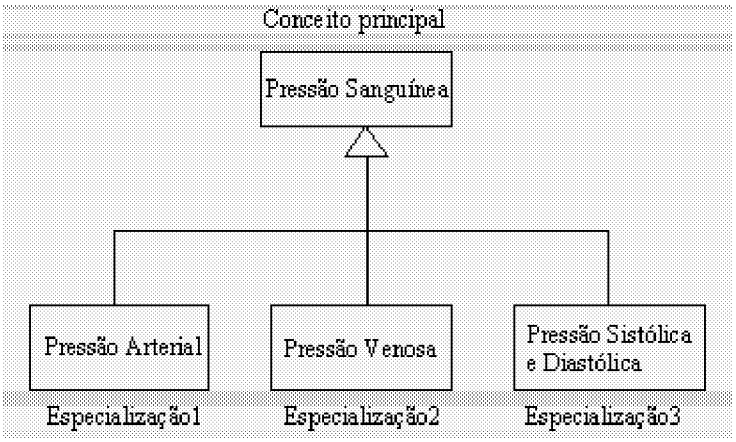


Figura 2.10: Variações de um mesmo conceito

estrutura do protocolo é permitido adicionar novos itens. Existem itens na estrutura que são obrigatórios, como os dados principais, sistólica e diastólica. A maneira de representar as restrições estruturais dentro do modelo de arquétipos é através do uso de cardinalidade. A partir deste arquétipo, poderíamos, por exemplo, especializá-lo para criar o conceito "pressão arterial sistólica e diastólica", o qual possui todos os itens obrigatórios do arquétipo pai e dos opcionais, o item protocolo passa a ser obrigatório. Então, quando um médico estiver se referindo ao conceito "pressão arterial sistólica e diastólica", na verdade ele está falando de uma versão do conceito "pressão sanguínea". A Seção 2.5.4.2 explora este assunto em maiores detalhes.

É importante mencionar que a intenção não é de se chegar a uma única definição, perfeita, de cada conceito geral. Por exemplo, não há necessidade de se ter apenas pressão sanguínea; pode haver também pressão venosa, pressão arterial, pressão sistólica e diastólica, etc. O importante é que para cada conceito que o domínio deseja utilizar, uma definição pode ser desenvolvida em termos de restrições estruturais.

De acordo com Thomas [8], os arquétipos possuem os seguintes propósitos:

- Definição dos conceitos de **conhecimento clínico** independente de sistema e tecnologia, pelos usuários do domínio clínico.
- Garantia da **qualidade da informação** do EHR, através da garantia de que o conteúdo do registro está sempre em conformidade com as restrições expressas pelos arquétipos.
- Possibilidade da **interoperabilidade** de sistema no nível de conhecimento, através do uso global dos arquétipos especializados e padronizados.

Os maiores benefícios dos arquétipos são:

- Permite aos usuários expressar seus conceitos formalmente dentro de um domínio. Arquétipos podem ser desenvolvidos e gerenciados por grupos de usuários, grupos de trabalho em padronização, ou algum outro fórum disponível, sem que haja a necessidade de referenciar sistemas que irão processar esses arquétipos.

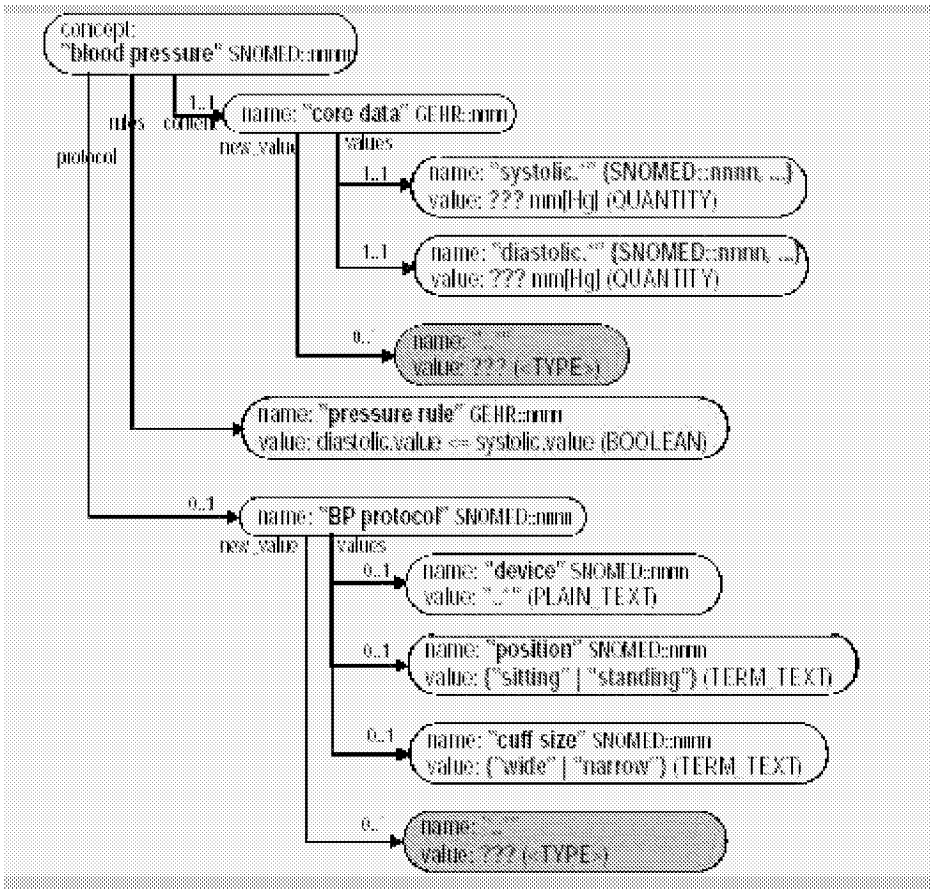


Figura 2.11: Um possível conceito de pressão sanguínea

- Habilita sistemas de informações a guiar e validar entradas dos usuários durante a criação e modificação da informação em tempo de execução, garantindo que todas as instâncias estão em conformidade com os requisitos do domínio.
- Garante interoperabilidade no nível de conhecimento, porque sistemas que se comunicam podem assumir as mesmas definições de arquétipos.
- Assegura que EHRs são a prova de tempo, isto é, que todos os EHRs criados através de software baseado em GEHR permanecem viáveis, apesar das mudanças nas estruturas clínicas ou conceitos.
- Permite que software seja a prova de tempo, porque os conceitos clínicos não são diretamente codificados em modelos de softwares.
- Fornece uma base bem definida para consultas eficientes de dados complexos.

O sistema de gerenciamento de arquétipos assegura que apenas uma única representação do GEHR é usada para um determinado conceito clínico. Desta forma, é garantido que partes do registro de saúde que foram transferidas serão entendidas da mesma maneira, e que sistemas de software, particularmente de apoio à decisão, podem fazer deduções sobre as estruturas.

Os arquétipos são obrigatórios para as classes que definem a estrutura do contéudo do registro, ou seja, para EHR_CONTENT, ORGANISER_ROOT e DEFINITION_CONTENT.

2.5.4.1 Ontologias

O termo ontologia refere-se à formalização do conhecimento de um domínio. Embora não exista uma classificação padrão do conhecimento, uma separação em níveis ontológicos é freqüentemente utilizada. A Tabela 2.1 apresenta os níveis ontológicos para o domínio de saúde.

No nível 0 estão os conceitos encontrados nos livros textos de medicina, vocabulários técnicos, e cursos de treinamento. Medicina é um dos poucos domínios que possuem conceitos básicos computáveis na forma de vocabulários estruturados, tais como SNOMED [44], ICD10 [43], etc.

No nível 1 estão os conceitos formados pela composição dos conceitos do nível 0. Eles são divididos em dois sub-níveis: o primeiro é chamado de conteúdo ubíquo, porque é entendido da mesma maneira por qualquer pessoa, como, por exemplo, índice de massa corporal, pressão sanguínea, etc. O segundo é chamado de conteúdo baseado em caso de uso, porque leva em consideração processos específicos que ocorrem de acordo com cenários particulares. Por exemplo, o conceito "reação adversa", o qual registra alguma reação contrária a algum medicamento ou outra substância, leva em consideração o meio de prescrição do medicamento ou substância. Este tipo de conceito é normalmente entendido diferentemente pelos usuários de domínios distintos.

No nível 2 estão os conceitos criados pelos usuários do domínio, para organizar os diversos itens de informação obtidos durante o encontro com o paciente. Exemplos incluem problemas, medicamentos, exercícios físicos, história familiar, etc.

No nível 3 estão os conceitos relacionados ao "empacotamento" lógico da informação (transação). Toda informação inserida no registro de saúde é armazenada através de transações, e cada uma delas está relacionada a um conceito específico. Por exemplo, a transação de conceito história familiar contém as doenças relevantes dos familiares do paciente. O EHR é formado pelas transações.

No nível 4 estão os conceitos formados pela seleção e "empacotamento" de informações para permitir comunicação com outros usuários. Tipicamente, estes conceitos são documentos, relatórios e *extracts*. O projeto do registro de saúde eletrônico do GEHR possui o conceito "registro de extração", o qual define o pacote de informação a ser enviado para outros sistemas.

nível	**significado**	**exemplos**
princípio (0)	vocabulários, fatos considerados sempre verdadeiros	livros textos, SNOMED, Read, ICPC,ICD10
conteúdo ubíquo (1a)	conceitos independentes de contexto, que possuem um mesmo significado para todos os usuários	pressão sanguínea, solicitação de medicamento, hemograma
conteúdo baseado em casos de uso (1b)	conceitos dependentes de contexto, definidos de acordo com os casos de uso	história de um membro da familia, reação adversa
organizacional (2)	conceitos de informação estrutural cujo objetivo é organizar informações	exercícios físicos, vacinações, problemas
armazenamento (3)	conceitos relativos a estrutura da informação armazenada	transações para lista de problemas, história da vacinação, estilo de vida
comunicação (4)	conceitos relativos ao empacotamento da informação a ser compartilhada	EHR extract, relatório, documento

Tabela 2.1: Classificação do conhecimento para o domínio da saúde

A composição de arquétipos é um mecanismo que permite construir arquétipos de níveis ontológicos mais altos a partir de arquétipos de níveis mais baixos. Porém, apenas algumas combinações são permitidas. A Figura 2.12 [9] ilustra um exemplo de composição de arquétipos em níveis. O conceito "assunto da história familiar" do nível de conteúdo só pode ser usado sob o conceito "história familiar" do nível organizacional. Não faz nenhum sentido, por exemplo, permitir colocar o arquétipo "hemograma" do nível de conteúdo sob o arquétipo "história familiar" do nível organizacional, porque eles não têm relação.

A composição pode ser combinações de arquétipos em dois níveis, isto é, armazenamento + organizacional, organizacional + conteúdo ou em três níveis, isto é, armazenamento + organizacional + conteúdo. Ela reduz bastante o número de arquétipos necessários para produzir uma grande quantidade de conceitos, porque um mesmo arquétipo pode participar de vários arquétipos compostos.

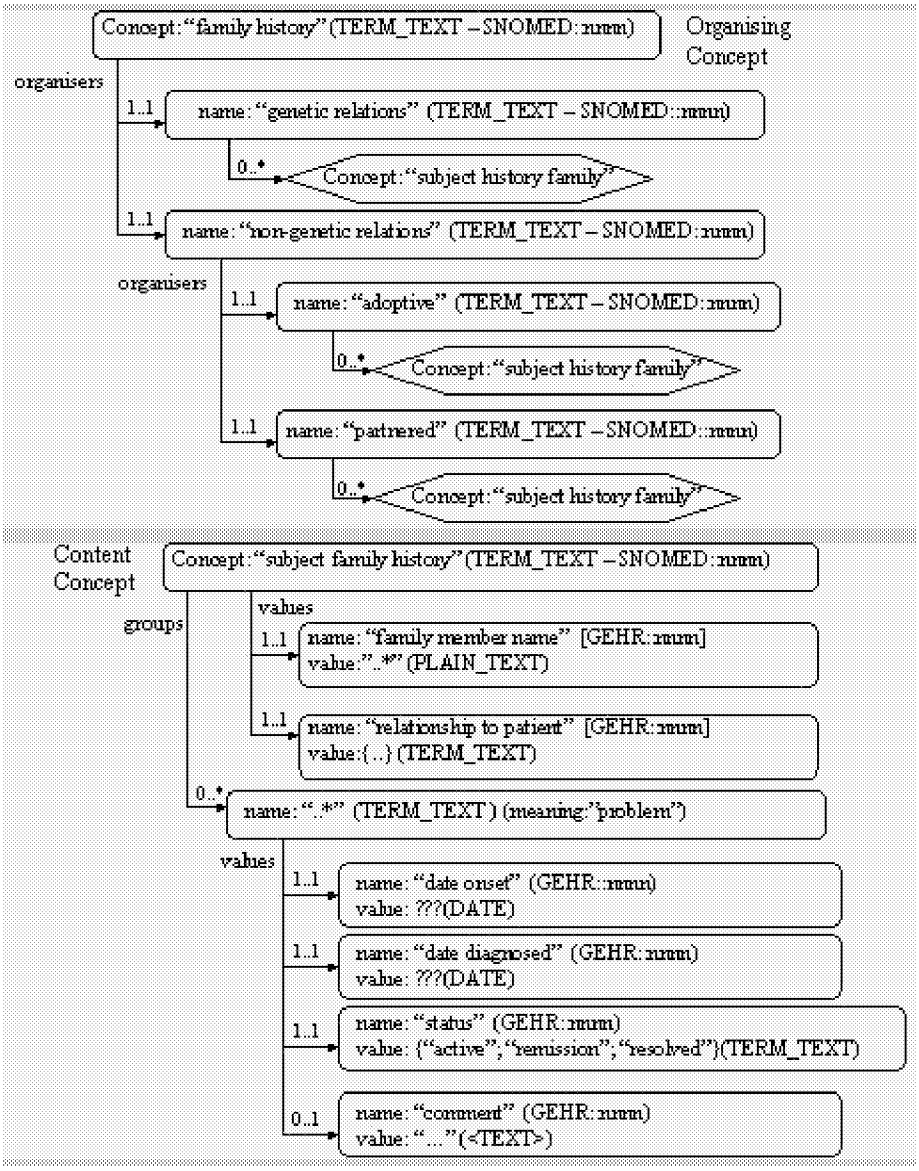


Figura 2.12: Arquétipo composto de história familiar

2.5.4.2 Especialização de Arquétipos

Especialização deve ocorrer quando é necessário alterar os conceitos considerados padrão. Normalmente os conceitos que estão relacionados de alguma maneira às normas legislativas, quase sempre são diferentes entre países, mesmo quando os profissionais da área de saúde pensam neles como sendo a mesma coisa. Por exemplo, o conceito "receita médica" varia de acordo com as normas do país.

Especialização deve ocorrer também devido à especialização do profissional. Por exemplo, as observações feitas por um clínico geral para o exame ONG (olhos, nariz e garganta) são mais básicas que as feitas por um otorrinolaringologista. O motivo mais forte para o uso de especialização é permitir que os profissionais trabalhem no seu contexto, através de arquétipos nacionais que são especializações de arquétipos padrão, e compartilhar informação internacionalmente através do arquétipo pai comum, desconsiderando detalhes irrelevantes.

2.5.4.3 Relacionamento entre Modelos de Referência e Arquétipos

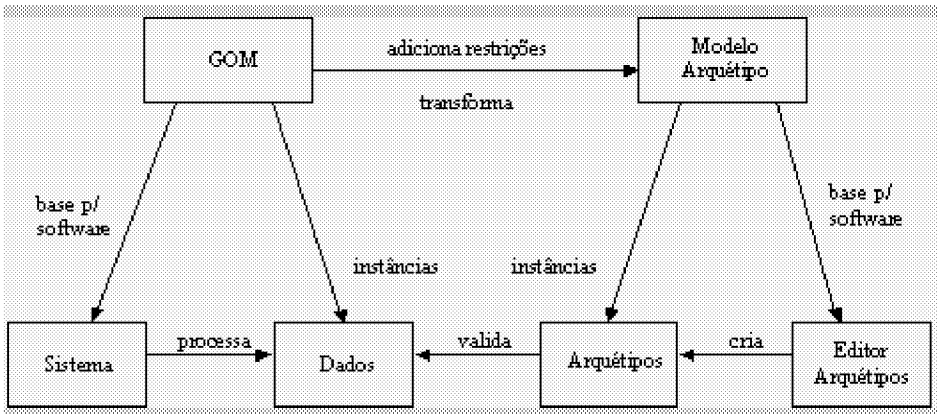


Figura 2.13: Relacionamento entre modelos GOM e arquétipos

A Figura 2.13 ilustra o relacionamento entre o GOM (modelo de referência), o modelo de arquétipos, e suas respectivas instâncias. O modelo de arquétipos é construído a partir do modelo GOM adicionando restrições. As instâncias do modelo de arquétipos são os próprios arquétipos, os quais são modelos de conceitos específicos. Os dados são instâncias do GOM que respeitam as restrições expressas pelos arquétipos.

2.6 Comparativo entre as abordagens

Para melhor compreender a escolha do GEHR como estudo de caso, a comparação entre as diversas abordagens é resumida através da Tabela 2.2.

Esses padrões sofrem da falta de modelos para expressar conceitos clínicos. O mesmo problema acontece com o GEHR original e com outros modelos similares. Normalmente, estes padrões incluem conceitos de informação e de domínio clínico no mesmo modelo, tornando-o muito complexo, não gerenciável e muito difícil de implementá–lo. Por

Padrões	GEHR(novo)	HL7 v3	CORBAMed	DICOM
Modelos de conceitos clínicos	X	-	-	-
Modelos de informação	X	X	-	X
Imagens de diagnóstico	-	-	qualidade inf.	qualidade sup.
Implementações	X	X	-	X
Escopo	paciente	total	total	radiologia
Trilhas de auditoria	X	-	-	-
Formalização	X(BON)	X(redes Petri)	-	-
Custo de definição	mais caro	mais barato	intermediário	intermediário
Custo de implementação	mais barato	mais caro	intermediário	intermediário

Tabela 2.2: Comparação entre as diferentes abordagens

exemplo, no modelo de informação do HL7 (RIM), encontramos as classes *Substance administration* e *Diet* que modelam conceitos clínicos de dieta alimentar e administração de medicamentos, respectivamente.

O GEHR australiano (novo) criou um modelo de dois níveis para resolver esse problema. O conteúdo clínico está especificado e padronizado, separadamente do modelo de informação GOM, em estruturas chamadas de arquétipos. Então, os conceitos citados acima são definidos através de arquétipos (modelos de conceitos clínicos). O HL7 tem demonstrado interesse em se trabalhar com modelos de conceitos na forma de *templates* ou arquétipos no estilo do GEHR. O CORBAMed não tem interesse em definir explicitamente modelos de conceitos clínicos e nem de informação . O DICOM v3 trabalha com modelo de informação, mas não demonstrou ainda interesse em trabalhar com *templates* ou arquétipos.

A qualidade da imagem de diagnóstico, como, por exemplo, radiologia, cardiologia, etc., do DICOM é bem superior à do CORBAMed, porque quase todos os seus serviços estão direcionados às imagens. Os outros protocolos, GEHR e HL7 não tratam imagens, mas possuem interfaces que permitem utilizar o padrão DICOM.

O protocolo HL7 descreve um padrão para as mensagens; a versão 2.3 do padrão se encontra totalmente implementada e está sendo utilizada com sucesso dentro dos Estados Unidos e em alguns países como Austrália e Nova Zelândia. A principal vantagem do HL7 (v2.x) é que já foi implementado e bastante testado. O mesmo não acontece com a v3.0, porque nem se quer foi finalizada a sua especificação. O GEHR possui um kernel implementado, o qual foi desenvolvido diretamente do GOM. O seu propósito é ser capaz de construir estruturas de registro de saúde e processá–las de acordo com a semântica do GOM. No entanto, o kernel não pode ser bastante testado, porque faz pouco tempo que foi implementado. O DICOM foi implementado e bastante testado pelos fornecedores de equipamentos de imagens. O CORBAMed não tem interesse em implementações, apenas em definir as interfaces padrão.

Mesmo que o CORBAMed esteja procurando definir interfaces para interoperabilidade, o seu escopo ainda é maior que o do GEHR, porque cobre todos os tipos de requisitos de informação hospitalar, incluindo administração, cobrança, alocação de recurso e gerenciamento de pedidos. Porém, seus documentos principais têm sido influenciados pelo projeto GEHR original. Estes são:

- PIDS - the Person(patient) IDentification Service
- COAS - the Clinical Observation Access Service
- TQS - the terminology query service

O *kernel* do GEHR tem sido uma implementação bem próxima do COAS, porque as definições das interfaces do COAS são bem próximas ao GOM. O HL7 também possui um escopo bastante abrangente, porque atende tanto a parte clínica quanto a administrativa. Já o DICOM é restrito as especialidades que trabalham com imagens, como radiologia, cardiologia, etc.

O GEHR permite fazer auditoria médica, ou seja, rastrear todas as entradas realizadas por um médico no registro de um paciente. Isto é importante porque o médico se responsabiliza pelas informações e desta maneira pode avaliar o seu trabalho e tratar as questões médico-legal. Os outros protocolos não permitem, porque não são orientados a registro.

O GEHR foi formalizado utilizando a notação BON, a qual é semi-formal. Existe um interesse em especificar o GEHR formalmente utilizando uma linguagem formal como Z++, Object-Z ou B [10]. Em contanto com o grupo australiano responsável pelo projeto GEHR, surgiu a idéia de formalizá-lo em CSP-OZ [24], o que constituiu a motivação inicial para esta dissertação. Podemos encontrar trabalhos na formalização do HL7 utilizando redes de Petri [76]. O CORBAMed e DICOM não mostram interesse em utilizar métodos formais.

O grupo do projeto GEHR apresentou um comparativo [80] do investimento de projeto da arquitetura do registro eletrônico (GEHR), da arquitetura de interface comum (CORBAMed) e da padronização de mensagem (HL7) versus custo de implementacação de cada um conforme a Figura 2.14. De acordo com o gráfico apresentado, verifica-se que o custo de projeto de arquitetura de registro é mais caro do que de arquitetura de interface comum e de mensagens. Em compensação, uma vez padronizada, a sua implementação é mais barata.

2.7 Considerações Finais

De acordo com o comparativo entre as diversas abordagens, realizado na Seção 2.6, podemos observar que o padrão GEHR atende quase todos os itens. Apesar de não tratar o item imagem de diagnóstico, ele permite utilizar outro padrão, como, por exemplo o DICOM. Desta forma, esta deficiência é contornada.

Em relação ao escopo, realmente o GEHR só considera a parte clínica do paciente, enquanto o HL7 e o CORBAMed inclui a parte administrativa. Mas, em compensação, o GEHR obriga os médicos a trabalharem com mais cuidado, porque possui trilhas de auditoria. Além disso, ele atende os requisitos éticos legais porque os médicos se responsabilizam pelas informações adicionadas no registro do paciente.

Apesar do custo de definição da arquitetura do GEHR ser maior que o dos outros padrões, sua implementação é bem mais barata. Além do mais, como a arquitetura do registro é genérica, provavelmente não deve sofrer alterações. O mesmo não acontece com os outros padrões porque sempre estão sofrendo manutenção.

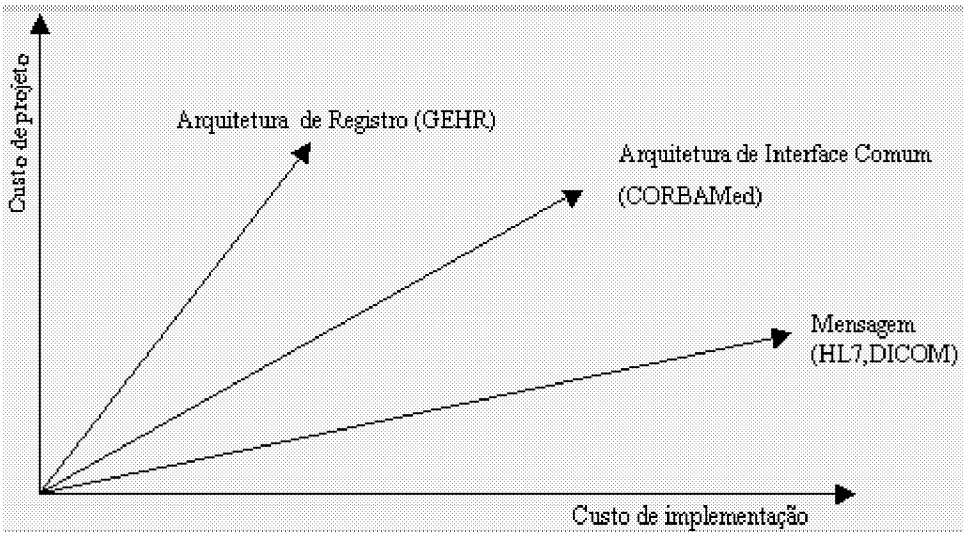


Figura 2.14: Comparação entre as diferentes abordagens

Apesar do GEHR não ter sido testado o bastante, nós o escolhemos como estudo de caso porque os benefícios que ele traz para os profissionais da área de sáude e principalmente para os pacientes são, potencialmente, bem maiores que os dos outros padrões apresentados. Por outro lado, o GEHR é bastante adequado ao estudo de caso já que é uma especificação complexa e orientada a objetos.

Capítulo 3

Estratégia de Modelagem

Este Capítulo apresenta notações e técnicas utilizadas na modelagem do nosso estudo de caso: a linguagem UML-RT *com sua formalização em* OZ*, a linguagem* CSP-OZ*, uma tradução formal dos componentes de* UML-RT *para processos em* CSP-OZ *e uma técnica de verificação de modelos de especificações em* CSP-OZ*.*

3.1 Estratégia de Modelagem

As linguagens UML-RT [79] e CSP-OZ [24], a tradução formal dos componentes de UML-RT para processos em CSP-OZ [25] e a técnica de verificação de modelos de especificações em CSP-OZ [30] são elementos necessários para a definição de uma estratégia de modelagem, a qual será utilizada no estudo de caso do Capítulo 5. Esta estratégia consiste das seguintes etapas:

1. Criar uma arquitetura de sistema utilizando diagramas de colaboração em UML-RT, pois a notação gráfica facilita o entendimento do sistema a ser desenvolvido.

2. Traduzir o modelo arquitetural descrito por diagramas de colaboração para processos em CSP-OZ.

3. Traduzir a especificação em CSP-OZ para CSP_M, utilizando a estratégia de Fischer [30], juntamente com o padrão de projeto, que propomos nesta dissertação, que permite a utilização de herança em CSP.

A motivação para a combinação destas linguagens é garantir que os modelos de sistemas gerados em cada etapa da modelagem sejam compatíveis. Utilizamos CSP-OZ como linguagem intermediária porque é grande a distância semântica entre UML-RT e CSP_M e também porque já havia na literatura dois trabalhos na mesma direção.

3.2 UML-RT

Como sistemas de software têm se tornado cada vez mais complexos, a comunidade de engenharia de software tem reconhecido a clara necessidade de especificar software no nível arquitetural. Isso resultou em duas linhas de trabalho. A primeira utiliza notações especializadas chamadas de ADLs (Architecture Description Languages); um exemplo é Wright [4], que é uma linguagem especificamente projetada para os aspectos comportamentais das descrições arquiteturais. A segunda utiliza notações de modelagem orientada a objetos, como por exemplo UML (Unified Modelling Language) [35], que é uma notação gráfica para modelagem de análise e projeto orientado a objetos.

UML-RT [79] é uma extensão de UML para modelagem de sistemas reativos e de tempo real. Embora o nome RT se refira a tempo real, o seu foco é a descrição de arquitetura de sistemas distribuídos interconectados. UML-RT define três novos construtores (cápsulas, portas e conectores), os quais são empregados nos diagramas de colaboração para obter a descrição de arquitetura. A Figura 3.1 apresenta um diagrama de colaboração contendo os três componentes.

As **cápsulas** descrevem componentes complexos, potencialmente concorrentes e possivelmente distribuídos, os quais podem interagir com seu ambiente. Elas podem conter outras cápsulas, as quais são chamadas de sub-cápsulas. As sub-cápsulas, por sua vez, podem conter outras sub-cápsulas e assim por diante. Uma **porta** é uma parte da implementação de uma cápsula que intermedia a interação da cápsula com o ambiente. Ela é um objeto que implementa uma interface específica. As portas estão associadas com **protocolos** que regulam o fluxo de informação que passa através das portas. A porta preta é a emissora de mensagens e a branca a receptora. Portas podem ser públicas ou privadas. Na Figura 3.1 as portas p_1,p_2, p_3, p_4 e p_5 são privadas e p_6 e p_7 são

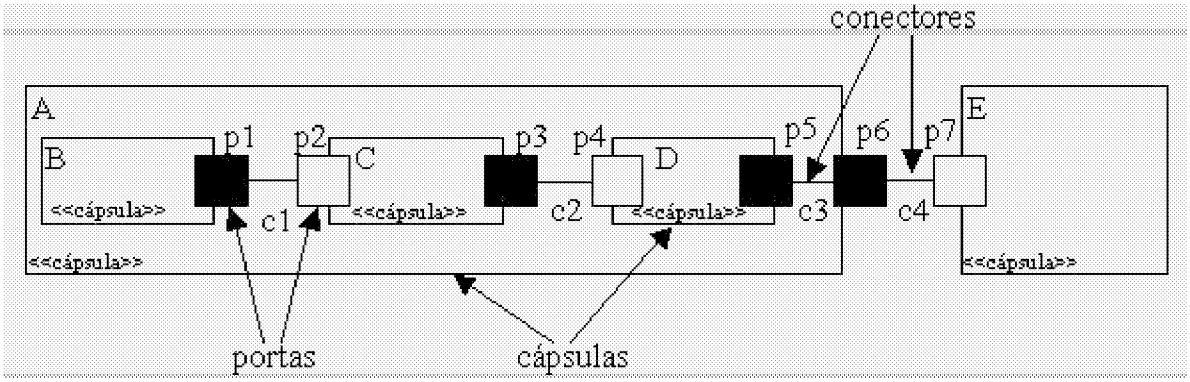


Figura 3.1: Exemplo de diagrama de colaboração em UML-RT

públicas. Os **conectores** são utilizados para conectar duas ou mais portas e descrevem o relacionamento de comunicação entre as cápsulas.

Outra notação freqüentemente utilizada em diagramas de colaboração de UML-RT é a de objeto múltiplo (representado por uma pilha de retângulos) como mostra a Figura 3.2. A quantidade de objetos de uma pilha representa o número de instâncias.

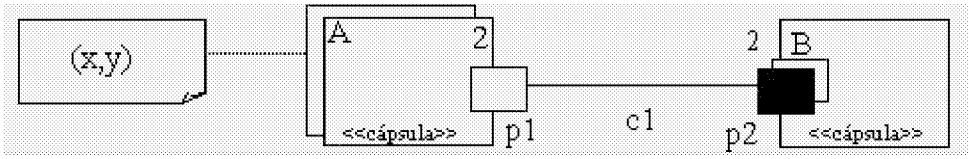


Figura 3.2: Exemplo de objeto múltiplo

Quando se trabalha com objetos múltiplos é necessário informar para qual objeto se está enviando a mensagem. No nosso exemplo, a cápsula A possui duas instâncias, x e y. Então, a porta da cápsula B terá de informar se está mandando mensagens para x ou y.

A vantagem de se utilizar UML-RT é porque ela fornece uma notação gráfica para a modelagem dos sistemas. Porém, UML-RT não tem uma semântica precisa. Uma outra abordagem seria a utilização de uma linguagem formal como CSP-OZ que possui uma semântica precisa. O uso em conjunto de métodos formais e linguagens de modelagem gráfica oferece benefícios gráficos e resolve o problema da ausência de uma semântica precisa. Podemos encontrar alguns trabalhos combinando métodos formais e UML [22, 49].

Para poder utilizar uma linguagem gráfica para modelagem da arquitetura do sistema de prontuário eletrônico (estudo de caso) e ao mesmo tempo formalizar esse modelo, utilizamos a técnica proposta em [25] para converter especificações UML-RT para CSP-OZ.

Formalização em Z da Sintaxe dos Diagramas de UML-RT

Fischer [25] especificou formalmente, usando Z, elementos presentes em descrições de arquitetura em UML-RT — como, por exemplo, portas, conectores, e cápsulas. Para que possa ser feita a tradução dos diagramas de colaboração em UML-RT para processos CSP, é necessário que haja uma descrição precisa da notação de UML-RT. Para atingir esse objetivo é utilizado Z e sua extensão, Object-Z, as quais têm sido freqüentemente utilizadas para descrição da sintaxe dos diagramas UML [49].

Como apresentado na Seção 3.2, o diagrama de colaboração é formado por cápsulas, portas, conectores e instâncias de cápsulas. Cada um desses elementos é representado através dos seguintes tipos básicos: *CapName*, *PortName*, *ConName*, *InstName*. Toda porta possui uma cor (preta ou branca). Essas cores são modeladas através do *free type* Colour que pode assumir os valores *black* e *white*.

$$Colour ::= black \mid white$$

Uma porta é representada pela classe Object-Z *Port*. O seu estado é formado pelo seu nome, a cápsula a qual está conectada, a cor que indica se a porta é de entrada (branca) ou de saída (preta), e por fim a multiplicidade da porta, que indica se ela é uma única porta (multiplicidade 1) ou porta múltipla (multiplicidade maior que 1). Por convenção, as portas brancas sempre têm multiplicidade 1. Como este trabalho não considera que portas múltiplas de uma cápsula possam ser conectadas a portas múltiplas de outra cápsula, mas apenas a portas únicas (de entrada) de cápsulas múltiplas, então, não há necessidade de se utilizar portas múltiplas de entrada.

Port

$$name : PortName$$
$$cname : CapName$$
$$colour : Colour$$
$$multi : \mathbb{N}_1$$

$$colour = white \Rightarrow multi = 1$$

Um conector é representado pela classe *Connector*, a qual possui dois componentes de estado: o nome do conector e um conjunto com duas portas. Por simplicidade, cada conector só pode se conectar a duas portas. Desta forma, estamos considerando apenas protocolo binário ou comunicação ponta-a-ponto. O nome de um conector não é sempre fornecido em diagramas de UML-RT, porém assumimos a existência de nomes únicos para todo conector.

Connector

$$name : ConName$$
$$ports : \mathbb{F}\, Port$$

$$\#ports = 2$$

Existem três tipos de cápsulas: a básica (*Capsule*), a qual não possui sub-cápsulas; a composta (*CompCapsule*) que possui no seu interior outras cápsulas interconectadas;

e a cápsula múltipla (*MultiCapsule*), que possui várias instâncias. Esses três tipos de cápsulas são formalizadas utilizando Object-Z. A classe *Capsule* reúne as características comuns das diferentes cápsulas. Ela é superclasse das classes *CompCapsule* e *MultiCapsule*. Cada cápsula possui uma quantidade de portas, que se comunicam com o ambiente.

A classe *Capsule* possui dois elementos de estado: o seu nome e um conjunto finito de portas públicas (acessadas pelo ambiente). O seu invariante determina que todas as portas devem pertencer a essa cápsula.

Capsule

$$name : CapName$$
$$ports : \mathbb{F}\, Port$$
$$\forall\, p : ports \bullet p.cname = name$$

A classe *ComCapsule* herda da classe *Capsule*. Herança é representada através da palavra-chave *inherit* e o nome da classe. Além dos atributos e invariante herdados, a classe possui mais dois atributos: o *scnames*, que representa o conjunto finito de subcápsulas, referenciadas pelo nome, e *conn*, o conjunto finito de conectores.

ComCapsule

inherit Capsule

$$scnames : \mathbb{F}\, CapName$$
$$conn : \mathbb{F}\, Connector$$
$$\forall\, c : conn \bullet c.ports \subseteq ports \cup \bigcup\{sn : scnames \bullet \mathcal{C}(sn).ports\}\ [1]$$
$$\forall\, c_1, c_2 : conn \bullet c_1.ports \cap c_2.ports \neq \varnothing \Rightarrow c_1 = c_2\ [2]$$
$$\forall\, c : conn \bullet \forall\, p, p' : c.ports \bullet p \neq p' \Rightarrow p.cname \neq p'.cname\ [3]$$
$$\forall\, c : conn \bullet \forall\, p, p' : c.ports \bullet (p \neq p' \Rightarrow$$
$$(p \in ports \Rightarrow p.multi = p'.multi \wedge p.colour = p'.colour)[4]$$
$$\wedge$$
$$(p, p' \notin ports \Rightarrow p.colour \neq p'.colour \wedge\ [5]$$
$$((p.multi = 1 \wedge p'.multi = \mathcal{C}(p.cname).multi)$$
$$\vee$$
$$(p'.multi = 1 \wedge p.multi = \mathcal{C}(p'.cname).multi))))$$

As portas das subcápsulas são consideradas privadas, ou seja, não podem ser acessadas pelo ambiente. Os conectores das portas privadas de uma subcápsula podem se conectar com portas privadas de outras subcápsulas ou com portas públicas da cápsula. Os predicados da classe *ComCapsule* possuem o seguinte significado:

- conectores só podem se conectar com portas da cápsula composta (condição 1). Esse predicado usa a função parcial $\mathcal{C}$ que mapeia o conjunto de nomes de cápsulas no conjunto de cápsulas. Ela é definida da seguinte forma:

$$\mathcal{C} : CapName \pfun \downarrow Capsule$$
$$\forall\, na : dom\ \mathcal{C} \bullet na = \mathcal{C}(na).name$$

A função *dom*, retorna o domínio de uma relação.

$$[X, Y]$$
$$dom : (X \leftrightarrow Y) \rightarrow \mathbb{P}\, X$$
$$\forall R : (X \leftrightarrow Y) \bullet \text{dom}\ R = \{x : X;\ y : Y \mid xRy \bullet x\}$$

- se existir dois conectores com as mesmas portas, então são idênticos (condição 2).
- conectores só podem se conectar com portas de diferentes cápsulas (condição 3).

  Com relação a multiplicidade das portas de um conector temos:

  - se uma porta do conector é pública, a multiplicidade e cor das duas portas do conector coincidem (condição 4).
  - Se as duas portas do conector são privadas, as cores das portas diferem, uma é porta única e a outra tem multiplicidade que coincide com a da subcápsula da porta única (condição 5).

A classe *MultiCapsule* é uma sub-classe da classe *Capsule* que, além de herdar os atributos da classe *Capsule*, possui uma outra cápsula com multiplicidade e nomes das instâncias. O seu invariante garante que a quantidade de nomes de instâncias deve ser igual a multiplicidade de cápsulas. Segue a classe MultiCapsule:

MultiCapsule
inherit Capsule
$$scname : CapName$$
$$multi : \mathbb{N}_1$$
$$inames : \mathbb{F}\, InstName$$
$$\#inames = multi$$

A cápsula múltipla é utilizada quando se deseja representar a comunicação entre um componente e um objeto múltiplo (instâncias de uma mesma classe) através de um mesmo canal.

Os sistemas podem ser construídos a partir de portas, conectores e cápsulas. A arquitetura de um sistema é definida através das cápsulas mais externas.

3.3 CSP-OZ

O principal propósito de especificação formal é fornecer descrições simples e precisas de sistemas de software. Em particular, para sistemas complexos ou grandes, esse objetivo é melhor alcançado utilizando mais de uma linguagem de especificação. Embora a maioria das linguagens de especificação possam ser usadas para especificar sistemas inteiros, elas não são adequadas para a modelagem de todos os aspectos de tais sistemas. Um exemplo onde a combinação das linguagens seria útil é em sistemas concorrentes ou distribuídos. Esses sistemas são constituídos de um número de processos que operam

concorrentemente e sincronizam em certos eventos. Álgebras de processos tais como CCS [58] e CSP [40] são adequadas para modelar as interações entre processos e a ordem em que as operações são realizadas no tempo. Linguagens baseadas em estado tais como Z [20] ou VDM [47] oferecem melhores facilidades para a especificação de estruturas complexas de dados e o que cada operação faz com elas.

A combinação em que estamos interessados é de Object-Z com CSP, porque ela nos oferece as vantagens que cada uma das linguagens possui, que são as técnicas comportamentais de CSP e as técnicas baseadas em estado de Object-Z. A vantagem de integrar as duas técnicas é que a combinação delas permite criar especificações mais adequadas, porque torna-se possível escolher a melhor técnica para cada aspecto do sistema. Essa combinação tem sido investigada por diversos pesquisadores como Smith [83], Fischer [27] e Mahony e Dong [54]. Object-Z [21] estende Z com a noção de classe, especificando o estado e as operações que atualizam esse estado. Nestes trabalhos, a linguagem Object-Z é usada para descrever os componentes de um sistema que são combinados com os operadores de CSP, os quais descrevem como os componentes sincronizam e se comunicam.

Dentre as combinações propostas, destacamos CSP-OZ [24], que estende Object-Z com a noção de canais de comunicação e sintaxe de CSP. Uma especificação CSP-OZ descreve um sistema como uma coleção de objetos, que interagem uns com os outros através da sincronização de canais. Cada objeto possui uma estrutura e um comportamento específico. A idéia por trás da integração desses dois formalismos é identificar toda classe Object-Z como um processo CSP. No entanto, para que isso aconteça é necessário que toda classe Object-Z tenha uma interface de comunicação (canais) e uma descrição comportamental em CSP.

3.3.1 Sintaxe de CSP-OZ

Apresentaremos uma visão da sintaxe da linguagem através do exemplo de um sistema bancário hipotético. Este sistema permite abrir conta corrente e poupança, realizar depósitos, resgates, solicitar saldo, e transferir valores entre contas. Para exemplificar a modelagem de paralelismo, adicionamos um componente chamado terminal de transferência financeira (TEF) que permite realizar pagamentos de compras com o banco. Estamos assumindo que o banco só possui uma agência, e por isso não é necessário incluí-la na especificação.

Começamos a modelagem através da classe CSP-OZ *Account*, a qual modela a entidade conta corrente. Antes da definição da classe *Account* introduzimos o tipo básico *Password*, o qual é descrito como um *given set*. Utilizamos *given sets* quando não há necessidade de detalhar a estrutura interna de um tipo. Logo após, temos as abreviações *Real* e *Number*, que representam, respectivamente, os possíveis valores utilizados nas operações bancárias e números de contas. Logo a seguir, temos o *free type Message* que possui os valores *AccountDoesNotExist*, *InsufficientBalance* e *WithoutReference*. Free types servem para modelar estruturas recursivas, objetos compostos, e coleções enumeradas.

$[Password]$
$Real == \mathbb{N}$

$Number == \mathbb{N}$

$Message ::= AccountDoesNotExist \mid InsufficientBalance \mid WithoutReference$

$Account(n : Number;\ v : Real;\ p : Password)$

$method\ deposit : [v? : Real]$
$method\ withdraw : [v? : Real]$
$method\ getBalance : [v! : Real]$
$method\ withdrawOk, withdrawNotOk$
$method\ getPassword : [p! : Password]$
$main = deposit?v \rightarrow main$
$\quad \Box$
$\quad withdraw?v \rightarrow (withdrawOk \rightarrow main \ \Box \ withdrawNotOk \rightarrow main)$
$\quad \Box$
$\quad getBalance!v \rightarrow main$
$\quad \Box$
$\quad getPassword!p \rightarrow main$

$num : Number$
$bal : Real$
$passw : Password$
$vl : Real$

$bal \geq 0$

INIT

$num = n \land bal = v \land passw = p \land vl = 0$

effect_deposit

$\Delta(bal)$
$v? : Real$

$bal' = bal + v?$

effect_getBalance

$\Delta()$
$v! : Real$

$v! = bal$

enable_withdrawOk

$bal \geq vl$

effect_withdrawOk

$\Delta(bal)$

$bal' = bal - vl$

enable_withdrawNotOk

$bal < vl$

effect_getPassword

$\Delta()$
$p! : Password$

$p! = passw$

effect_withdraw

$\Delta(vl)$
$v? : Real$

$vl' = v?$

A definição da classe começa com o seu nome seguido dos parâmetros opcionais, que, neste caso, são utilizados para inicializar as variáveis de estado. Em seguida, encontramos as definições dos canais *deposit*, *withdraw*, *getBalance*, *getPassword*, *withdrawOk* e *withdrawNotOk*, os quais utilizam a palavra-chave *method*. De acordo com Fischer [24], a palavra-chave *method* introduz métodos implementados pela classe e a palavra-chave *chan* declara métodos de outros objetos que podem ser utilizados pela classe. Os canais *deposit*, *withdraw* e *getBalance* comunicam valores porque possuem algum tipo de parâmetro, seja ele de entrada (?), de saída (!) ou simples (.). Por outro lado, os canais *withdrawOk* e *withdrawNotOk* não comunicam valores.

O comportamento concorrente de uma classe CSP-OZ é introduzido através da palavra-chave *main*, a partir da qual outras equações podem ser adicionadas para obter uma descrição mais estruturada (hierarquia de processos). A equação *main* define a ordem em que as operações devem ser executadas. No nosso exemplo, ela descreve o comportamento da classe *Account* em termos de escolha externa entre os eventos *deposit*, *withdraw*, *getBalance* e *getPassword*. A comunicação somente acontece quando o ambiente deseja realizar um desses eventos oferecidos pelo processo *main*.

Com relação à parte de dados, definimos o esquema de estado que contém quatro atributos: *num* representando o número da conta, *bal* o saldo da conta, *passw* a senha de acesso e *vl* o valor que se deseja debitar. Tivemos que adicionar o atributo *vl* no estado de *Account* porque esta especificação será utilizada pela estratégia de verificação de modelos (Fischer), a qual testa as condições apenas em função dos valores das variáveis de estado.

Além disso, o estado possui um invariante que diz que o saldo não pode ser negativo. Em seguida, temos o esquema de inicialização, o qual inicializa as variáveis de estado. Esse evento acontece antes de qualquer outro evento.

Todas as operações que possuem um canal associado devem ter o prefixo *effect* em frente ao nome da operação. Quando um evento c de CSP ocorre, a respectiva operação *effect_c* é executada, possivelmente modificando os atributos. Se um canal não tem um esquema associado, significa que não ocorrerá mudança de estado.

A declaração Δ apresenta os atributos que poderão ser alterados pelas operações. O predicado que se encontra nos esquemas *effect* descrevem a alteração de valores dos atributos e os valores dos parâmetros de saída quando a operação é executada. As operações associadas a canais possuem também um esquema *enable* associado, no qual é especificado os estados e parâmetros simples em que o canal não pode recusar a comunicação: a guarda da operação. Se o esquema *enable* for omitido, significa que o predicado é sempre verdadeiro. E no caso do esquema *enable* ser falso a operação não é executada.

A classe *Account* possui as seguintes operações:

- *effect_deposit*: recebe o valor *v?*, através da comunicação do canal *deposit*, e altera o componente de estado *bal* adicionando o valor recebido (*v?*). A variável *bal´* (pronunciada *bal* linha) descreve o estado resultante de *bal* após a execução da operação.

- *effect_getBalance*: atualiza o parâmetro de saída *v!* com o valor do saldo da conta (*bal*).

- *effect_withdrawOk*: subtrai o saldo do valor sacado. Esta operação só pode ser

executada se o predicado que se encontra no esquema *enable_withdrawOk* for verdadeiro, ou seja, se o valor que se deseja debitar for menor ou igual ao saldo da conta ($bal >= vl$).

- *effect_withdrawNotOk*: não altera o estado e nem gera algum valor de saída. A operação só é executada se o predicado que se encontra no esquema *enable_withdrawNotOk* for verdadeiro, ou seja, se o valor que se deseja debitar for maior que o saldo da conta ($bal < vl$).

- *effect_getPassword*: atualiza o parâmetro de saída *p!* com o valor da senha da conta (*passw*).

- *effect_withdraw*: recebe o valor *v?*, através da comunicação do canal *withdraw*, e altera o componente de estado *vl* adicionando o valor recebido (*v?*). A variável *vl´* descreve o estado resultante de *vl* após a execução da operação.

A segunda classe CSP-OZ a ser apresentada é a *Bank*, a qual modela alguns serviços de Banco.

$Bank(sqId : \text{seq}\, Number)$

$method\ createAccount : [p? : Password]$
$method\ enterAcc : [n? : Number]$
$method\ depositB, withdrawB : [v? : Real]$
$method\ getBalanceB, exit, getMoney, notGetIndObj : []$
$method\ transferValue : [cliAccNum, compAccNum : Number;\ v : Real]$
$chan\ deposit, withdraw : [v! : Real]$
$chan\ message : [m : Message]$
$chan\ withdrawOk, withdrawNotOk, transferOk, transferNotOk : []$
$chan\ getBalance : [b? : Real]$
$local\_chan\ notFound : [n : Number]$
$local\_chan\ add, update : [cp : \downarrow Account]$
$lacal\_chan\ found : [n : Number;\ cp? : \downarrow Account]$
$local\_chan\ foundAccs : [compAccNum, cliAccNum : Number;\ compAcc?, cliAcc? : Account]$
$local\_chan\ notFoundAccs : [compAccNum, cliAccNum : Number]$
$local\_chan\ getIndObj : [ind : Number]$
$main = createAccount?p \rightarrow$
$\quad (getIndObj?r \rightarrow New\ c : Account(r, 0, p) \bullet add.c \rightarrow main$
$\quad \Box$
$\quad notGetIndObj \rightarrow message?m \rightarrow main)$
$\Box$
$enterAcc?n \rightarrow$
$\quad (found.n?ac \rightarrow MovAcc(ac)$
$\quad \Box$
$\quad notFound.n \rightarrow message?m \rightarrow main)$

$$
\begin{array}{l}
\quad \Box \\
\quad transferValue?compAccNum?cliAccNum?v \rightarrow \\
\quad\ (foundAccs.compAccNum?compAcc.cliAccNum?cliAcc \rightarrow \\
\quad\ \ (cliAcc\ [|\ withdraw, withdrawOk, withdrawNotOk\ |] \\
\quad\ \ (withdraw!v \rightarrow \\
\quad\ \ \ ((withdrawOk \rightarrow update.cliAcc \rightarrow SKIP); \\
\quad\ \ \ \ (compAcc\ [|\ deposit\ |] \\
\quad\ \ \ \ (deposit!v \rightarrow upadate.compAcc \rightarrow transferOk \rightarrow SKIP));\ main \\
\quad\ \ \ \Box \\
\quad\ \ \ withdrawNotOk \rightarrow transferNotOk \rightarrow main))) \\
\quad\ \ \Box \\
\quad\ \ notFoundAccs.compAccNum.cliAccNum \rightarrow transferNotOk \rightarrow main) \\
MovAcc(ac) = depositB?v \rightarrow (ac\ [|\ deposit\ |]\ (deposit!v \rightarrow SKIP));\ MovAcc(ac) \\
\quad\ \ \Box \\
\quad\ \ withdrawB?v \rightarrow \\
\quad\ \ (ac\ [|\ withdraw, withdrawOk, withdrawNotOk\ |]\ (withdraw!v \rightarrow \\
\quad\ \ \ ((withdrawOk \rightarrow getMoney \rightarrow SKIP);\ MovAcc(ac) \\
\quad\ \ \ \Box \\
\quad\ \ \ (withdrawNotOk \rightarrow message?m \rightarrow SKIP);\ MovAcc(ac)))) \\
\quad\ \ \Box \\
\quad\ \ getBalanceB \rightarrow \\
\quad\ \ \ (ac\ [|\ getBalance\ |]\ (getBalance?b \rightarrow SKIP));\ MovAcc(ac) \\
\quad\ \ \Box \\
\quad\ \ exit \rightarrow update.ac \rightarrow main
\end{array}
$$

$$
\begin{array}{|l}
\hline
accounts : \mathbb{P}\ \downarrow Account \\
seqIds : \mathrm{seq}\ Number \\
msg : Message \\
\hline
\forall a, b : Account \mid a, b \in accounts \land a.num = b.num \bullet a = b \\
\hline
\end{array}
$$

INIT

$$
\begin{array}{|l}
\hline
accounts = \varnothing \\
seqIds = sqId \qquad \text{[class parameter]} \\
\hline
\end{array}
$$

enable_getIndObj

$$
\begin{array}{|l}
\hline
seqIds \neq <> \\
\hline
\end{array}
$$

effect_getIndObj

$$
\begin{array}{|l}
\hline
\Delta(seqIds) \\
ind? : Number \\
\hline
ind? = head\ seqIds \\
seqIds' = tail\ seqIds \\
\hline
\end{array}
$$

enable_notGetIndObj

$$
\begin{array}{|l}
\hline
seqIds = <> \\
\hline
\end{array}
$$

effect_notGetIndObj

$$
\begin{array}{|l}
\hline
\Delta(msg) \\
\hline
msg' = WithoutReference \\
\hline
\end{array}
$$

enable_found

$n : Number$

$\exists\, c : accounts \mid c.num = n$

effect_found

$\Delta()$

$n : Number$

$cp? :\downarrow Account$

$\exists\, c : accounts \mid c.num = n \bullet cp? = c$

enable_notFound

$n : Number$

$\nexists c : accounts \mid c.num = n$

effect_notFound

$\Delta(msg)$

$n : Number$

$msg' = AccountDoesNotExist$

effect_add

$\Delta(accounts)$

$cp : \downarrow Account$

$accounts' = accounts \cup \{cp\}$

effect_withdrawNotOk

$\Delta(msg)$

$msg' = InsufficientBalance$

enable_foundAccs

$\Delta()$

$compAccNum : Number$

$cliAccNum : Number$

$\exists\, c1, c2 : accounts \mid$

$c1.num = compAccNum$

$c2.num = cliAccNum$

effect_foundAccs

$\Delta()$

$compAccNum : Number$

$cliAccNum : Number$

$compAcc? : Account$

$cliAcc? : Account$

$\exists\, c1, c2 : accounts \mid$

$c1.num = compAccNum$

$c2.num = cliAccNum \bullet$

$compAcc? = c1 \land cliAcc? = c2$

enable_notFoundAccs

$\Delta()$

$compAccNum : Number$

$cliAccNum : Number$

$\nexists c1, c2 : accounts \mid c1.num = compAccNum \lor c2.num = cliAccNum$

effect_message

$\Delta()$

$m! : Message$

$m! = msg$

effect_update

$\Delta(accounts)$

$c :\downarrow Account$

$\exists\, as : accounts \mid as.num = c.num \bullet$

$accounts' = (accounts \setminus \{as\}) \cup \{c\}$

A classe possui outros elementos da sintaxe de CSP-OZ: o primeiro é a palavra-chave *local_chan* utilizada na declaração de canais que possuem operações locais. O segundo é o construtor *New*, responsável por criar instâncias de um determinado tipo. No nosso exemplo, ele pode criar objetos dos tipos *Account* (*New c:Account(r,0,p)*) e *Saving* (*New s:Saving(r,0,p,0)*). A classe *Saving* está definida a seguir. O terceiro elemento são os canais polimórficos (*add, found, update*), que permitem receber como parâmetro um

valor do tipo *Account* ou *Saving* (subclasse). Polimorfismo é um mecanismo que permite que o tipo de uma variável a ser declarada pertença a uma dada coleção de classes e não a uma única classe. Nas linguagens orientadas a objetos, é comum que tal coleção de classes seja todas as subclasses de alguma classe comum. Uma declaração polimórfica é feita através do símbolo $\downarrow$. Além dos canais polimórficos, *Bank* possui o componente de estado polimórfico *accounts*, que pode conter elementos do tipo *Account* ou *Saving*.

Uma classe CSP-OZ é ao mesmo tempo um tipo e um processo, e nós podemos observar o uso desta facilidade na classe *Bank*. O evento *update.ac* possui o parâmetro simples ac que é do tipo *Account*. Essa mesma variável ac é um processo que sincroniza com o processo $deposit!v \rightarrow$ SKIP, o qual se encontra dentro do processo *MovAcc(c)*.

Alguns canais, como por exemplo *createAccount* e *enterAcc*, não têm esquemas associados porque não alteram variáveis de estado, suas guardas são sempre verdadeiras e não geram nenhuma saída. Eles simplesmente recebem valores que são utilizados na criação dos objetos ou repassados para outros canais.

Estamos utilizando semântica de cópia na classe *Bank* porque esta classe também será usada pelo padrão de projeto (apresentado no próximo capítulo), que trabalha com semântica de cópia.

A terceira classe CSP-OZ a ser modelada é a *Saving*, a qual possui as mesmas operações que a classe *Account*, e mais a *interest*, representando a operação render juros.

Saving$(n : Number;\ v : Real;\ p : Password)$

method interest : $[t? : Ratio]$

inherit Account

$main = interest?t \rightarrow getBalance?s \rightarrow deposit.(t * s) \rightarrow main$

$ratio : Ratio$

INIT

$ratio = 0$

effect_interest

$\Delta()$

$t? : Ratio$

$ratio' = t?$

A classe *Saving* é um pouco diferente da classe *Account* porque ela utiliza o conceito de herança. Semanticamente, herança é caracterizada pela conjunção da parte de Z da superclasse com a parte de Z da subclasse, e composição paralela das partes de CSP com sincronização nos eventos comuns às duas classes. As equações de processo da superclasse são herdadas pela subclasse e podem ser usadas por ela. No nosso exemplo, *Saving* reutiliza os eventos *getBalance* e *deposit*.

Após a declaração dos canais, vem a palavra *inherit* seguida do nome da superclasse. Em Object-Z, a sintaxe de herança é um pouco diferente: simplesmente os nomes das classes herdadas são listados. Apesar de não aparecer no nosso exemplo, a palavra chave *inherit-interface* pode ser usada para importar somente a interface da superclasse. Como *Saving* é subclasse de *Account*, então ela herda todas as variáveis de estado, operações e comportamento de *Account*.

A classe *Saving* possui a mais o método *interest*, que altera o atributo taxa com o último valor aplicado. Em *main*, os eventos *getBalance* e *deposit*, herdados de *Account*, aparecem depois de *interest* porque *Saving* precisa do valor do saldo para então realizar o depósito (taxa*saldo), que representa o rendimento da poupança.

CSP-OZ também permite renomear operações herdadas. Por exemplo, a expressão *inherit Figura[clara/escura]* indica a herança da classe Figura com a renomeação da operação escura para clara. A renomeação pode ser feita em canais, definições locais, operações, processos CSP, e também em nomes de variáveis definidas dentro de esquemas, canais e operações. Se uma variável ocorre em mais de uma operação ou esquema, todas as ocorrências devem ser renomeadas. A renomeação de operações herdadas pode comprometer o polimorfismo, pois a subclasse deixa de ter a operação herdada e passa a ter uma outra operação com o mesmo corpo da que deveria ser herdada. O processo que define o comportamento dos objetos de *Saving* é a composição paralela do *main* de *Account* com o *main* de *Saving*, sincronizando na interseção dos seus alfabetos.

O surgimento da classe *Saving* ocasionou a adição de novos eventos à classe *Bank*, como apresentado abaixo. Em relação a parte de Z, não houve a necessidade de colocar explicitamente os esquemas relativos aos novos canais porque eles não alteram o estado, mas apenas repassam valores.

A classe *Bank* abaixo, apresenta apenas o seu comportamento em relação ao processo *Saving*. Podemos observar que este comportamento é similiar ao que ele tem em relação ao processo *Account*. A principal diferença é a inclusão de novos eventos, como, por exemplo, *interestB* e *interest*, que são eventos próprios de poupança.

$$
\begin{array}{l}
Bank(sqId : \mathrm{seq}\ Number) \\
\ldots \\
method\ createSaving : [p? : Password] \\
method\ enterSav : [n? : Number] \\
method\ interestB : [r? : Ratio] \\
chan\ interest : [r! : Ratio] \\
main = createSaving?p \rightarrow \\
\qquad (getIndObj?r \rightarrow New\ s : Saving(r, 0, p, 0) \bullet add.s \rightarrow main \\
\qquad \Box \\
\qquad notGetIndObj \rightarrow message?m \rightarrow main) \\
\qquad \Box \\
\qquad enterSav?n \rightarrow \\
\qquad (found.n?s \rightarrow MovSav(s) \\
\qquad \Box \\
\qquad notFound.n \rightarrow message?m \rightarrow main)
\end{array}
$$

$$
\begin{array}{l}
MovSav(s) = depositB?v \rightarrow (s\ [|\ deposit\ |]\ (deposit!v \rightarrow SKIP));\ MovSav(s) \\
\qquad\qquad \Box \\
\qquad\qquad withdrawB?v \rightarrow \\
\qquad\qquad (s\ [|\ withdraw, withdrawOk, withdrawNotOk\ |]\ (withdraw!v \rightarrow \\
\qquad\qquad\quad ((withdrawOk \rightarrow getMoney \rightarrow SKIP);\ MovSav(s) \\
\qquad\qquad\quad \Box \\
\qquad\qquad\quad (withdrawNotOk \rightarrow message?m \rightarrow SKIP);\ MovSav(s)))) \\
\qquad\qquad \Box \\
\qquad\qquad getBalanceB \rightarrow \\
\qquad\qquad\ (s\ [|\ getBalance\ |]\ (getBalance?b \rightarrow SKIP));\ MovSav(s) \\
\qquad\qquad \Box \\
\qquad\qquad interestB?t \rightarrow (s\ [|\ interest\ |]\ (interest!t \rightarrow SKIP));\ MovSav(s) \\
\qquad\qquad \Box \\
\qquad\qquad exit \rightarrow update.s \rightarrow main \\
...
\end{array}
$$

A última classe CSP-OZ a ser modelada é a *Terminal*, a qual solicita ao banco débito na conta remetente e crédito na favorecida.

Terminal($n : Number$)

$$
\begin{array}{l}
channel\ readCard : [ct? : Number] \\
channel\ getValue : [vl? : Real] \\
channel\ dialNumber, transferOk, transferNotOk \\
channel\ conectionOk, conectionNotOk, printReceipt \\
channel\ transferValue : [cliAccNum!, compAccNum! : Number;\ vl! : Real] \\
main = readCard?cliAccNum \rightarrow getValue?vl \rightarrow dialNumber \rightarrow \\
\qquad (conectionOk \rightarrow transferValue!cliAccNum!compAccNum!vl \rightarrow \\
\qquad\quad (transferOk \rightarrow printReceipt \rightarrow main \\
\qquad\quad \Box \\
\qquad\quad transferNotOk \rightarrow main) \\
\qquad \sqcap \\
\qquad conectionNotOk \rightarrow main)
\end{array}
$$

$$companyAccount : Number$$

INIT

$$companyAccount = n$$

effect_transferValue

$$
\begin{array}{l}
\Delta() \\
cliAccNum! : Number \\
compAccNum! : Number \\
vl! : Real \\
\hline
compAccNum! = companyAccount
\end{array}
$$

A classe *Terminal* não possui elementos novos de sintaxe. Ela só inclui o esquema

effect_transferValue, o qual solicita a transferência de um determinado valor da conta do cliente para a conta do lojista. Os outros canais não possuem esquemas associados.

A arquitetura do sistema é modelada da seguinte forma:

System

$$
\begin{aligned}
&main = New\ b : Bank \bullet \\
&\quad New\ t1 : Terminal(1) \bullet \\
&\quad New\ t2 : Terminal(2) \bullet \\
&\quad New\ t3 : Terminal(3) \bullet \\
&\quad b \\
&\quad [|\ \{|\ transferValue, transferOk, transferNotOk\ |\}\ |] \\
&\quad (t1 ||| t2 ||| t3))
\end{aligned}
$$

O sistema é caracterizado pela execução paralela do processo *Bank* com 3 processos *Terminal*, que estão rodando independentemente uns dos outros. Elas sincronizam no conjunto de eventos *transferValue*, *transferOk* e *transferNotOk*.

A sintaxe de CSP-OZ [24], dada pela notação BNF (*Backus-Naur Form*), pode ser vista abaixo. Uma especificação em CSP-OZ é uma composição de elementos das linguagens CSP e Object-Z, que obedecem a alguns critérios referentes ao escopo. O escopo de uma especificação de classe em CSP-OZ é representado por um retângulo aberto do lado direito, com o nome da classe na parte superior.

Uma especificação em CSP-OZ é uma seqüência de parágrafos, similar a uma especificação em Z ou Objetc-Z. Um parágrafo CSP-OZ é definido como segue:

$Paragraph_O ::= Paragraph_C$	-Parágrafo CSPZ
$\mid Class_O$	-Classe Object-Z
$\mid Class_C$	-Classe CSP-OZ

A linguagem CSPZ [26] é uma extensão conservativa das linguagens CSP e Z, preservando os aspectos sintáticos e semânticos de cada uma. Os seguintes aspectos são encontrados nas especificações em CSPZ:

1. parte de CSP - contém todos os canais, tipos de dados, processos e seus comportamentos.

2. parte de Z - contém esquemas e estruturas de dados que representam o estado do sistema, a inicialização do estado e operações com suas pré e pós condições. Para cada evento de CSP, existe um esquema em Z associado, o qual define a troca de estado.

A sintaxe de CSPZ é uma extensão da sintaxe padrão de Z com dois novos parágrafos: declarações de canais (Channel) e processos (Process). O $Paragraph_C$ é uma lista de parágrafos de CSPZ.

$Paragraph_C ::= Paragraph_Z$	-Parágrafo de Z
$\mid$ Channel	-Canal
$\mid$ Process	-Processo

A definição dos parágrafos de Z ($Paragraph_Z$) pode ser encontrada em [24].

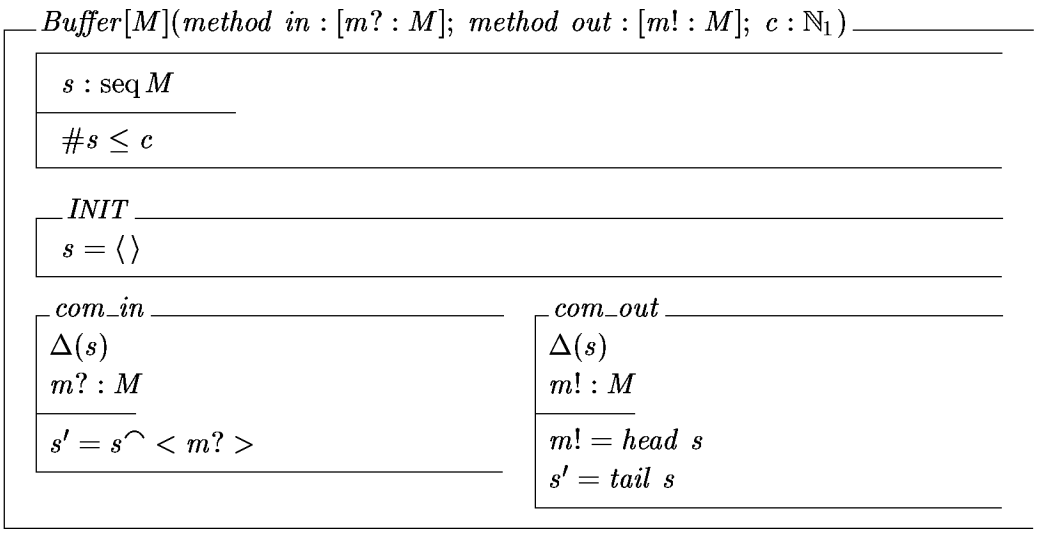
A principal diferença entre as classes $Class_C$ e $Class_O$ é que a primeira possui uma semântica comportamental e a segunda não. A classe CSP-OZ é uma versão orientada a objetos de um processo CSP. Classes Object-Z podem ser usadas para modelar os aspectos de dados de um sistema que não estão diretamente ligados a um processo. Classes CSP-OZ podem herdar ou instanciar classes Object-Z, mas o inverso não pode acontecer, porque as classes Object-Z não levam em consideração o comportamento do sistema. Uma classe Object-Z tem a estrutura abaixo. Na verdade, a sintaxe adotada é uma adaptação da versão original de Object-Z, como descrito em [24].

$Class_O$ ::=

$Word[FormalParamList][(SchemaText)]$

$[VisibilityList]$
$[inheritedClasses$
...
$InheritedClasses]$
$[LocalDefinition$
...
$LocalDefinition]$
$[StateSchema]$
$[InitialStateSchema]$
$[Operation$
...
$Operation]$

A classe é o item sintático mais importante de CSP-OZ. Ela é representada sintaticamente por um retângulo aberto com nome na parte superior, uma lista de parâmetros formais (*FormalParamList*) e os parâmetros de esquema texto (*SchemaText*). Os dois tipos de parâmetros citados acima são opcionais e restritos ao escopo da classe. Os parâmetros formais introduzem tipos básicos, e os de esquema texto, as declarações de canais e contantes. O parâmetro de esquema texto é um conceito novo que não existe na linguagem Object-Z original. Quando a classe é instanciada, tanto os parâmetros formais quanto os parâmetros de esquema são substituídos pelos reais. Por exemplo, a classe *Buffer* está parametrizada através do tipo de mensagem M que pode armazenar e liberar mensagens, através dos canais *in* e *out*, respectivamente, e que possui a capacidade de armazenar c mensagens.



A classe Buffer pode ser instanciada, por exemplo, para Buffer[$\mathbb{Z}$](a,b,3). Isto significa que os canais *in* e *out* são nomenados para a e b, respectivamente; a recebe as mensagens de valor do tipo inteiro e as armazena numa seqüência de inteiros cuja capacidade é limitada em 3 mensagens; b retira a mensagem da cabeça da lista e a envia.

A lista de visibilidade (*VisibilityList*) é uma lista de constantes, variáveis de estado, esquema de estado inicial, e operações que podem ser referenciados pelo ambiente. Se nenhuma lista de visibilidade existe, todas as características da classe são visíveis. Ela é usada somente em classes Object-Z.

O próximo item é a seqüência de superclasses (*InheritedClasses*) que são herdadas pela classe. O conceito básico de herança em CSP-OZ e Object-Z é o mesmo: as definições são unidas (usualmente através de conjunção) com as definições de mesmo nome da classe herdeira. Como em Object-Z, herança múltipla é permitida. Uma classe Object-Z só pode herdar de outras classes Object-Z. O mesmo não acontece com as classes CSP-OZ (ver a seguir) que podem herdar tanto de classes CSP-OZ quanto de Object-Z.

As definições locais (*LocalDefinition*) de uma classe incluem tipos, constantes, abreviações, tipos livres, e esquemas, os quais só podem ser usados dentro da classe. A sintaxe das definições locais é a mesma que em Z. O nome das declarações locais devem ser únicos dentro da classe.

As variáveis de estado de uma classe são definidas dentro do esquema de estado (*StateSchema*), que possui também o invariante da classe. Invariante é um predicado que deve ser preservado por todos os estados da classe. O esquema de estado inicial (*InitialStateSchema*) define o estado inicial de uma classe.

Por fim, as operações (*Operation*) descrevem as possíveis mudanças de estado que uma classe pode realizar. Um esquema de operação é um esquema com uma lista de variáveis de estado (Δ *list*) que devem ser alteradas, uma declaração de parâmetro e um predicado.

Para todo canal *op* declarado na interface, deve existir um esquema *effect_op* e um esquema opcional *enable_op*, ou somente um esquema *com_op*. O esquema *enable_op* define a guarda da operação *op*. Ou seja, se *enable_op* for válido então o evento *op* correspondente pode ocorrer, caso contrário o mesmo é refutado. Se o esquema *enable_op* for omitido então o evento *op* não pode ser refutado (*enable_op* $\equiv$ true). Dentro deste esquema não é permitido Δ *list* e nem parâmetros de saída. O esquema *effect_op* denota

o efeito da operação op e é um esquema como operação de Z tradicional, sem quaisquer restrições. Note que o uso de *enable_op* e *effect_op* pode introduzir divergências, quando o predicado $enable\_op \wedge \neg \text{ pre } effect\_op$ for válido. Quando o esquema *com_op* é usado, ele representa uma combinação padrão de *enable_op* e *effect_op*. Assim sendo, a guarda é dada por pre *com_op* e o efeito pelo próprio *com_op*. Dessa forma, ao se usar *com_op*, no lugar do par *enable_op* e *effect_op*, tomou-se a decisão por não introduzir divergências através da parte de Z, pois $enable\_op \wedge \neg \text{ pre } effect\_op$ nunca será válido.

A classe CSP-OZ é sintaticamente similar a uma classe Object-Z. A diferença é que a classe Object-Z possui opcionalmente uma lista de visibilidade, e a CSP-OZ possui uma interface e uma lista de processos de CSPZ, que juntos formam a parte de CSP da classe.

$Class_C ::=$

Word[*FormalParamList*][(*SchemaText*)]

[*Interface*
...
Interface]
[*inheritedClasses*
...
InheritedClasses]
[*LocalDefinition*
...
LocalDefinition]
[*Process*
...*Process*]
[*StateSchema*]
[*InitialStateSchema*]
[*Operation*
...
Operation]

A interface (*Interface*) define o alfabeto (conjunto de eventos) de uma classe CSP-OZ, que pode ser usado para a comunicação. Os objetos de uma classe CSP-OZ podem fazer chamadas a métodos da própria classe (*local_chan*) e de outras classes (*method* e *chan*). Quando um método é implementado pela classe, utiliza-se a palavra-chave *method* e, quando é apenas usado pela classe, utiliza-se a palavra-chave *chan*.

Como já dito, a idéia principal de CSP-OZ é mapear as operações de Object-Z em um ou mais eventos de CSP. Se considerarmos cada operação Object-Z relacionada a um único evento (*single event view*), então na declaração dos canais utilizamos as palavras-chaves *method* ou $method_1$, *chan* ou $chan_1$, *local_chan* ou $local\_chan_1$. (Esta visão foi adotada na tese de Smith [82] e nas primeiras combinações de CSP e Object-Z [83, 29]). Por outro lado, se desejarmos mapear uma operação Object-Z em dois eventos CSP-OZ (*double event view*), então utilizamos as declarações $method_2$, $chan_2$, $local\_chan_2$. Fischer argumenta o seguinte [24]: "A visão de único evento utiliza um conjunto mínimo de eventos e é portanto fácil de analisar, porém se outros métodos são chamados durante a execução de uma operação, a visão de único evento deixa de ser um modelo adequado. Por essa razão, nós também incluímos a visão de evento duplo, onde cada operação é mapeada em dois eventos: um para leitura de valores de parâmetros de entrada e outro

para valores de parâmetros de saída". Segue um exemplo onde apresentamos um método declarado como duplo na interface da classe C.

C

$method_2\ in : [i? : \mathbb{N};\ o! : \mathbb{Z}]$

$main = \Box e : \mathbb{Z} \bullet in?f.e \rightarrow Stop$

O processo C.main é equivalente a □ e:ℤ • in_b?f → in_e.e → Stop.

A escolha entre as duas visões é uma decisão de projeto, porque depende do nível de abstração desejado. A visão de evento duplo permite modelar especificação em um nível mais concreto, próximo ao de uma implementação, porque normalmente os eventos de CSP se transformam em métodos não-atômicos. Já a visão de único evento permite modelar a especificação em nível de abstração mais alto, pois considera os eventos atômicos.

A parte de CSP (*Process*) é um conjunto de equações que se encontram na forma "Nome do processo CSP = processo CSP". Os canais que se encontram dentro da parte de CSP devem ser um subconjunto dos canais declarados dentro da interface. Se a parte de CSP não está vazia, então um processo com a palavra chave *main* deve ser definido. Outros processos podem ser usados para melhorar a legibilidade da parte de CSP.

3.3.2 A Semântica de CSP-OZ

Antes de introduzirmos a semântica de CSP-OZ, vamos fazer uma breve explicação sobre os três modelos semânticos para CSP [40, 15, 5, 6].

A semântica de um processo CSP pode ser descrita em termos de conjuntos de *traces*, de falhas, e de falhas e divergências. Um trace é uma seqüência de comunicações com o ambiente, que ocorrem em uma determinada ordem. Dado um processo P, traces(P) é definido como o conjunto de todos os possíveis traces que podem ser realizados pelo processo. Por exemplo, seja P um processo definido pela equação P = (a → b → STOP), então $traces(P) = \{<>, < a >, < a, b >\}$. O trace <> pertence ao conjunto de traces de qualquer processo.

O modelo de traces é o modelo mais simples, porém, ele não consegue descrever completamente o comportamento do processo. Por exemplo, suponha que o processo Q é definido pela equação Q = a → STOP □ b → STOP e o processo R pela equação R = a → STOP ⊓ b → STOP. Os traces dos dois processos são exatamente os mesmos ($traces(Q) = traces(R) = \{<>, < a >, < b >\}$), mas o comportamento não é, porque no caso do processo Q, os eventos a e b são oferecidos ao ambiente e em R não são; quem decide qual evento realizar é o próprio processo R.

O modelo de *traces* é útil quando se deseja verificar se um processo não realiza nenhum evento que não deveria realizá-lo. O modelo de *traces* diz apenas o que o processo pode fazer, mas não diz nada sobre o que pode recusar a realizar.

O segundo modelo semântico, chamado de modelo de falhas (failures), já trata a questão de refutação de eventos que podem levar a falha do processo. Ele descreve um processo Q como um conjunto de pares (s,X), onde s é um trace que pertence ao conjunto traces(Q), e X é o conjunto de eventos que Q pode recusar depois que ele tenha realizado o trace s. Por exemplo, para os processos Q e R definidos acima, temos:

failures(Q) = {(<>,{}), (<a>,{a,b}), (<b>,{a,b}), (<a>,{a}), (<a>,{b}), (<a>,{}), ...} e
failures(R) = {(<>,{a}), (<>,{b}), (<>,{}), (<a>,{a,b}), (<b>,{a,b}), (<a>,{a}), (<a>,{b},), (<a>,{}),...}.
Como podemos observar no exemplo, o conjunto de falhas do processo R possui dois elementos que não existem no conjunto de falhas de Q, que são (<>,{a}),(<>,{b}), isto porque o processo Q sempre oferece ao ambiente os eventos a e b. No caso do processo R, ele pode decidir recusar um dos dois eventos.

Esse modelo apesar de identificar as falhas, não consegue identificar um comportamento divergente de um processo, ou seja, se um processo pode realizar uma seqüência infinita de eventos não observáveis pelo ambiente. Logo, deve-se adicionar o conceito de divergência.

As divergências de um processo são o conjunto de todos os traces que podem levar o processo a divergir. O modelo de Falhas-Divergências representa um processo Q pelo par ($\mathcal{F}$,$\mathcal{D}$), onde $\mathcal{F}$ é um conjunto de pares (s,X) representando as falhas do processo e $\mathcal{D}$ é o conjunto que registra os traces, que levam a divergência. Por exemplo, o processo U = a $\rightarrow$ U $\Box$ b $\rightarrow$ U nunca diverge, ou seja, o seu conjunto de divergências é $\mathcal{D}$(U) = {}. Por outro lado, o processo T = a $\rightarrow$ T $\Box$ T pode divergir antes de realizar qualquer evento (trace vazio) ou a partir de qualquer seqüência formada pelo evento a, ou seja, o seu conjunto de divergência é $\mathcal{D}$(T) = {<>,<a>,<a,a>,<a,a,a>,...}.

A linguagem CSP-OZ possui uma semântica formal baseada no modelo de Falhas e Divergências de CSP. A semântica da especificação de uma classe CSP-OZ é derivada em dois passos:

- a parte de Z da classe é dada uma semântica de falhas e divergências (via uma tradução para CSP) e
- a semântica da parte de Z é combinada com a semântica da parte de CSP através do operador de composição paralela de CSP.

A semântica de CSP-OZ [28] foi definida baseada na semântica de Z como apresentada no documento [20]. Tipos, expressões e esquemas são interpretados de acordo com essa semântica. Toda classe Object-Z então é vista como um processo, onde os métodos da classe respeitam as regras dos eventos de comunicação. Os seguintes pontos explicam as principais considerações da semântica das classes Object-Z:

- Todo método, cuja guarda é verdadeira no estado corrente, pode ser executado.
- Os valores para os parâmetros de entrada são determinados através da comunicação do ambiente com o objeto.
- Os valores dos parâmetros de saída e das variáveis de estado são sempre escolhidos não-deterministicamente a partir de um conjunto de possíveis valores especificados dentro do esquema *effect* do método. A escolha dos valores não é influenciada pelo ambiente.
- Uma chamada de método, cuja guarda é verdadeira, porém a pré-condição do esquema effect não é satisfeita, leva a divergência do processo.

Essa semântica permite habilitação e bloqueio de métodos: as guardas podem ser usadas para bloquear ou habilitar a execução dos métodos quando o objeto se encontra em um determinado estado. A execução de métodos habilitados quando a sua pré-condição não é satisfeita leva o processo a execução de um número infinito de eventos internos (não observados pelo ambiente).

3.4 Tradução de Diagramas UML–RT para CSP-OZ

De acordo com o trabalho de Fisher [25], toda cápsula de um diagrama UML-RT pode ser instanciada para um processo CSP específico. Portas são modeladas através de canais. Subcápsulas dentro de alguma cápsula devem ser colocadas em composição paralela com um conjunto de sincronização apropriado, garantindo as interconexões entre cápsulas como definido pelos conectores. Se uma cápsula é básica, ou seja, não contém subcápsulas, a única parte que é usada na descrição da arquitetura é o nome da cápsula. A definição do processo ou da classe CSP-OZ, que está associado a alguma cápsula, deve ser especificada a parte.

Antes de apresentar a tradução em detalhes e formalmente, apresentaremos uma tradução informal através de exemplos. A Figura 3.1 apresenta quatro cápsulas básicas, as quais não possuem subcápsulas. Para esses casos, assumimos que já temos a definição da classe CSP-OZ e só precisamos usar seus nomes (classes B, C, D e E). Todas as portas dessas cápsulas possuem multiplicidade igual a 1. Não é necessário usar endereçamento, pois neste caso não existe objeto múltiplo. Para derivar o processo da cápsula composta A, temos que compor os processos B, C e D em paralelo, nesta ordem. Para que processos em CSP-OZ se comuniquem, é preciso que os canais tenham o mesmo nome e tipo. Se utilizarmos, na tradução, os nomes das portas, então não acontecerá comunicação entre os processos porque cada porta tem um nome próprio. Como solução utilizamos os nomes dos conectores como conjunto de sincronização e renomeamos os nomes das portas para nomes de conectores das cápsulas. No nosso exemplo, temos as seguintes renomeações: $R_B = \{p_1 \mapsto c_1\}$, $R_C = \{p_2 \mapsto c_1,\ p_3 \mapsto c_2\}$ e $R_D = \{p_4 \mapsto c_2,\ p_5 \mapsto c_3\}$.

A comunicação entre as cápsulas B, C e D é modelada através da sincronização de CSP. Apesar da linguagem UML-RT permitir a comunicação assíncrona, este trabalho só considera a síncrona. Para obter o processo da cápsula composta A, teremos que fazer a renomeação do canal c_3 para p_6 (porta pública). O processo A resultante fica da seguinte forma:

$$\mathrm{A} = ((B[R_B]_{\{c_1\}} \parallel_{\{c_1,c_2\}} \mathrm{C}[R_C])_{\{c_1,c_2\}} \parallel_{\{c_2,c_3\}} \mathrm{D}[R_D])\ [c_3 \mapsto p_6] \setminus \{c_1,c_2\}$$

O processo CSP descrevendo a arquitetura do sistema é a composição paralela das duas cápsulas A e E, novamente aplicando renomeação, sincronização e ocultação das portas privadas.

$$\mathrm{Sistema} = (A[p_6 \mapsto c_4]_{\{c_4\}} \parallel_{\{c_4\}} \mathrm{E}[p_7 \mapsto c_4]) \setminus \{c_4\}$$

Com relação ao exemplo da Figura 3.2, como existem múltiplos objetos, então temos que trabalhar com endereçamento. A porta p_2, com multiplicidade 2, indica a possibilidade da cápsula B escolher entre os dois receptores de mensagens enviadas pela porta

p_2. Isso é modelado parametrizando o processo B com o parâmetro formal Adr_p para representar o conjunto de possíveis receptores. Suponha que o processo CSP da cápsula B seja da seguinte forma:

B({x,y}) = receive?w →
if w < 0
then p_2.x!w → B({x,y})
else p_2.y!w → B({x,y})

O processo B({x,y}) repetidamente recebe um valor w e verifica se ele é menor que zero. Se for, B envia a mensagem para o endereço x via a porta múltipla p_2. Senão, B envia a mensagem para o endereço y.

Suponha, agora, que o processo CSP da cápsula A seja da seguinte forma:

A = p_1?s → Print(s);A

Então, A repetidamente recebe um valor pela porta p_1, através do parâmetro s, e imprime o valor de s.

O próximo passo é construir o processo para a cápsula múltipla A. Seja MA o nome desse processo. A semântica de uma cápsula múltipla é o entrelaçamento de todas as suas instâncias. Para alcançar o endereço correto das instâncias, a comunicação sobre a porta p_1 deve ser renomeada para p_1.in, onde in é o conjunto de nomes de instâncias.

$$\mathrm{MA} = |||_{in:\{x,y\}} \mathrm{A}[p_1 \mapsto p_1.\mathrm{in}]$$

Dessa forma o nome in da instância é transmitido como parte do valor através do canal p_1. Utilizando o processo A apresentado anteriormente, obtemos:

$$\mathrm{A}[p_1 \mapsto p_1.\mathrm{in}] = (p_1.\mathrm{in?s} \rightarrow \mathrm{Print(s)});\mathrm{A}[p_1 \mapsto p_1.\mathrm{in}]$$

Para finalizar, o processo CSP que descreve a arquitetura do sistema fica da seguinte forma:

$$\mathrm{Sistema} = (B(\{x,y\})[p_2 \mapsto c_1]_{\{c_1\}} \|_{\{c_1\}} \mathrm{MA}[p_1 \mapsto c_1]) \setminus \{c_1\}$$

A cápsula B pode usar nomes de instâncias x e y como parâmetro para o canal p_2 e então enviar mensagens para a instância desejada.

Definição Formal da Tradução de Diagramas de UML–RT para CSP

Reproduzimos, a seguir, as definições formais da maneira como são apresentadas em [25]. Será apresentada a função τ responsável pela tradução de uma cápsula em um processo CSP. Vamos começar definindo os tipos básicos Process, Chans e Val, que são conjuntos de processos, canais e valores, respectivamente. Um dado pode ser um valor ou uma dupla, cujo o primeiro elemento é o nome da instância e o segundo, o valor. Events representa o alfabeto do processo.

$$[Process, Chans, Val]$$

$$Data ::= basic\langle\langle Val\rangle\rangle \mid comp\langle\langle InstName \times Data\rangle\rangle$$

$$Events == Chans \times Data$$

Por convenção, dados da forma comp(in,v) são abreviados para in.v, e dados basic(v) são abreviados para v. Seguindo as convenções de CSP, eventos (ch,d) são escritos como ch.d; então um evento típico é descrito como ch.$in_1...in_m$.v onde m $\in \mathbb{N}$. A seqüência $in_1...in_m$ de nomes de instâncias participam do endereço para onde o valor v deve ser enviado pelo canal ch.
Para todo processo existe um alfabeto de eventos associados:

$$\alpha : Process \rightarrow \mathbb{P}\, Events$$

A função de tradução

$$\tau : Capsule \rightarrow ProcessEquations$$

gera um conjunto de equações de processo da forma:

$$name(parameterList) = ProcessExpression$$

Esse conjunto é definido indutivamente sobre a estrutura sintática das cápsulas. Apresentamos a seguir as equações de processo de acordo com cada tipo de cápsula.
(1) Seja BC uma cápsula básica com BC.ports = $\{p_1,...,p_m,q_1,...,q_n\}$ onde m,n $\in \mathbb{N}$ e $p_1,..,p_n$ são portas simples, isto é, com pi.multi = 1 para i = 1,..,m, e $q_1,...,q_n$ são portas múltiplas , isto é, com q_j.multi > 1 para j =1,..,n. Então utilizamos novos parâmetros formais $Adr_1,...,Adr_n$ para representar conjuntos de nomes de instâncias que servirão como endereços através dos quais as múltiplas portas $q_1,...,q_n$ podem se conectar.

A função τ gera a seguinte equação de processo τ(BC) para BC:

$$\tau(\text{BC}) \equiv \text{BC.name}(Adr_1,...,Adr_n) = \text{RHS}$$

No nosso contexto $\equiv$ significa identidade sintática e RHS significa um processo com o alfabeto α(RHS) contido em

$$(\{p_1,..,p_m\} \times \text{Data}) \cup (\{q_1\} \times \text{comp}(Adr_1 \times \text{Data}) \cup ... \cup \{q_n\} \times \text{comp}(Adr_n \times \text{Data}))$$

No nosso caso, RHS é o nome da classe CSP-OZ correspondente.

(2) Seja CC uma cápsula composta com CC.ports = $\{p_1,...,p_m,q_1,...,q_n\}$ onde m,n $\in \mathbb{N}$ e $p_1,...,p_m,q_1,...,q_n$ tem o mesmo significa que em (1). Então, utilizamos novos parâmetros formais $Adr_1,...,Adr_n$ e definimos

$$\tau(\text{CC}) \equiv \text{CC.name}(Adr_1,...,Adr_n) =$$
$$((\|_{SN:CC.scnames}\ [\mathcal{C}(\text{SN}).\text{ports}[R_{SN,CC}]] \bullet \text{SN}(B_1,...,B_{k(SN)})[R_{SN,CC}])[P_{CC}]) \setminus H_{CC} \quad \text{[Eq1]}$$

$$\tau(\mathcal{C}(\text{SN})) \quad \text{[Eqs2]} \quad \forall\ \text{SN:CC.scnames}$$

Assim, τ(CC) gera uma nova equação de processo (Eq1), onde o lado direito usa repetidamente o operador de composição paralela alfabetizada e equações (Eqs2), obtidas aplicando a função tradução τ indutivamente para todas as subcápsulas $\mathcal{C}$(SN) dentro

de CC. O paralelismo alfabetizado se repete sobre todos os nomes SN das subcápsulas de CC.

Suponha que r_1,...,$r_{k(SN)}$ sejam portas múltiplas da subcápsula SN. Então, em Eq1 os parâmetros de endereço real B_i, com i = 1,...,k(SN), para essas múltiplas portas são definidos como segue:

- $B_i = Adr_{p_j}$ se r_i está conectada a uma múltipla porta pública p_j de CC.
- B_i = MC.inames, se r_i está conectada a uma única porta p_j de uma múltipla cápsula MC dentro de CC.

A renomeação $R_{SN,CC}$ troca nomes de portas para nomes de conectores:

$R_{SN,CC}$ = {pn:PortName; cn:ConName | ∃ p:$\mathcal{C}$(SN).ports; c:CC.conn •
p ∈ c.ports ∧ pn = p.name ∧ cn = c.name • (pn ↦ cn) }

A renomeação P_{CC} é usada para renomear conectores, que se encontram entre uma porta privada de uma subcápsula e uma porta pública da cápsula, para o nome da porta pública:

P_{CC} = {cn:ConName; pn:PortName | ∃p:CC.ports;c:CC.conn •
p ∈ c.ports ∧ pn = p.name ∧ cn = c.name • (cn ↦ pn) }

Por fim, todos os nomes dos conectores restantes são escondidos:

H_{CC} = { c:CC.conn • c.name}

(3) Seja MC uma cápsula múltipla com

$\mathcal{C}$(MC.cname).ports = {p_1,..,p_m,q_1,...,q_n} onde m,n ∈ ℕ e p_1,...,p_m,q_1,...,q_n tem o mesmo significado que em (1). Novamente utilizamos novos parâmetros formais Adr_1,...,Adr_n e definimos

τ(MC) ≡ MC.name(Adr_1,...,Adr_n) = $|||_{in:MC.inames}$ MC.cname(Adr_1,...,Adr_n)[(in)] [Eq1]

$\tau(\mathcal{C}$(MC.cname)) [Eqs2]

Então, τ(MC) gera uma equação de processo (Eq1), onde o lado direito usa repetidamente um operador de entrelaçamento, e as equações (Eqs2), obtidas aplicando a função tradução τ indutivamente para todas as subcápsulas dentro de MC. O operador de entrelaçamento se repete sobre todas as instâncias dessa cápsula.

Cada instância é formalizada aplicando o operador de renomeação denotado através do sufixo[(in)], onde in é o nome de uma instância. Para um dado processo P esse operador é definido através de

P[(in)] ≡ P[{ch:Chans; d:Data | ch.d ∈ α(P) • ch.d ↦ ch.in.d }]

Para finalizar, nós assumimos que um diagrama de colaboração implicitamente contém uma cápsula *Sistema* que engloba todas as cápsulas que aparecem dentro do diagrama. O processo CSP para a cápsula *Sistema* fornece a descrição da arquitetura.

3.5 Model Checking CSP-OZ

Ferramentas de apoio a especificação e verificação podem ajudar bastante em projetos de sistemas de software e de hardware. "Model checking" [17] é uma técnica automática para provar propriedades esperadas de sistemas concorrentes, especificados em uma notação formal. Exemplos de propriedades de interesse são ausência de "deadlock", "livelock". O "model-checker" verifica exaustivamente os estados de um sistema especificado para saber se existe algum em que a propriedade não é verdadeira. Se não encontrar tal estado, então a propriedade é verdadeira para esse sistema. Mas, se encontrar um estado, então o verificador de modelos fornece uma seqüência de passos para chegar ao estado em que a propriedade não é verdadeira e possivelmente um contra-exemplo. A sua aplicabilidade se dá pelo fato de existir ferramentas automáticas que realizam tais provas automaticamente.

O principal desafio no uso desta técnica é lidar com o problema de explosão de estados. Esse problema ocorre em sistemas com muitos componentes que podem interagir uns com os outros, ou em sistemas que possuem estruturas de dados que podem assumir muitos valores diferentes. Em tais casos, a quantidade de estados pode ficar enorme. Pesquisadores tem feito considerável progresso em relação a esse problema [18, 85].

Os verificadores de modelos têm sido aplicados extensivamente na verificação de hardware [7] e de protocolo [55]. A sua aplicação em sistemas de software tem sido limitada por várias razões. Uma delas é o estado de tais sistemas ser grande. Isto pode resultar na solicitação, pelo verificador de modelo, de uma quantidade de memória e tempo inaceitáveis. Outra razão é que as notações formais dos verificadores de modelos existentes permitem somente tipos e construtores de tipos simples porque foram criados para a modelagem de hardware [13], ou então são notações baseadas em eventos (álgebras de processo) as quais não são adequadas para modelagem de estruturas de dados [74].

Um dos trabalhos de Fischer e Wehrheim [30] foi desenvolver uma abordagem para traduzir especificações em csp-oz para um dialeto de CSP chamado CSP_M [75]. A tradução é necessária para que se possa fazer verificação automática das propriedades da especificação através do verificador de modelo FDR [31]. Esta é uma ferramenta baseada na teoria de álgebra de processo de CSP [40] e que é utilizada para verificar as propriedades de especificações em CSP. A tradução fornece um método para codificar classes CSP-OZ em notação CSP_M. A Tabela 3.1 apresenta um resumo da notação CSP comparada com a de CSP_M.

A linguagem CSP_M combina descrições de processos em CSP puro com uma pequena linguagem de programação funcional, que inclui conceitos como casamento de padrão (pattern matching), avaliação por demanda (lazy evaluation), construtor para compreensão de conjuntos e seqüências, etc. A parte funcional também pode ser usada para implementar especificação de dados de Z.

CSP	**CSPm**	**Descrição**
stop	STOP	Deadlock
skip	SKIP	Finalização com sucesso
a $\rightarrow$ P	a -> P	Prefixo simples
a?x $\rightarrow$ P	a?x -> P	Prefixo de entrada
a!x $\rightarrow$ P	a!x -> P	Prefixo de saída
a?b?:X!v $\rightarrow$ P	a?b?:X!v -> P	Prefixo compelxo
P $\Box$ Q	P [] Q	Escolha externa
P $\sqcap$ Q	P | | Q	Escolha interna
PA	PA	Esconde um conjunto de eventos
if b then P else Q	if b then P else Q	Escolha condicional
if b then P else stop	b & P	Guarda booleana
P[|X|] Q	P[|X|] Q	Composição paralela
P $\|\|\|$ Q	P $\|\|\|$ Q	Entrelaçamento
P $\gg$ Q	P[c <-> c'] Q	Piping
P(s)	P(s)	Parametrização
P(f(s))	let s' = f(s) within P(s')	Declaração local
P $\fatsemi$ Q	P;Q	Composição sequencial
$\Box_{i \in T} P_i$	[] i:T P(i)	Processo indexado [1]
P_i	P(i)	Parametrização

Tabela 3.1: Sumário da notação CSP$_M$ quando comparado com CSP

Tradução de uma Classe CSP-OZ para CSP$_M$

A tradução é dividida em duas partes: A primeira traduz as declarações Object-Z, como por exemplo, tipos básicos, descrições axiomáticas, esquema de estado, esquema de estado inicial, esquemas de operações, etc. A segunda define o comportamento dinâmico das classes, isto é, gera um termo de processo para cada classe.

Claramente, não é possível traduzir todos os construtores de CSP-OZ, porque existe uma grande distância entre a linguagem de especificação Z e a sublinguagem funcional CSP$_M$, utilizada pela ferramenta FDR. Então, para conseguir uma tradução automática, é necessário impor restrições sobre a parte de Z.

Os tipos em Z podem ser especificados livremente, ou seja, os seus valores não precisam ser limitados. Porém, todo parâmetro e variável deve ter um tipo. Em CSP$_M$ os valores dos tipos declarados necessitam ser restringidos. Desta forma, quando declaramos o tipo de um canal, na verdade estamos determinando os possíveis valores que podem passar pelo canal.

A linguagem CSP$_M$ possui dois tipos de dados pré-definidos, o *Int* representando os inteiros e o *Bool* os boleanos. A partir desses tipos, é possível construir outros tipos, como por exemplo, tipos compostos (tuplas). Além disso, CSP$_M$ permite usar as funções Set e Seq para definir conjuntos e seqüências de outros tipos. Desta forma, podemos ter classes que declaram canais desses tipos. Por outro lado, nenhum tipo parametrizado pode ser definido. Logo, não podemos manipular esquema de classe genérica que possui declaração de canais com algum tipo genérico. Além disso, não é permitido declarar

tipos livres, porque construtores de tipos arbitrários não são possíveis em CSP$_M$.

Exemplo

Para ilustrar a tradução utilizaremos a classe CSP-OZ `Account`, que está especificada na Seção 3.3.1. A especificação começa com o tipo básico `Password`, o qual não pode ser traduzido, porque não fixa valores reais. Como `Password` é o tipo do canal `getPassword`, e como toda declaração de canal precisa de um tipo, teremos de declarar valores explícitos para `Password`. Este tipo agora passa a ser um conjunto de elementos conhecidos, como, por exemplo:

```
Password == {1,2}
```

A classe `Account` utiliza as abreviações `Real` e `Number`, que representam os valores movimentados nas contas e números de contas, respectivamente. Como `Real` e `Number` são abreviações do conjunto dos números naturais, que é infinito, então teremos que limitar esses conjuntos para evitar a explosão de estados.

```
Real == {0,1}
Number == {2,3}
```

Os valores para os conjuntos `Password`, `Real` e `Number` foram escolhidos de forma ad-hoc. Como conseqüência, a especificação resultante pode perder propriedades em relação a especificação original.

A classe `Account` possui um invariante representado pela função `checkAcc(b)`. Ela é responsável por garantir que o saldo da conta não fica negativo. A função recebe como parâmetro o saldo (b) e testa se ele é maior que zero. Se for, retorna verdadeiro, senão, falso.

```
check(b) = b >= 0
```

Traduzindo Classes

Começamos com a tradução das declarações dos canais, os quais são as interfaces da classe. Para uma declaração do tipo ch = [p_1:$tipo_1$;p_2:$tipo_2$,...,p_n:$tipo_n$], nós adicionamos o seguinte *script*: `channel` ch:$tipo_1$.$tipo_2$,...,$tipo_n$. Segue as declarações dos canais do nosso exemplo.

```
channel deposit:Real
channel withdraw:Real
channel getbalance:Real
channel withdrawOk,withdrawNotOk
channel getPassword:Password
```

Seguindo a semântica de CSP-OZ, toda classe é traduzida para um processo. Isto é feito utilizando o construtor `let-within` de CSP$_M$. O processo `Account` possui três parâmetros para inicializar suas variáveis de estado: o primeiro (n), inicializa o número da conta, o segundo (v), o saldo e o terceiro (p), a senha. Os valores destes parâmetros são obtidos através dos eventos *createAccount* e *getIndObj*, realizados pelo processo `Bank`, como apresentado na Seção 3.3.1. `Account` é definido da seguinte forma:

```
Account(n,v,p) =
```

Primeiro, declaramos o conjunto de todas as operações executadas pelo processo da parte de Z (`Ops`). Em seguida, definimos o processo `main` e substituimos os parâmetros dos seus eventos de saída(!) por entrada (?), porque a parte de CSP descreve o comportamento do processo independentemente dos dados (não se refere a atributos da classe). Por este motivo, todos os parâmetros de nomes de canais devem ser de entrada.

```
let
  Ops = {deposit,withdraw,getBalance,withdrawOk,withdrawNotOk,
         getPassword}
  main = deposit?v -> main
         []
         withdraw?v -> (withdrawOk -> main [] withdrawNotOk -> main)
         []
         getBalance?v -> main
         []
         getPassword?s -> main
```

A descrição a seguir, representa o conjunto de todos os eventos utilizados na definição do processo `main`.

```
  CSPOps = Ops
```

Em relação a parte de Z, a idéia é traduzir os esquemas de estado e de estado inicial para conjuntos e as operações de Z para funções. O estado `state` representa o conjunto de todos os possíveis valores que o objeto `Account` pode assumir. Ele é formado por uma tupla contendo todos os elementos do estado. O estado possui um invariante, representado pela função `checkAcc(b)`, que não permite que a conta fique com saldo negativo. Nas operações em que variáveis de estado são modificadas, é necessário introduzir esta função para garantir que a operação só será efetivada se o saldo continuar positivo. O *init* é o conjunto de possíveis valores de inicialização das variáveis de estado.

```
state = {(num,bal,passw,vl) | num <- Number, bal <- Real,
       passw <- Password, vl <- Real, checkAcc(bal)}

init = {(num',bal',passw',vl') | (num',bal',passw',vl') <- state,
        num'== n, bal'== v, passw'== p, vl'== vl, checkAcc(bal')}
```

Para cada operação op existem quatro funções associadas descritas a seguir: in(op) e out(op) são conjuntos de possíveis valores de entrada e saída, enable(op)(s) retorna um valor boleano (se op está habilitada em s) e effect(op)(s,i) é um conjunto de possíveis valores de saída e de estados que podem ser alcançados depois de aplicar a operação op para o estado s com os valores de entrada i.

As operações `deposit` e `withdraw`, relaciondas a função `in`, possuem como parâmetro de entrada valores do tipo `Real`. As outras operações não possuem parâmetros de entrada, por isso o resultado da operação é o conjunto cujo elemento é um conjunto vazio.

```
in(deposit) = Real
in(withdraw) = Real
in(withdrawOk) = {{}}
in(withdrawNotOk) = {{}}
in(getBalance) = {{}}
in(getPassword) = {{}}
```

Apenas as operações `getBalance` e `getPassword`, relaciondas a função `out`, possuem como parâmetro de saída valores do tipo `Real` e `Password`, respectivamente. As outras operações retornam um conjunto vazio, porque não possuem valores de saída.

```
out(deposit) = {{}}
out(withdraw) = {{}}
out(withdrawOk) = {{}}
out(withdrawNotOk) = {{}}
out(getBalance) = Real
out(getPassword) = Password
```

Todas operações possuem guardas, que servem para testar as pré-condições das operações de Z descritas através de esquemas. As guardas, deste processo de tradução, se baseiam apenas no valor corrente dos componentes de estado. Os valores das variáveis de entrada não são considerados. Diante deste fato, para conseguir definir as guardas das operações `withdrawOk` e `withdrawNotOk`, foi necessário adicionar a variável "vl" no estado de `Account` para receber o valor do saque desejado.

Apesar da função enable não considerar as variáveis de entrada, o esquema enable de CSP-OZ as considera. As únicas operações que não possuem guardas triviais são `withdrawOk` e `withdrawNOk`, como apresentadas a seguir.

```
enable(deposit)((num,bal,passw,vl)) = true
enable(withdraw)((num,bal,passw,vl)) = true
enable(withdrawOk)((num,bal,passw,vl)) =  bal >= vl
enable(withdrawNotOk)((num,bal,passw,vl)) = bal < vl
enable(getBalance)((num,bal,passw,vl)) = true
enable(getPassword)((num,bal,passw,vl)) = true
```

A função effect é a mais complexa das quatro. Como exemplificação, temos a função `effect(deposit)((num,bal,passw,vl),i)`, que é o conjunto de estados do estado `state` e parâmetros de saída (neste caso {}), tal que o predicado do *effect_deposit* é válido. A tradução do predicado é direta. As duas últimas linhas captura a semântica do *delta list*: variáveis que não occorem dentro do *delta list* permanecem inalteradas. As operações restantes são traduzidas da mesma forma.

```
effect(deposit)((num,bal,passw,vl),i) =
  {({},(num',bal',passw',vl')) | (num',bal',passw',vl') <- state,
    num'== num, bal'== bal + i, passw'== passw,
    vl'== vl, checkAcc(bal')}

effect(withdraw)((num,bal,passw,vl),i)) =
  {({},(num',bal',passw',vl')) | (num',bal',passw',vl') <- state,
```

```
        num'== num, bal'== bal, passw'== passw,
        vl'== i, checkAcc(bal')}

    effect(withdrawOk)((num,bal,passw,vl),_) =
      {({},(num',bal',passw',vl')) | (num',bal',passw',vl') <- state,
        num'== num, bal'== bal - vl, passw'== passw,
        vl'== vl, checkAcc(bal')}

    effect(withdrawNotOk)((num,bal,passw,vl),_) =
      {({},(num',bal',passw',vl')) |(num',bal',passw',vl') <- state,
        num'== num, bal'== bal, passw'== passw,
        vl'== vl, checkAcc(bal')}

    effect(getBalance)((num,bal,passw,vl),_) =
      {(o,(num',bal',passw',vl')) |(num',bal',passw',vl') <- state,
        num'== num, bal'== bal, passw'== passw, o <- Real,
        vl'== vl, o == bal,checkAcc(bal')}

    effect(getPassword)((num,bal,passw,vl),_) =
      {(o,(num',bal',passw',vl')) |(num',bal',passw',vl') <- state,
        num'== num, bal'== bal, passw'== passw, o <- Password,
        vl'== vl, o == passw,checkAcc(bal')}

 within Semantics(Ops,in,out,enable,effect,init,CSPOps,main,event)
```

De acordo com a semântica de CSP-OZ, o comportamento de uma classe é a composição paralela do processo Z_MAIN definido abaixo, o qual modela a parte de Z e o processo main, representando a parte de CSP. A função Semantics traduz a parte de Z em processo e o coloca em paralelo com a parte de CSP. Esta função é definida da seguinte forma:

```
Semantics(Ops,in,out,enable,effect,init,CSPOps,main,event) =
 let
   DIV = DIV
   Z_PART(s) =
     [] op:Ops @ enable(op)(s) & [] i:in(op) @
     (if empty(effect(op)(s,i))
      then (|~| o:out(op) @ event(op,i,o) -> DIV)
      else (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> Z_PART(s')))
   Z_MAIN =  |~| (num,bal,passw,vl):init @ Z_PART((num,bal,passw,vl))
 within Z_MAIN [|{|op | op <- CSPOps|}|] main
```

O processo Z_PART recebe como parâmetro o estado corrente s do objeto. Este processo oferece uma escolha externa sobre todas as operações habilitadas. Os parâmetros de entrada são escolhidos pelo ambiente, através da escolha externa sobre in(op). Por outro lado, os parâmetros de saída e o estado final quem decide é a própria classe, através da escolha interna sobre as funções out(op) e effect(op). Este processo de tradução não considera o construtor New e nem a passagem de objetos com parâmetros.

De acordo com a semântica de CSP, se effect(op)(s,i) for igual ao conjunto vazio, significa que a operação foi habilitada, mas o predicado não foi satisfeito e o processo diverge depois de executar a operação op. A ferramenta FDR não representa a divergência como um processo. Por conseguinte, para tornar qualquer processo divergente é necessário introduzir uma chamada a um processo sabido ser divergente. Isto é conseguido facilmente através da equação DIV = DIV, como exibido na função `Semantics` acima.

A função `event(op,i,o)` traduz os parâmetros `op`, `i` e `o` em evento CSP válido. Ela é descrita da seguinte forma:

```
event(op,i,o) = (if i == {}
                 then (if o == {}
                       then op
                       else op.o)
                 else if o == {}
                      then op.i
                    else op.i.o)
```

O processo `Z_MAIN` é resultado de uma escolha não-determinística sobre todos os possíveis valores iniciais de `Z_PART`.

Tivemos que realizar algumas adaptações na função `Semantics` para atender às necessidades do nosso padrão de projeto que será apresentado no Capítulo 4. Adicionamos o parâmetro `LocOps`, o qual representa o conjunto de nomes de canais locais. Ele é utilizado para esconder do ambiente os canais locais do processo. Além disso, adicionamos na função `Semantics` o teste "if op == terminate" para que o processo da parte de Z pudesse ser finalizado juntamente com a parte de CSP, depois de realizarem o evento `terminate`. Segue a definição da função `Semantics` adaptada.

```
Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event) =
 let
   DIV = DIV
   Z_PART(s) =
     [] op:Ops @ enable(op)(s) & [] i: in(op) @
     (if empty(effect(op)(s,i))
      then (|~| o:out(op) @ event(op,i,o) -> DIV)
      else if op == terminate
           then (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> SKIP)
           else (|~| (o,s'):effect(op)(s,i) @ event(op,i,o) -> Z_PART(s')))
   Z_MAIN = |~| s:init @ Z_PART(s)
within (Z_MAIN [|{| op | op <- CSPOps |}|] main)\{| op | op <- LocOps |}
```

Capítulo 4

Padrão de Projeto em CSP_M para Especificações em CSP-OZ

Este Capítulo apresenta um padrão de projeto, usando a notação CSP_M, que permite representar conceitos de orientação a objetos como processos. Inicialmente, são apresentadas as características de CSP-OZ *capturadas em* CSP_M *que são utilizadas na definição do nosso padrão de projeto. Além do padrão, nossa principal contribuição, o capítulo apresenta duas outras abordagens para utilização de conceitos de orientação a objetos em* CSP. *Para finalizar, são apresentadas as conclusões sobre as três abordagens e o porquê de se ter escolhido a primeira.*

4.1 Introdução

A origem dos padrões de projeto se deu com os trabalhos feitos pelo arquiteto Cristopher Alexander no final dos anos 70 [2, 3]. Neles, encontramos exemplos e descrições de métodos utilizados para documentação de padrões. A sua contribuição, apesar de ser voltada para arquitetura, pode ser redirecionada para a área de software.

Os padrões permaneceram esquecidos por algum tempo, até que ressurgiram na conferência sobre programação orientada a objetos (OOPSLA) em 87 com o trabalho de Beck e Cunningham [11], que fala sobre uma linguagem de padrões para projetar janelas em Smalltalk. A partir de então, têm surgido vários artigos, revistas e livros [77, 51], que falam de padrões de software, os quais descrevem soluções para problemas que ocorrem freqüentemente no desenvolvimento de software e que podem ser reusadas por outros desenvolvedores.

Um padrão descreve uma solução para um problema que ocorre com freqüência durante o desenvolvimento de software, podendo ser considerado como um par "problema/solução" [14]. Projetistas familiarizados com certos padrões podem aplicá-los imediatamente a problemas de projeto, sem ter que redescobri-los [34].

Padrões de software podem se referir a diferentes níveis de abstração no desenvolvimento de sistemas orientados a objetos. Assim, existem padrões arquiteturais em que o nível de abstração é bastante alto, padrões de análise, padrões de projeto, padrões de código, dentre outros [78, 34, 32]. O uso de padrões proporciona um vocabulário comum para a comunicação entre projetistas, criando abstrações em um nível superior ao de programação e garantindo uniformidade na estrutura do software [33].

A especificação inicial do GEHR em CSP-OZ levou à constatação de que a estratégia para verificação de modelos, proposta por Fischer, não trata herança. Para resolver este problema, criamos um padrão de projeto em CSP, o qual será apresentado na Seção 4.3, que permite trabalhar com classes e suas especializações. Além desse padrão, apresentaremos na Seção 4.5 outras duas abordagens que também permitem modelar herança em CSP. Para facilitar o entendimento das abordagens, utilizaremos o exemplo de banco, conta e poupança, que se encontra na Seção 3.3.1, devido a sua simplicidade.

4.2 Capturando Características de CSP-OZ em CSP_M

A linguagem CSP-OZ, descrita no Capítulo 3, permite especificar sistemas concorrentes cujas estruturas de dados são modeladas usando o paradigma orientado a objetos. Portanto, objetos e classes, encapsulamento de dados e operações, subtipos e herança, criação e remoção dinâmica de objetos, são conceitos válidos em CSP-OZ.

Verificação de modelos em CSP-OZ utiliza uma estratégia elaborada por Fischer [30]. Porém, observamos que essa estratégia não trata alguns dos conceitos de orientação a objetos supracitados.

Utilizando os exemplos das classes CSP-OZ *Bank*, *Account* e *Saving* da Seção 3.3.1, observamos que os seguintes itens não são tratados pela estratégia de verificação de modelos para especificações em CSP-OZ:

- O processo *ac*, que representa uma conta na classe *Bank*, ora é um processo (acc [|getBalance|] (getBalance?b $\rightarrow$ SKIP)), ora é um valor (exit $\rightarrow$ update.acc $\rightarrow$ main).

- A criação dinâmica de objetos em CSP-OZ é dada através do construtor *New* como observado na classe *Bank* (*New c:Account(r,0,p)* • *add.c* $\rightarrow$ *main*).

- A remoção dinâmica é dada pelo processo SKIP, o qual está presente em diversas partes do fluxo de *Bank*.

- A herança é obtida adicionando a palavra-chave *inherit*, como apresentada na classe *Saving*.

- Uma variável polimórfica é declarada usando o símbolo de declaração polimórfica $\downarrow$ seguido do tipo da superclasse. Por exemplo, a classe *Bank* possui o atributo *accounts*:$\downarrow$ *Account*, que modela uma coleção de contas, ou seja, ele pode conter objetos do tipo conta e poupança.

Estes problemas podem ser tratados. Por exemplo, a seguir, apresentamos uma solução que permite CSP_M lidar com os conceitos de orientação a objetos utilizados pela linguagem CSP-OZ:

- **processos como valor:** O problema de processos serem comunicados através de canais foi resolvido da seguinte forma: se um processo A deseja receber um processo B através de comunicação, então comunica-se, na realidade, uma tupla contendo o estado do processo B. Quando A precisar trabalhar com B como um processo, este será gerado dinamicamente através da invocação de uma função posteriormente detalhada, onde o tipo e o estado do processo (a tupla comunicada) serão usados como parâmetros reais de entrada.

- **criação dinâmica de objetos:** A criação dinâmica é obtida através de uma função que recebe o tipo e o estado do processo como parâmetros reais de entrada, e através do casamento de padrão determina-se que processo será ativado. O presente trabalho utiliza semântica de cópia, a qual pode ocasionar problemas na distribuição, quando mais de uma aplicação se referenciar o mesmo processo. Além disso, ele está considerando a criação e utilização de uma instância de classe por vez.

- **remoção dinâmica de objetos**: A função `Semantics`, como já explicada na Seção 3.5, traduz a parte de dados (Object-Z) de uma especificação CSP-OZ em um processo CSP e o coloca em paralelo com a parte em CSP. A remoção de objetos é alcançada estendendo-se esta função de tal forma que quando o evento `terminate` ocorrer na parte de dados (OZ), a função `Semantics` deve-se comportar como o processo `SKIP`.

- **herança:** O processo da subclasse cria o processo que representa o objeto da superclasse, usando o tipo da superclasse e os valores atuais das variáveis (herdadas) da subclasse. O processo criado é então colocado em paralelo com o processo cliente através dos eventos específicos da subclasse e dos redefinidos. Desta forma, é possível criar uma hierarquia de classes que herdam propriedades e comportamentos de outra classe, e que definem novas propriedades e comportamentos específicos na subclasse. Podemos encontrar na definição da subclasse o reuso de código, pois utilizamos a própria superclasse na descrição da subclasse. Esta abordagem é um

pouco diferente da utilizada pela estratégia de verificação de modelos proposta por Fischer, porque ela mantém a identificação da superclasse e a outra não. Este trabalho não considera herança múltipla.

- **estrutura de dados polimórfica:** As estruturas de dados polimórficas são definidas através de *datatypes*. Por exemplo, o tipo Object, declarado logo abaixo, é um conjunto que contém elementos do tipo conta (Acc.{...}) e poupança (Sav.{...}). Um objeto conta é formado pelo construtor Acc seguido da tupla que representa o seu estado e um objeto poupança é formado pelo construtor Sav seguido da tupla que compõe o seu estado. Number é o conjunto de números de contas e poupanças; Real, o conjunto dos valores de saldo das contas e poupança; Passw, o conjunto de valores de senhas; e Ratio, o conjunto de taxas de juros.
  datatype Object =
  Acc.{(a,b,c) | a <- Number,b <- Real,c <- Passw } |
  Sav.{(a,b,c,d)| a <- Number,b <- Real,c <- Passw,d <- Ratio }

- **subtipos:** As relações de subtipos para formalismos orientados a objetos descrevem relacionamentos entre superclasses e subclasses, permitindo a substituição de valores dos tipos por valores de seus subtipos. Em relação aos objetos ativos com uma descrição de comportamento associada, relações de subtipos devem garantir o princípio da substitubilidade com respeito ao comportamento dinâmico das classes. O trabalho de Wehrheim [89] define padrões de subtipos, tanto para a parte estática (Object-Z) quanto para a dinâmica, que garantem tal princípio. Desta forma, qualquer classe especificada que seguir esses padrões garante o princípio da substitubilidade. O presente trabalho respeita esses padrões de subtipos e portanto garante tal princípio por construção.

- **ligações dinâmicas:** O mecanismo de ligações dinâmicas é emulado por "divisão de trabalho" entre o processo da superclasse e o da subclasse. Por exemplo, quando a classe cliente solicita a realização de um determinado evento ao processo da superclasse, ele mesmo o realiza, mas quando o cliente solicita ao processo da subclasse, encontramos duas situações:

  1. evento herdado e não redefinido: o processo da superclasse realiza este evento;
  2. evento redefinido: tanto o processo da superclasse quanto o da subclasse realizam a sua parte da tarefa em relação ao mesmo evento. Porém, devemos deixar claro que o comportamento do processo da superclasse é mantido pelo processo da sua subclasse (herança comportamental), ou seja, clientes que solicitam serviços ao processo da subclasse não devem notar alguma diferença em relação ao comportamento do processo da superclasse quando eventos da superclasse são utilizados.

No esforço de adicionar estes conceitos na estratégia de verificação de modelos para CSP-OZ, elaboramos um padrão de projeto, que será apresentado na próxima seção, o qual incorpora todas as soluções citadas anteriormente. Entretanto, apesar de Object-Z permitir herança múltipla, não tratamos essa questão no padrão. Por outro lado, apesar do foco ser em CSP-OZ, os conceitos capturados pelo padrão são gerais o suficiente para

servirem de base para o mapeamento de outras linguagens de especificação orientadas a objetos.

4.3 Descrição do Padrão de Projeto

De uma forma geral, o padrão de projeto para CSP possui a estrutura ilustrada na Figura 4.1, formada pelos seguintes elementos:

1. `procName`: variável que referencia um processo criado através da invocação da função `proc`, a qual recebe como parâmetros o tipo do processo (`objectType`) e uma tupla (`entity`) com os valores correntes para a inicialização do estado do processo. A função `proc` será detalhada posteriormente.

2. `Flow`: processo cliente que possui o fluxo de controle realizado pelo processo de referência `procName`, além de outros eventos que o permite sincronizar com o ambiente. Ele é formado por dois processos: `exit -> getStateName?y -> update.y -> SKIP`, que possui os eventos necessários para que `procName` e `Flow` possam ser finalizados com sucesso, e `Events`, que contém os eventos principais.

3. `let-within`: construtor que cria um processo resultante do paralelismo dos processos `procName` e `Flow`.

Todo processo criado possui uma referência (`procName`) associada a ele. Apesar do paralelismo ser feito entre `Flow` e `procName`, por questão de legibilidade utilizaremos o nome da classe, ao invés de sua referência.

O padrão funciona da seguinte maneira: o `procName` é um objeto (processo) passivo que oferece seus eventos ao processo `Flow`. O `Flow`, por sua vez, é um processo ativo que escolhe o evento desejado para ser realizado pelo objeto (processo).

```
let
 procName = proc(objectType, entity)
 Flow = Events
        []
        exit -> getStateName?y -> update.y -> SKIP
within (procName[|{|...|}|] Flow);main
```

Figura 4.1: estrutura do padrão de projeto

Este padrão deveria ser utilizado por uma ferramenta que recebe como entrada uma especificação em CSP-OZ e gera uma especificação em CSP_M. Porém, nada impede de utilizá-lo diretamente em CSP_M.

Apresentamos uma parte do comportamento do processo `Bank` (Seção 4.4), ilustrado através da Figura 4.2, para mostrar a aplicação do padrão de projeto. Observamos que `Bank` após realizar alguns eventos (...), passa a se comportar como o processo resultante da composição seqüencial de `(procSav [|{|...|}|] Flow)` e `main`. O primeiro processo da composição é formado pelo paralelismo dos processos `procSav` e `Flow`. O `procSav` é uma referência ao processo `Saving`, o qual é criado através da invocação da função `proc`; e o `Flow` é um processo que faz parte do fluxo de `Bank` utilizado para comunicar `Bank` e `Saving`.

Para que o processo (`procSav [|{|...|}|] Flow`) possa ser finalizado com sucesso, é necessário que `Flow` realize os eventos `exit`, para avisar ao processo `Saving` que o ambiente deseja finalizá-lo; `getStateSav`, para receber o estado de `Saving`; e `update`, para substituir o estado antigo pelo atual. Em paralelo, o `Saving`, o qual será apresentado mais adiante, também realiza os eventos `exit` e `getStateSav`, além de outros, para então ser finalizado (`SKIP`). Após a finalização dos processos `Saving` e `Flow`, `Bank` volta a oferecer os seus eventos iniciais.

```
Bank(cts,sqId) =
  ...
  main =
    ...
    (let
     e = ...
     procSav = proc(TSAVING,e)
     Flow = depositB?v -> deposit.v -> Flow
            ...
            []
            exit -> getStateSav?st2 -> update.st2 -> SKIP
  within (procSav[|{|deposit,exit,getStateSav,...|}|] Flow); main)
  ...
```

Figura 4.2: comportamento de Bank em relação à criação do processo Saving

Vamos apresentar alguns diagramas de interação para fornecer uma visão geral do comportamento entre os objetos que participam do padrão. Estes diagramas utilizam os seguintes objetos:

- *Process Collection*: processo que possui alguma coleção de entidades (*Entity*) como componente de estado.
- *Entity*: representa uma tupla com valores correntes do estado de um processo.
- *Process*: processo cuja entidade pertence à coleção de entidades do *Process Collection*.
- *SubProcess*: processo correspondente à subclasse que determina o processo *Process*.

A linguagem de padrões é composta de cinco padrões de projeto: o primeiro é o de criação de entidades, que tem a finalidade de criar objetos no formato de tuplas para que possam atualizar variáveis de estado de processos e serem passados como parâmetros de canais. O segundo é o de criação de processsos, cujo objetivo é criar processos com os valores correntes obtidos através da sua entidade. O terceiro é o de criação de sub-processos, que tem a finalidade de criar processos com herança. O quarto padrão apresenta o comportamento, de uma forma genérica, dos processos que não possuem herança e o quinto padrão, o comportamento dos processos que têm herança. Cada um dos cinco padrões possui um diagrama de interação associado.

Os diagramas de interação descritos a seguir possuem seus elementos instanciados em CSP, através do exemplo do sistema bancário apresentado na Seção 4.4. As figuras desta seção, que possuem dois diagramas, apresentam dois cenários: o primeiro (lado esquerdo), refletindo o caso de sucesso e o segundo, o de insucesso.

Começamos apresentando dois diagramas de interação, ilustrados na Figura 4.3, para mostrar a criação de entidades (*Entity*). No primeiro, temos um ator, o usuário (*User*),

que solicita ao processo *Process Collection* a criação de uma entidade (*createEntity*). Este, por sua vez, solicita a si mesmo o próximo índice disponível (*getObjInd*) e requisita a criação da nova entidade (*create*), a qual é incluída na sua coleção (*add*). No segundo, o usuário solicita a criação de uma entidade, mas o *Process Collection* não consegue um próximo índice disponível (*notGetObjInd*) e por isso avisa ao ambiente (*User*) que a entidade não pode ser criada (*message*).

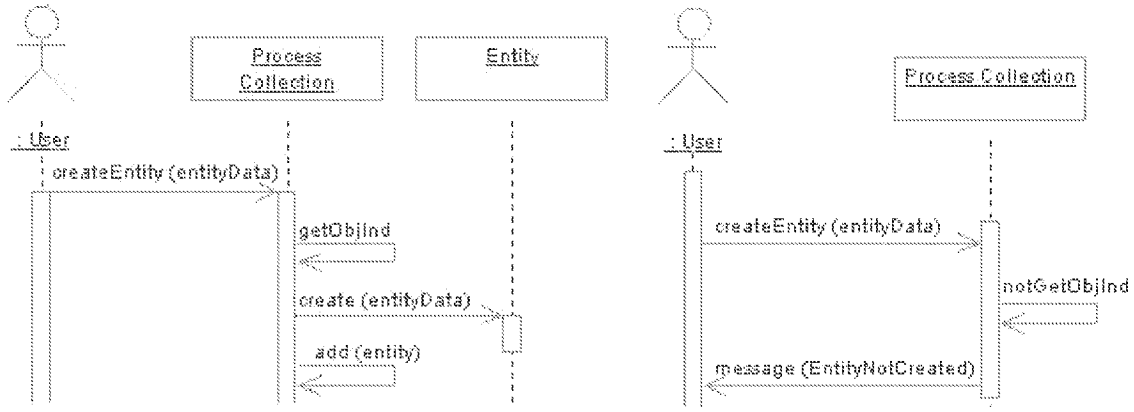


Figura 4.3: criação de entidades

Utilizamos o exemplo de criação de entidades do tipo `Account` para ilustrar a criação de entidades em CSP utilizando o padrão. O trecho da especificação, apresentada através da Figura 4.4, pertence ao fluxo de eventos do processo `Bank`.

```
main =   createAccount?p ->
         (getObjInd?ind ->
           add.(AccCtxt.(ind,0,p,0)) -> main
           []
           notGetObjInd -> message?m -> main)
        ...
```

Figura 4.4: criação de entidades do tipo Account

Observamos que o ambiente solicita a criação de uma conta (`createAccount`) ao processo `Bank`. Com o objetivo de deixar a especificação mais clara, substituimos o nome do método *createEntity* da Figura 4.3 por `createAccount` (Figura 4.4). O processo `Bank`, após realizar o evento `createAccount`, procura o próximo índice disponível. Se encontrá-lo (`getObjInd`), então cria uma nova conta representada através do *free type* `AccCtxt.(ind,0,p,0)`, adiciona ela no seu conjunto de contas (`add`), e volta a oferecer seus eventos iniciais. O canal `add` é dito ser polimórfico porque pode receber como parâmetro entidades do tipo *Account* ou *Saving*. A entidade conta é representada pelo construtor `AccCtxt.(ind,b,p,w)`. As variáveis `ind`, `b`, `p` e `w` representam o número da conta, o saldo, a senha e o valor a ser debitado, respectivamente. Como comentado na Seção 3.3.1, houve a necessidade de adicionar `w` no estado de `Account` porque a função `enable`, da estratégia de verificação de modelos (Fischer), testa as condições apenas em função dos valores das variáveis de estado.

Continuando o fluxo de `Bank`, se o próximo índice não for encontrado (`notGetObjInd`), o ambiente é avisado que a entidade não pode ser criada (`message`) e `Bank` volta a oferecer seus eventos iniciais.

A Figura 4.5 também apresenta dois diagramas de interação: o primeiro mostra como um processo sem herança, referente a uma determinada entidade (*Entity*), é criado dinamicamente. O ambiente (*User*) entra com o identificador da entidade que deseja trabalhar como processo (*enterEntity*). O processo *Process Collection* encontra a entidade que possui o identificador (*found*) e solicita a criação do processo (*create*). O segundo mostra que quando o *Process Collection* não encontra a entidade (*notFound*), então avisa ao ambiente (*User*) que a mesma não foi encontrada (*message*).

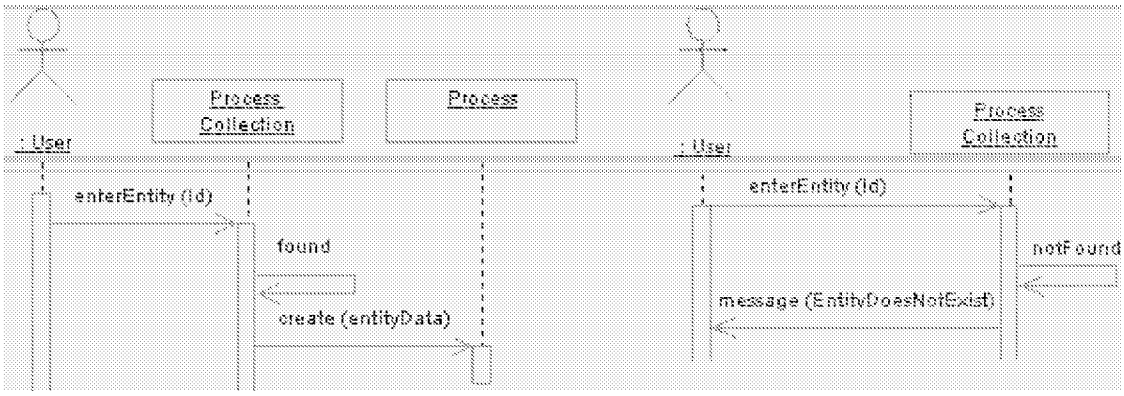


Figura 4.5: criação de processos sem herança

Utilizamos o exemplo de criação do processo `Account` para ilustrar a criação de um processo em CSP sem herança. O trecho da especificação apresentada na Figura 4.6 pertence ao fluxo de eventos do processo `Bank`. O ambiente entra com o identificador da conta (`enterAcc`), o qual é utilizado na busca da entidade que o possui como primeiro elemento da tupla. Se a conta existir (`found`), então é criado um processo `Account` aplicando a função `proc`, a qual recebe como parâmetros o tipo do objeto (`TACCOUNT`) e a tupla de variáveis de estado do processo (e), a qual é recuperada através da função `recoverTuple(obj)`. Esta função recebe como parâmetro uma entidade do tipo conta (`AccCtxt.(ind,b,p,w)`) e retorna a tupla `(ind,b,p,w)` que representa o seu estado. Se a conta não existir (`notFound`), o processo `Bank` avisa ao ambiente que a conta não existe (`message`) e volta a oferecer seus eventos iniciais. O processo `Flow` será explicado mais adiante. Como o canal `found` é polimórfico, restringimos seus valores para o conjunto `ObjectContextAcc`.

A criação dinâmica de um processo que é uma subclasse da classe *Process* é ilustrada pela Figura 4.7, a qual possui também dois diagramas de interação. Podemos observar que ela é bastante parecida com a criação de processos que não possuem herança (Figura 4.5). A diferença está na criação do sub-processo *SubProcess*, que por sua vez também solicita a criação de um processo *Process*, cujas variáveis de estado possuem valores referentes ao *SubProcess*. Isto é necessário porque os eventos, que são herdados do processo *Process*, devem ser realizados pelo próprio *Process* (divisão de trabalho).

A criação de um processo com herança em CSP é ilustrada na Figura 4.8. Verificamos que a criação do sub-processo `Saving` (Figura 4.2) cria automaticamente um processo

```
main =  ...
        enterAcc?an ->
         (found?obj:ObjectContextAcc ->
           (let
            e = recoverTuple(obj)
            procAcc = proc(TACCOUNT,e)
            Flow = ...
           within (procAcc[|{|...|}|] Flow);main)
         []
          notFound -> message?m -> main)
        ...
```

Figura 4.6: criação do processo Account

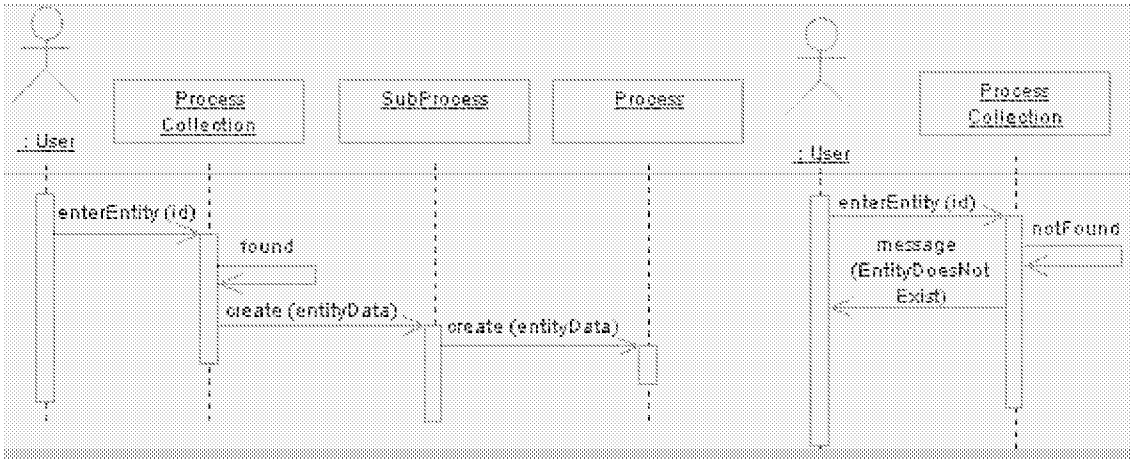


Figura 4.7: criação de sub-processos

do tipo `Account` com os valores das variáveis (herdadas) de `Saving`. O operador "\" (*hide*), que aparece no `within` mais interno, é utilizado para esconder os eventos que não devem sincronizar com o ambiente (`Bank`). A Função `Semantics` faz a tradução da parte de Z para processo e o coloca em paralelo com a parte de CSP.

```
Saving(nu,bl,ps,va,rt) =
   let
    ...
    main =
     let
       e = (nu,bl,ps,va)
       procAcc = proc(TACCOUNT,e)
       Flow = ...
     within (procAcc [|{|...|}|] Flow) \ {|...|}
   within Semantics(...)
```

Figura 4.8: comportamento do processo Saving

A partir dos processos criados, como explicado na Figuras 4.5, eles se comportam como mostra a Figura 4.9. O evento *event_i*, onde i é um número natural, representa todos os eventos oferecidos pelo *Process Collection*. O *event_i1* e o *event_in* representa a lista de eventos oferecidos pelo *Process*. Quando o *Process Collection* realiza o evento *event_i1*, obrigatoriamente, solicitará a realização de pelo menos um evento do tipo *event_in*, podendo ou não realizar o evento correspondente do tipo *retEvent_in*. Os

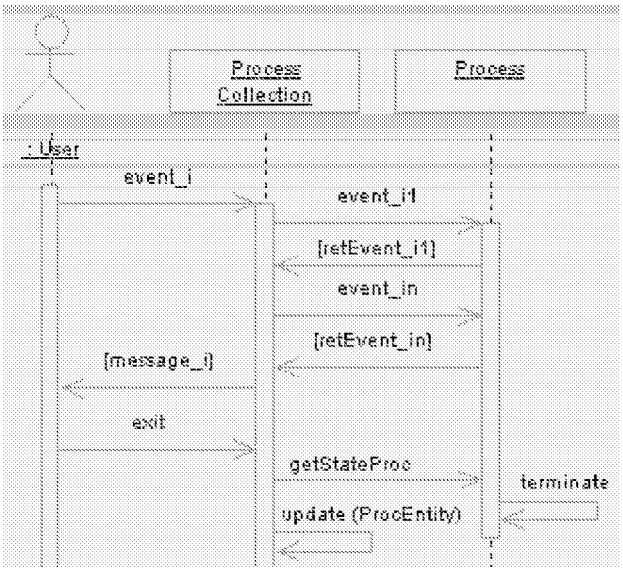


Figura 4.9: comportamento dos processos do padrão de projeto sem herança

eventos que estão entre colchetes são opcionais porque são utilizados de acordo com a necessidade.

A finalização do *Process* só ocorre depois do *Process Collection* receber do ambiente o evento *exit* e solicitar o estado ao *Process* (*getStateProc*), que após fornecê-lo finaliza (*terminate*). O *Process Collection*, por sua vez, após receber o estado, atualiza a sua coleção de entidades (*update*) e volta a oferecer os seus eventos iniciais ao ambiente.

Para ilustrar o uso do padrão sem herança, segue o exemplo, apresentado na Figura 4.10, que utiliza os processos `Bank` e `Account`, os quais são instâncias dos processos *Process Collection* e *Process*, respectivamente. Como podemos observar, `Bank` pode receber solicitações do usuário, como, por exemplo, `depositB`, `withdrawB`, `getBalanceB` e `exit`. Este, por sua vez, solicita ao processo `Account` o método equivalente à mensagem recebida pelo ambiente, com exceção do `exit`, que faz com que `Bank` solicite o estado de `Account` (`getStateAcc`) para depois atualizar a sua coleção de contas. `Account`, depois de fornecer seu estado, finaliza. Utilizamos nomes de eventos seguidos do nome ou parte do nome do processo, como por exemplo `getStateAcc`, porque torna a especificação mais clara.

As Figuras 4.11 e 4.12 apresentam a descrição em CSP_M dos processos `Bank` e `Account` utilizando o padrão de projeto (sem herança). Podemos observar na Figura 4.11 que o processo `Flow` sincroniza com o processo `Account` nos eventos `deposit`, `withdraw`, `withdrawOk`, `withdrawNotOk`, `getBalance`, `getStateAcc` e `exit`. Os outros eventos do `Flow` permitem que o mesmo sincronize com o ambiente. Depois que o `Account` finaliza juntamente com o `Flow`, o processo `Bank` volta a oferecer seus eventos iniciais `createAccount`, `enterAcc`, `createSaving` e `enterSav`. Os eventos `getBalanceDup` e `depositDup` (Figura 4.12) serão comentados mais adiante. Existe a limitação de utilizar uma única instância do processo `procAcc` por vez, como comentado na Seção 4.2.

O padrão que apresentamos a seguir permite a utilização de herança. Ele considera três situações:

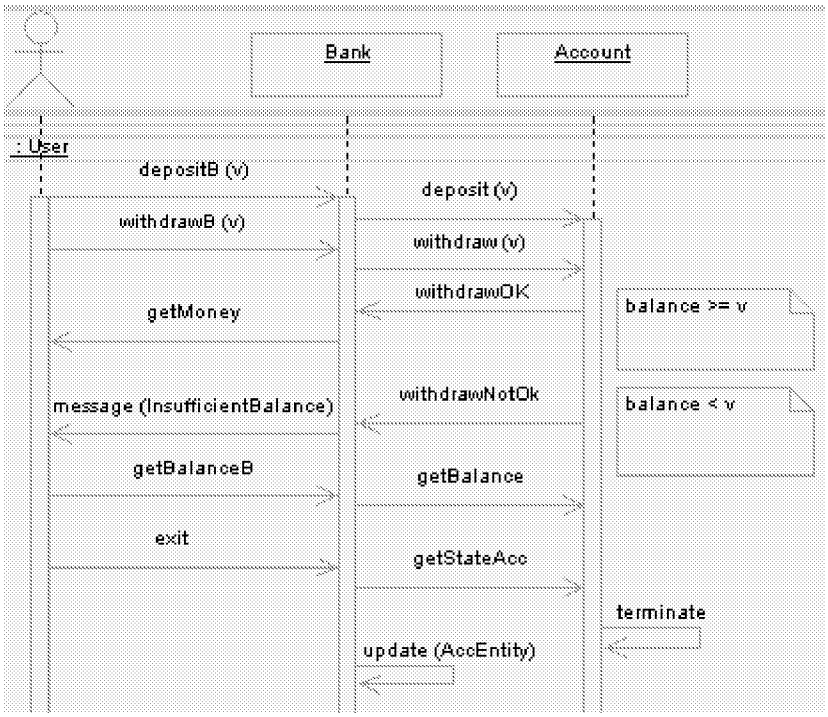


Figura 4.10: exemplo do comportamento dos processos Bank e Account

1. Eventos herdados e não redefinidos: o *Process* é quem realiza os eventos herdados pelo *SubProcess*, pois somente ele pode acessar as suas variáveis de estado. Como comentado na Seção 4.2, o *SubProcess* herda o comportamento do *Process* (herança comportamental).

2. Eventos herdados e redefinidos: o *Process* realiza o evento original e o *SubProcess* realiza o evento redefinido, pois cada um só pode ter acesso as suas variáveis de estado.

3. Eventos duplicatas: existem eventos que fazem parte do conjunto de sincronização dos processos *Process*, *SubProcess* e *Process Collection*. Porém, em determinadas situações, o *SubProcess* deseja sincronizar apenas com o *Process*. Para atingir esse objetivo foi necessária a duplicação de eventos.

A Figura 4.13 mostra o comportamento dos processos do padrão considerando herança. Existem eventos, como por exemplo *event_k*, que fazem com que o processo *Process Collection* solicite diretamente ao processo *Process* a realização de um determinado evento, como é o caso de *event_k1*. Isto acontece quando se trata de eventos herdados e não redefinidos, que devem ser realizados pelo próprio *Process*. No caso de se ter eventos redefinidos, como por exemplo *event_l*, a solicitação de realização destes eventos é feita diretamente ao *SubProcess* que os realiza e, em seguida, delega ao *Process* parte da realização dos mesmos.

Quando se deseja uma sincronização entre os processos *SubProcess* e *Process*, sem envolver o *Process Collection*, deve-se utilizar os eventos duplicatas, como está ilustrado pelo processo *SubProcess* que, após realizar o evento *event_m1*, solicita ao *Process* a

```
main =  ...
        enterAcc?an ->
         (found?obj:ObjectContextAcc ->
          (let
             e = recoverTuple(obj)
             procAcc = proc(TACCOUNT,e)
             Flow = depositB?v -> deposit.v -> Flow
                        []
                        withdrawB?v -> withdraw.v ->
                            (withdrawOk -> getMoney -> Flow
                            []
                            withdrawNotOk -> message?m -> Flow)
                        []
                        getBalanceB -> getBalance?b -> Flow
                        []
                        exit -> getStateAcc?st2 ->
                           update.(AccCtxt.st2) -> SKIP
           within (procAcc [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
                              getBalance,getStateAcc,exit|}|] Flow);main)
         []
         notFound -> message?m -> main)
        ...
```

Figura 4.11: comportamento da parte de Bank referente à interação com Account

```
main =  deposit?v -> main
        []
        withdraw?v -> (withdrawOk -> main
                          []
                          withdrawNotOk -> main)
        []
        getBalance?v -> main
        []
        exit -> getStateAcc?st -> terminate -> SKIP
        []
        getBalanceDup?v -> main
        []
        depositDup?b -> main
```

Figura 4.12: parte comportamental do processo Account

realização de eventos duplicatas (*eventDup_m1*, ..., *eventDup_mn*). Isto ocorre, tipicamente quando a realização de uma ação no *SubProcess* é descrita em termos de ações de *Process*.

Com relação à finalização de processo (remoção dinâmica de processo), o *Process Collection*, depois de receber a solicitação de *exit*, solicita o estado ao *SubProcess* que por sua vez solicita ao *Process*. Este último, após fornecer seu estado, finaliza. O *SubProcess* depois de receber o estado de *Process*, recupera o seu próprio estado, envia o estado completo para o *Process Collection* e finaliza. O *Process Collection*, após receber o estado completo atualiza a sua coleção e volta a oferecer os eventos *event_k*, *event_l*, *event_m* e *exit*.

Para ilustrar o uso de herança, utilizaremos os processos `Bank`, `Saving` e `Account`, os quais são instâncias dos processos *Process Collection*, *SubProcess* e *Process*, respectivamente. Como podemos observar através da Figura 4.14, `Bank` recebe as mesmas solicitações que as do exemplo de `Bank` e `Account` (Figura 4.10), com exceção de `interestB`, que é responsável pelo rendimento de juros da poupança. No caso das solicitações serem as mesmas, elas são repassadas para o processo `Account`, porque somente ele pode acessar as variáveis de estado que foram herdadas dele. Podemos dizer que `Saving` se

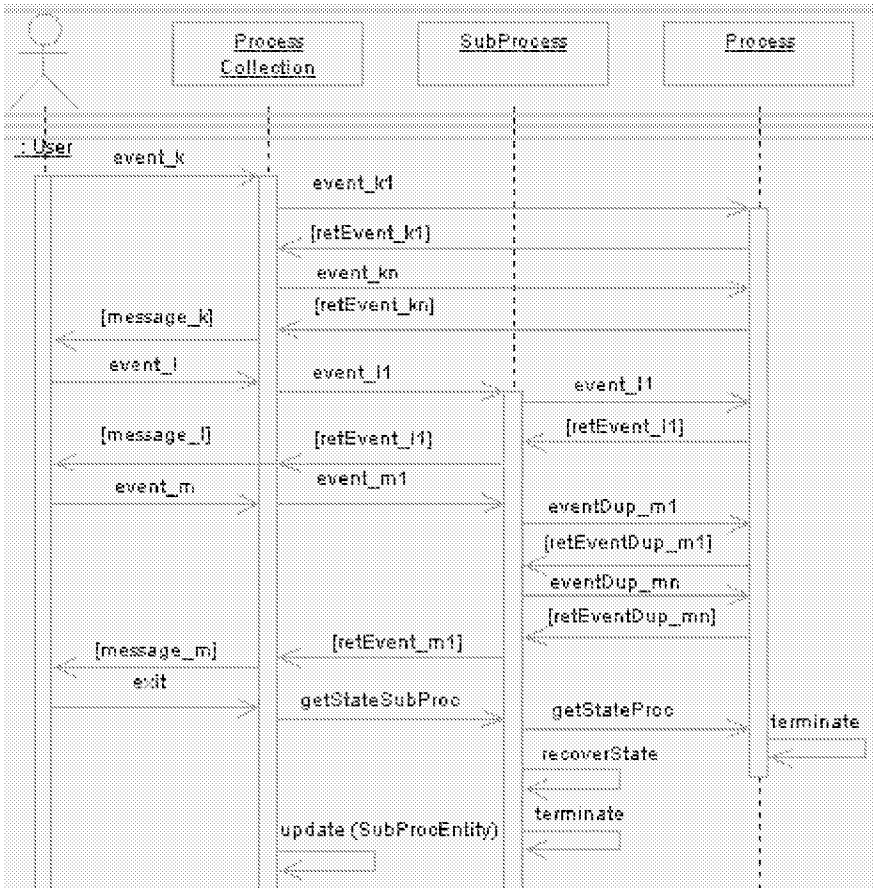


Figura 4.13: comportamento dos processos do padrão de projeto com herança

comporta como `Account` para os eventos que foram herdados de `Account`.

Com relação ao evento `interestB`, o processo `Bank`, após recebê-lo, solicita o rendimento dos juros ao processo `Saving`, através do evento `interest`. `Saving`, por sua vez, solicita ao `Account` o saldo e o depósito do rendimento, através dos respectivos eventos duplicatas `getBalanceDup` e `depositDup`. Esses últimos eventos são ditos duplicatas porque fazem exatamente o que `deposit` e `getBalance` de `Account` fazem. O motivo da duplicação é permitir uma sincronização apenas entre `Account` e `Saving`, sem envolver `Bank`. Neste exemplo, não existem eventos redefinidos. Eles serão considerados durante a explicação das outras duas abordagens.

Por fim, temos o evento `exit` que faz com que `Bank` solicite o estado de `Saving` (`getStateSav`), que por sua vez, solicita o estado de `Account` (`getStateAcc`). Este, depois de retornar seu estado, finaliza (`terminate`). `Saving`, depois de receber parte do seu estado que se encontra em `Account`, recupera o seu próprio estado (`recoverState`), retorna o estado completo para `Bank`, e finaliza (`terminate`). E `Bank`, ao receber o estado de `Saving`, atualiza a sua coleção de contas (`update`) e volta a oferecer seus eventos iniciais.

Utilizamos os exemplos dos processos `Saving` (Figura 4.15) e `Bank` (Figura 4.16) para apresentar a descrição de processos do padrão de projeto em CSP com herança. Podemos observar que o processo `Flow` de `Bank` está sincronizando com o processo

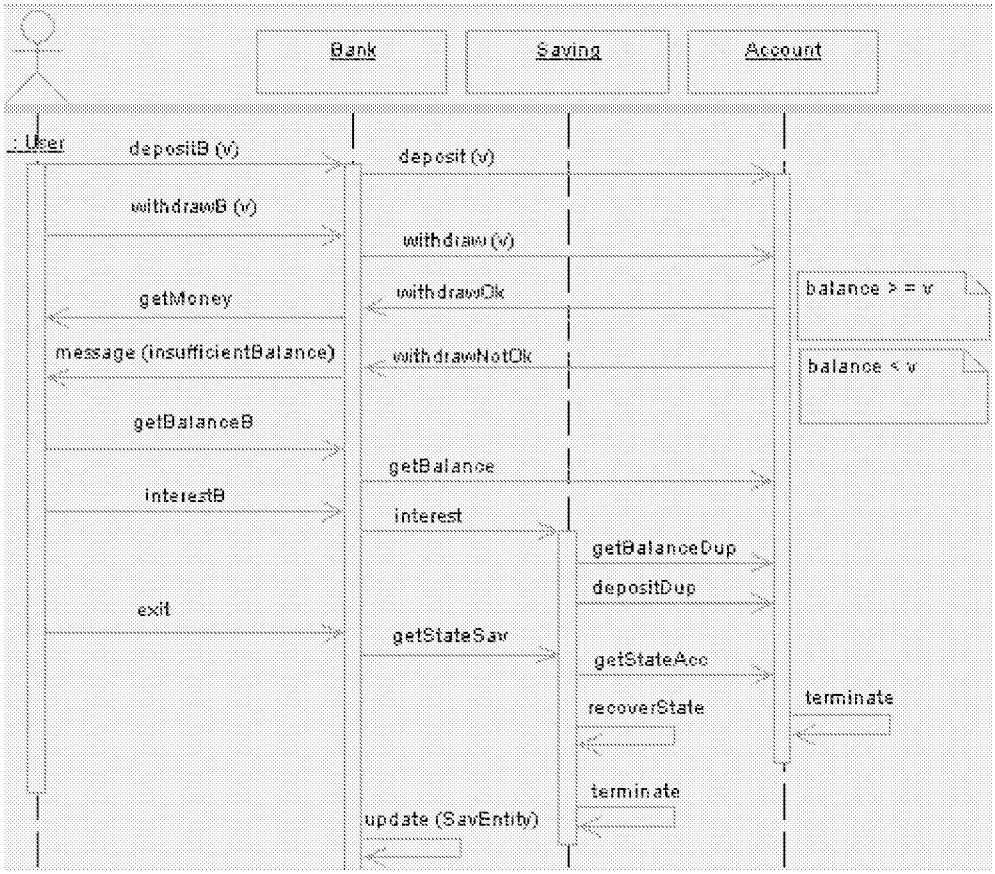


Figura 4.14: exemplo do comportamento dos processos Bank, Saving e Account

`Saving` nos eventos `deposit`, `withdraw`, `withdrawOk`, `withdrawNotOk`, `getBalance`, `interest`, `getStateSav` e `exit`. Os outros eventos de `Flow` permitem que ele sincronize com o ambiente. Os eventos duplicatas `getBalanceDup` e `depositDup` permitem que `Saving` sincronize com `Account` (Figura 4.12). Verificamos que o `main` de `Saving` é resultado do paralelismo dos processos `Account` e `Flow`. O `Flow` possui apenas os eventos próprios de `Saving`. Neste exemplo não há eventos redefinidos.

```
main =
        let
          e = (nu,bl,ps,va)
          procAcc = proc(TACCOUNT,e)
          Flow = exit -> getStateAcc?st1 -> recoverState?st ->
                      getStateSav!makeContext(st1,st) ->terminate -> SKIP
                 []
                 interest?t -> getBalanceDup?b -> depositDup.(b*t) -> Flow
        within (procAcc [|{|getStateAcc,getBalanceDup,depositDup, terminate,
                exit|}|] Flow) \ {|getBalanceDup,depositDup|}
```

Figura 4.15: parte comportamental do processo Saving

O padrão de projeto apresentado necessita de duplicação dos eventos herdados que não foram redefinidos, para não participarem do conjunto de sincronização do *Process Collection*. Além disso, os eventos duplicatas que se encontram na superclasse são

```
main =  ...
        enterSav?sn ->
         (found?obj:ObjectContextSav ->
          (let
             e = recoverTuple(obj)
             procSav = proc(TSAVING,e)
             Flow = depositB?v -> deposit.v -> Flow
                        []
                        withdrawB?v -> withdraw.v ->
                           (withdrawOk -> getMoney -> Flow
                           []
                           withdrawNotOk -> message?m -> Flow)
                        []
                        getBalanceB -> getBalance?b -> Flow
                        []
                        interestB?r -> interest.r -> Flow
                        []
                        exit -> getStateSav?st1 ->
                            update.(SavCtxt.st1) -> SKIP
          within (procSav [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
                             getBalance,interest,getStateSav,exit|}|] Flow);main)
         []
         notFound -> message?m -> main)
        ...
```

Figura 4.16: comportamento da parte de Bank referente à interação com Saving

visíveis ao ambiente, permitindo acesso indevido aos mesmos. Como não podemos torná-los invisíveis ao ambiente, porque isto os faria invisíveis também às suas especializações, então ao utilizar esse padrão deve-se ter o cuidado com os eventos duplicatas para não gerarem erros na fase de implementação. Este padrão não leva em consideração o estado comportamental do objeto quando o mesmo é inicializado.

4.4 Utilizando o Padrão

Inicialmente, especificamos o sistema bancário usando a linguagem CSP-OZ, como apresentado na Seção 3.3. Utilizamos o trabalho de Fischer [30], que propõe um método para verificar especificações em CSP-OZ via uma tradução para o dialeto de CSP (CSP_M). A especificação do sistema foi, então, convertida para CSP_M, utilizando o método estendido juntamente com o padrão, para que fosse possível usar o verificador de modelos FDR, visando analisar a especificação. Vale salientar que este processo de tradução não é totalmente automático porque nem todo predicado de lógica de primeira ordem é computável. Segue a tradução de uma parte da especificação do sistema bancário (hipotético) de CSP-OZ para CSP_M, utilizando o padrão de projeto. As especificações completas dos processos `Bank`, `Account` e `Saving` se encontram no Apêndice A.

O conjunto `ObjType` possui as constantes `TACCOUNT`, `TBANK` e `TSAVING`, que representam os tipos de objetos. Estas constantes são passadas como parâmetro da função `proc`, que é responsável pela criação de processos.

```
datatype ObjType = TACCOUNT | TBANK | TSAVING
```

Todo objeto possui um identificador próprio. `Number` representa o conjunto de identificadores para os objetos do tipo `Account` e `Saving`. Estamos utilizando conjuntos com poucos elementos para evitar a explosão de estados.

```
nametype Number ={2,3}
```

A função `proc` cria processos dinamicamente. Seus parâmetros de entrada são o tipo do processo e os valores de inicialização das variáveis de estado. Estes parâmetros determinam o processo que será ativado.

```
proc(TBANK,(cc,sa)) = Bank(cc,sa)
proc(TACCOUNT,(n,b,p,v)) = Account(n,b,p,v)
proc(TSAVING,(n,b,p,v,r)) = Saving(n,b,p,v,r)
```

A classe `Bank` utiliza os conjuntos `Real` e `Ratio` que representam os possíveis valores utilizados nas operações bancárias.

```
nametype Real = {0,1}
nametype Ratio = {0}
```

Toda conta e poupança possui uma senha de acesso. Essas senhas pertencem ao conjunto `Passw`.

```
nametype Passw = {1}
```

Na tradução, o *given set* `Message`, apresentado na Seção 3.3.1, é implementado como um *free type* (`datatype`) para que FDR possa analisar o modelo. O conjunto `Message` possui todas as mensagens utilizadas pelo processo `Bank`.

```
datatype Message =  notOkAccount | insuficientBalance | withoutReference
```

Todas as classes da especificação devem ter um conjunto de possíveis valores para o seu estado. As classes `Account` e `Saving` possuem os conjuntos `AccountContext` e `SavingContext`, respectivamente. Utilizamos o nome do processo seguido do sufixo `Context` para diferenciar do nome do processo.

```
nametype AccountContext ={(n,b,p,v) | n <- Number, b <- Real,
                          p <- Passw, v <- Real}

nametype SavingContext = {(n,b,p,v,r) |  n <- Number, b <- Real,
                         p <- Passw, v <- Real, r <- Ratio}
```

Existem canais, como, por exemplo, `update` e `add`, que podem receber objetos do tipo conta e poupança. Estes canais são do tipo `ObjectContext`, o qual contém elementos do conjunto de contas (`AccCtxt.AccountContext`) e do conjunto de poupanças (`SavCtxt.SavingContext`).

```
datatype ObjectContext = AccCtxt.AccountContext | SavCtxt.SavingContext
```

Os conjuntos `ObjectContextAcc` e `ObjectContextSav` contêm objetos do tipo conta e poupança, respectivamente.

```
nametype ObjectContextAcc = {AccCtxt.acc | acc <- AccountContext}
nametype ObjectContextSav = {SavCtxt.sav | sav <- SavingContext}
```

Segue a declaração dos canais (interface).

```
channel getObjInd:Number
channel notGetObjInd
channel createAccount:Passw
channel update:ObjectContext
channel add:ObjectContext
channel depositB:Real
channel withdrawB:Real
channel found:ObjectContext
channel notFound
channel message:Message
channel getBalanceB
channel createSaving:Passw
channel deposit,withdraw:Real
channel getBalance:Real
channel getStateAcc: AccountContext
channel getStateSav: SavingContext
channel withdrawOk,withdrawNotOk
channel getMoney
channel interestB:Ratio
channel interest:Ratio
channel enterAcc:Number
channel enterSav:Number
channel terminate
channel depositDup:Real
channel getBalanceDup:Real
channel recoverState:Ratio
channel exit
```

A seguir, apresentamos o processo `System` o qual é formado apenas pelo processo `Bank`. `Bank` é referenciado através da variável `procBank` e criado através da função `proc`, a qual recebe como parâmetros o tipo `TBANK` (constante) e uma tupla de valores (`bkState`) para a inicialização das variáveis de estado `accounts` (conjunto de contas) e `seqId` (seqüência de identificadores de conta e poupança).

```
System =  let
            bkState = ({},<2,3>)
            procBank = proc(TBANK,bkState)
          within (procBank)
```

O processo `Bank` é responsável pelas operações bancárias, como por exemplo: abrir uma conta, abrir uma poupança, creditar, debitar, e consultar saldo.

```
Bank(cts,sqId) =
```

A constante local `Ops` contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
 let
    Ops = {createAccount, enterAcc, add, getObjInd, notGetObjInd, message,
           createSaving, enterSav, found, notFound, update}
```

A constante LocOps contém apenas os nomes dos canais locais.

```
LocOps = {add,update,getObjInd,notGetIndObj,found,NotFound}
```

O processo main está particionado de acordo com o padrão de projeto apresentado na Seção 4.3. A primeira parte do main foi originada, basicamente, pela instanciação do padrão ilustrado na Figura 4.3 referente ao objeto Account, onde p representa a senha da conta, ind o número da conta (identificador), AccCtxt.(ind,0,p,0) a entidade conta com o seu estado inicializado, e m a mensagem.

```
main = createAccount?p  ->
            (getObjInd?ind ->
                 add.(AccCtxt.(ind,0,p,0)) -> main
             []
             notGetObjInd -> message?m -> main)
       []
```

A segunda parte foi originada através da instanciação do padrão ilustrado na Figura 4.5 e tem por objetivo recuperar a entidade conta de número an, e criar um processo do tipo Account, ou avisar da sua inexistência. Tivemos que restringir os valores do canal found para ObjectContextAcc, porque FDR não permitiu que o parâmetro de entrada pudesse receber tuplas de tamanhos distintos, como é o caso de conta que possui quatro elementos e poupança cinco.

```
enterAcc?an ->
 (found?obj:ObjectContextAcc ->
 (let
    e = recoverTuple(obj)
    procAcc = proc(TACCOUNT,e)
```

O processo Flow é responsável pela sincronização dos processos Bank e Account. Essa parte do main foi originada pela instanciação do padrão ilustrado na Figura 4.9, e tem como finalidade solicitar serviços ao processo do tipo Account.

```
Flow = depositB?v -> deposit.v -> Flow
       []
       withdrawB?v ->  withdraw.v ->
           (withdrawOk -> getMoney -> Flow
            []
            withdrawNotOk -> message?m -> Flow)
       []
       getBalanceB ->  getBalance?b -> Flow
       []
       exit -> getStateAcc?st2 ->
         update.(AccCtxt.st2) -> SKIP
```

O construtor let-within cria um processo que é resultado do paralelismo entre os processos Account e Flow. Isto significa que Bank, após realizar o evento found (acima), passa a se comportar como o processo procAcc [|{|...|}|] Flow, e após finalizar esse processo com sucesso, Bank volta a oferecer seus eventos iniciais.

```
 within (procAcc
         [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
         getBalance,getStateAcc,exit|}|] Flow);main)
[]
notFound -> message?m -> main)
```

O processo Saving é idêntico ao Account, exceto por instanciar os padrões para o tipo Saving, como apresentado através das Figuras 4.3, 4.7 e 4.13, e de utilizar dois eventos a mais: interestB e interest.

```
[]
createSaving?p ->
  (getObjInd?ind ->
  add.(SavCtxt.(ind,0,p,0,0)) -> main
   []
  notGetObjInd -> message?m -> main)
[]
enterSav?sn ->
 (found?obj:ObjectContextSav ->
   (let
    e = recoverTuple(obj)
    procSav = proc(TSAVING,e)
    Flow = depositB?v -> deposit.v -> Flow
            []
            withdrawB?v -> withdraw.v ->
              (withdrawOk -> getMoney -> Flow
               []
               withdrawNotOk -> message?m -> Flow)
            []
            getBalanceB ->  getBalance?b -> Flow
            []
            interestB?r -> interest.r -> Flow
            []
            exit -> getStateSav?st1 ->
             update.(SavCtxt.st1) -> SKIP
 within (procSav
         [|{|deposit,withdraw,withdrawOk,withdrawNotOk,
         getBalance,interest,getStateSav,exit|}|] Flow);
         main)
[]
notFound -> message?m -> main)
```

A constante local CSPOps contém o conjunto de nomes de canais usados pelo processo referente a parte de CSP. Neste caso, os nomes de canais são os mesmos utilizados pelo processo referente a parte de Z.

```
CSPOps = Ops
```

A constante SizeSeq define o tamanho máximo para a lista de identificadores de contas e poupanças.

```
SizeSeq = 2
```

O conjunto SeqIds é resultado da aplicação da função FSeq, a qual cria um conjunto de seqüências do tipo do primeiro argumento e cada elemento terá cardinalidade máxima dada pelo segundo argumento (maiores detalhes no Apêndice A).

```
SeqIds = FSeq({2,3},SizeSeq)
```

O estado de Bank possui os seguintes elementos: accounts, conjunto de contas e de poupanças; msg, mensagem que Bank pode apresentar ao ambiente; seqId, seqüência de identificadores utilizados na criação dos objetos conta e poupança; type, tipo do objeto utilizado; num, número de identificador de conta e de poupança. A função enable, responsável pela habilitação e desabilitação das operações, não considera as variáveis de entrada, apenas as de estado. Como certas operações precisavam ser habilitadas e desabilitadas, então tivemos que adicionar no estado o elemento type e num, os quais não são inerentes a classe. Por exemplo, a função enable aplicada a operação found (definida no Apêndice A) necessita do tipo e do identificador do objeto para verificar se ele se encontra na coleção de contas (accounts).

A função checkAccSav, a qual está definida no Apêndice A, é responsável por garantir que os números dos identificadores sejam únicos. Isto é, essa função caracteriza o invariante do processo Bank.

```
state = {(accounts,msg,seqId,type,num)|
          accounts <- Set(ObjectContext),msg <- Message,
          seqId <- SeqIds, type <- ObjType, num <- Number,
          checkAccSav(accounts,seqId)}
```

A inicialização da parte de Z do processo Bank é dada pela constante init.

```
init = {(accounts',msg',seqId',type',num') |
        (accounts',msg',seqId',type',num') <- state,
         accounts' == cts, seqId'== sqId,
         checkAccSav(accounts',seqId')}
```

Por questões didáticas, a parte de Z referente à habilitação de operações e mudanças do estado será apresentada e detalhada para uma operação. Maiores detalhes sobre o restante das operações podem ser encontrados no Apêndice A.

A parte de Z é descrita por um conjunto de funções que são utilizadas pela função Semantics (Seção 3.5). Portanto, a seguir apresentamos e explicamos as equações in, out, enable e effect, referentes ao canal update. Começamos pela função in(update) que define o conjunto de elementos de entrada para a operação update. Por esta definição, podemos ver que a operação update só pode receber como parâmetro objetos do tipo ObjectContext, ou seja, contas correntes (AccCtxt.(n,b,p,v)) e poupanças (SavCtxt.(n,b,p,v,r)).

```
in(update) = ObjectContext
```

Complementarmente, `out(update)` define o conjunto de elementos de saída. Neste caso, a operação `update` não produz qualquer saída.

```
out(update) = {{}}
```

Continuando, temos `enable(update)((accounts,msg,seqId,type,num))`, que caracteriza a habilitação e desabilitação de operações, que neste caso sempre habilita a operação `update`.

```
enable(update)((accounts,msg,seqId,type,num)) = true
```

Para finalizar, `effect(update)((accounts,msg,seqId,type,num),cs)` executa a operação `update`. Ela recebe como parâmetros a tupla de possíveis valores para o estado de `Bank` e o objeto atualizado `cs` que substituirá o antigo. O resultado da função é um conjunto com tuplas onde o primeiro elemento é o conjunto vazio, porque ela não produz valores de saída, e o segundo é a tupla de variáveis de estado atualizadas. Neste caso, somente a coleção `accounts` é atualizada; as demais permanecem inalteradas.

Para realizar a operação `update` é necessário das seguintes funções: `firstFind` para recuperar o primeiro elemento da entidade; `diff`, que faz parte de FDR, para retirar um elemento do conjunto, neste caso o objeto que tem o mesmo identificador que `cs`; `union`, que também faz parte de FDR, para adicionar o elemento atualizado; e `checkAccSav` para garantir identificadores únicos.

```
effect(update)((accounts,msg,seqId,type,num),cs) =
     {({},(accounts',msg',seqId',type',num')) |
      (accounts',msg',seqId',type',num') <- state,
      accounts'==  union(diff({ e | e <- accounts,
      firstFind(e) == firstFind(cs)},accounts),{cs}),
      msg'== msg,seqId'== seqId,type'== type,
      num'== num,checkAccSav(accounts',seqId')}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `Account` modela a conta corrente. Ele recebe como parâmetros valores para inicializar as suas variáveis de estado.

```
Account(nu,bl,ps,va) =
```

A constante local `Ops` contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
 let
    Ops = {deposit,getStateAcc,withdraw,withdrawOk,withdrawNotOk,
           getBalance,terminate,getBalanceDup,depositDup}
```

O processo `Account` não possui canais locais.

```
    LocOps = {}
```

O processo main possui dois eventos de depósito (deposit e depositDup) e dois de solicitar saldo (getBalance e getBalanceDup). Os eventos duplicatas, depositDup e getBalanceDup, só sincronizam com processos de suas subclasses; neste exemplo, com o Saving que possui o mesmo identificador de Account. Os outros dois eventos, deposit e getBalance, juntamente com os demais, sincronizam com o processo Bank (segue o padrão Figura 4.9).

```
main =  deposit?v -> main
        []
        withdraw?v -> (withdrawOk -> main
                       []
                       withdrawNotOk -> main)
        []
        getBalance?v -> main
        []
        exit -> getStateAcc?st -> terminate -> SKIP
        []
        getBalanceDup?v -> main
        []
        depositDup?b -> main
```

O estado de Account é formado pelos seguintes elementos: num que é o identificador da conta; bal, o saldo; pw, a senha; vl, o valor que se deseja debitar. Foi necessário a inclusão do atributo vl no estado para que a função enable pudesse habilitar e desabilitar as operações withdrawOk e withdrawNotOk, já que neste processo de tradução, a função enable não considera as variáveis de entrada, apenas as de estado.

A função checkAcc representa o invariante de Account e é responsável por garantir que o saldo da conta nunca fica negativo.

```
state = {(num,bal,pw,vl) | num <- Number,
         bal <- Real, pw <- Passw,vl <- Real,checkAcc(bal)}
```

A inicialização do estado de Account é dada pela seguinte constante.

```
init = {(num',bal',pw',vl') | (num',bal',pw',vl') <- state,
        num' == nu, bal' == bl, pw'== ps,vl'== va,checkAcc(bal')}
```

Vamos apresentar e explicar as equações in, out, enable e effect referentes ao canal getStateAcc. O restante das operações se encontram no Apêndice A. A função in aplicada à operação getStateAcc define o conjunto de elementos de entrada para a operação getStateAcc. Neste caso, a operação não recebe nenhuma entrada.

```
in(getStateAcc) = {{}}
```

A função out aplicada à operação getStateAcc define o conjunto dos elementos de saída para a operação getStateAcc. Neste exemplo, a operação getStateAcc produz valores do tipo AccountContext.

```
out(getStateAcc) = AccountContext
```

A condição `enable(getStateAcc)((num,bal,pw,vl))` sempre habilita a operação `getStateAcc`.

```
enable(getStateAcc)((num,bal,pw,vl)) = true
```

Para finalizar, a função `effect(getStateAcc)((num,bal,pw,vl),_)` gera como saída a tupla de variáveis (atualizadas) do estado de `Account` (`o`), a qual é um dos elementos do conjunto `AccountContext`. Vale notar que as variáveis de estado permanecem inalteradas.

```
effect(getStateAcc)((num,bal,pw,vl),_) =
 {(o,(num',bal',pw',vl')) | (num',bal',pw',vl') <- state,
   num' == num, bal'== bal, pw'== pw, o <- AccountContext,
   o == (num',bal',pw',vl'),vl'== vl, checkAcc(bal')}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `Saving` modela poupança. Como ele é uma especialização de `Account`, ao ser criado, ele próprio cria um processo `Account` com valores que fazem parte dos seus parâmetros. Os parâmetros de `Saving` servem para inicializar as variáveis de estado dos dois processos.

```
Saving(nu,bl,ps,va,rt) =
```

A constante local `Ops` contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
let
   Ops = {recoverState,interest,terminate}
```

O conjunto `LocOps` contém os nomes dos canais locais do processo `Saving`.

```
   LocOps = {recoverState}
```

O processo `main` possui a criação do processo `Account` o qual realiza os eventos herdados da classe `Account`.

```
   main =
    let
      e = (nu,bl,ps,va)
      procAcc = proc(TACCOUNT,e)
```

O processo `Flow` possui apenas os eventos específicos de `Saving`. Os eventos herdados não aparecem no `Flow` porque o seu processo `Account` é quem os realiza (segue padrão da Figura 4.13).

```
      Flow =   exit -> getStateAcc?st1 -> recoverState?st ->
                  getStateSav!makeContext(st1,st) -> terminate -> SKIP
                     []
               interest?t -> getBalanceDup?b -> depositDup.(b*t) -> Flow

    within (procAcc [|{|getStateAcc,getBalanceDup,depositDup,terminate,
                    exit|}|] Flow)\{|getBalanceDup,depositDup|}
```

O estado possui apenas um elemento, a taxa (r), porque os outros são herdados e manipulados pelo processo Account.

```
state = {r | r <- Ratio}
```

A inicialização do estado de Saving é dada pela seguinte constante.

```
init = {r' | r' <- state, r'== rt}
```

Apresentamos e explicamos as equações in, out, enable e effect referentes ao canal interest. A função in(interest) define o conjunto de elementos de entrada para a operação interest. Neste caso, a operação recebe valores do tipo Ratio como entrada.

```
in(interest) =  Ratio
```

A função out(interest) define o conjunto de elementos de saída. Neste exemplo, a operação interest não produz qualquer saída.

```
out(interest) = {{}}
```

A condição enable(interest) define que a operação interest está sempre habilitada.

```
enable(interest)(r)= true
```

Para finalizar, a função effect(interest) atualiza a variável de estado r, a qual recebe o valor da taxa corrente (ra).

```
effect(interest)(r,ra) = {({},r') | r' <- state, r'== ra}
```

A função Semantics tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

A especificação do sistema bancário descrita nesta seção mostrou que a utilização do padrão de projeto em conjunto com a estratégia de verificação de modelos estendida permitem criar especificações em CSP utilizando conceitos de orientação a objetos. Esta especificação foi verificada pela ferramenta FDR, a qual confirmou a ausência de *deadlock*. A especificação não foi testada em relação as outras propriedades.

4.5 Outras Abordagens

Propomos duas outras abordagens que também permitem implementações em CSP utilizando herança. Como o comportamento das três abordagens é o mesmo quando não é considerada herança, então vamos focar apenas na parte que trata herança para mostrar as suas diferenças. A segunda abordagem não utiliza eventos duplicatas, mas por outro lado, usa o processo *Process Collection* como um gerenciador de eventos, o qual controla todos os eventos relacionados ao processo *SubProcess*. Nesta abordagem, não acontece uma sincronização apenas entre *SubProcess* e o *Process* sem envolver o *Process Collection*, exceto no momento em que o *SubProcess* solicita os valores do estado de *Process*.

As classes do diagrama de interação da segunda abordagem são as mesmas utilizadas pelo padrão de projeto. A Figura 4.17 apresenta o comportamento da segunda abordagem, a qual utiliza o processo *Process Collection* como um gerenciador de eventos que permite utilizar classes e subclasses. Como podemos observar, o ambiente *User* solicita um serviço ao *Process Collection*, que por sua vez solicita um ou mais eventos ao processo *SubProcess* (*event_m1, event_mn*). Se o evento solicitado for um evento herdado e não redefinido, então ele será repassado ao processo *Process* (*event_l1*). Se for redefinido (*event_m2*), o *SubProcess* além de realizá-lo delega ao *Process* parte da realização do mesmo.

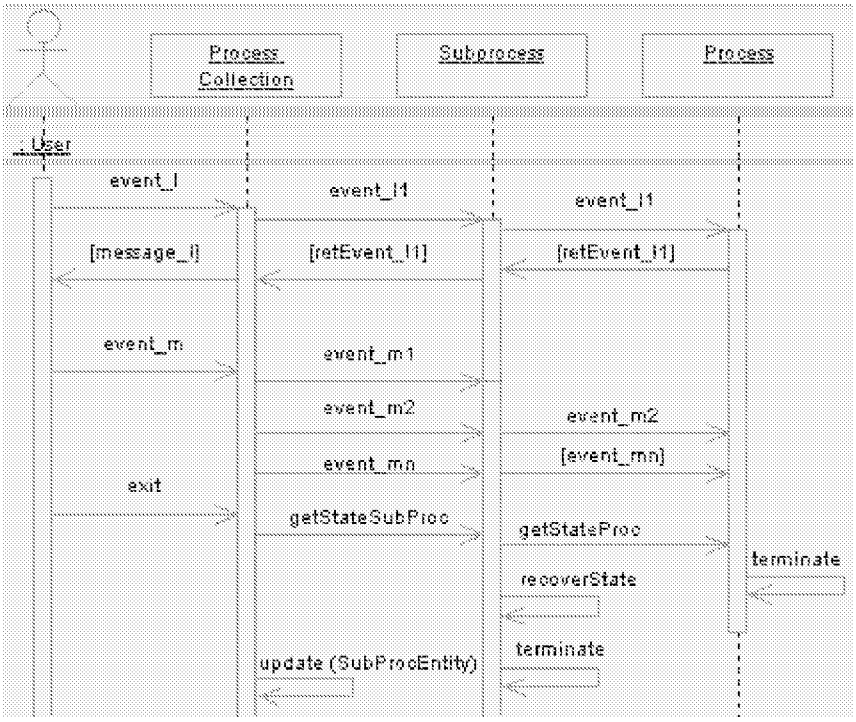


Figura 4.17: comportamento dos processos gerenciados pelo Process Collection

Para ilustrar o uso da segunda abordagem, apresentamos na Figura 4.18 os processos *Bank*, *Special Account* e *Account*, os quais são instâncias dos processos *Process Collection*, *SubProcess* e *Process*, respectivamente. Utilizamos, neste exemplo, o processo *SpecAccount* ao invés de *Saving* para mostrar a redefinição de eventos. O processo *Saving* não possui eventos redefinidos. Podemos observar que todos os eventos herdados e não redefinidos, como *withdraw(v)* e *getBalance*, são recebidos pelo processo *Special Account* e repassados para o *Account*, porque o processo responsável pela realização desses eventos é o *Account*. No caso de eventos redefinidos, como *deposit(v)*, o processo *Special Account* atualiza a sua variável de estado (bonus'= bonus + (ratio * v)) e solicita o *deposit(v)* ao processo *Account*, que também atualiza o seu estado (balance'= balance + v). Com relação aos eventos específicos de *Special Account*, como é o caso do *getBonus*, o processo *Bank*, após receber a solicitação de *bonusB*, passa a ter o controle das solicitações relacionadas ao *Special Account*. Por exemplo, *Bank* solicita os eventos *getBonus?bo*, *deposit.bo* e *resetBonus*. Com relação ao evento *exit*, a abordagem se

comporta da mesma forma que o padrão definido anteriormente.

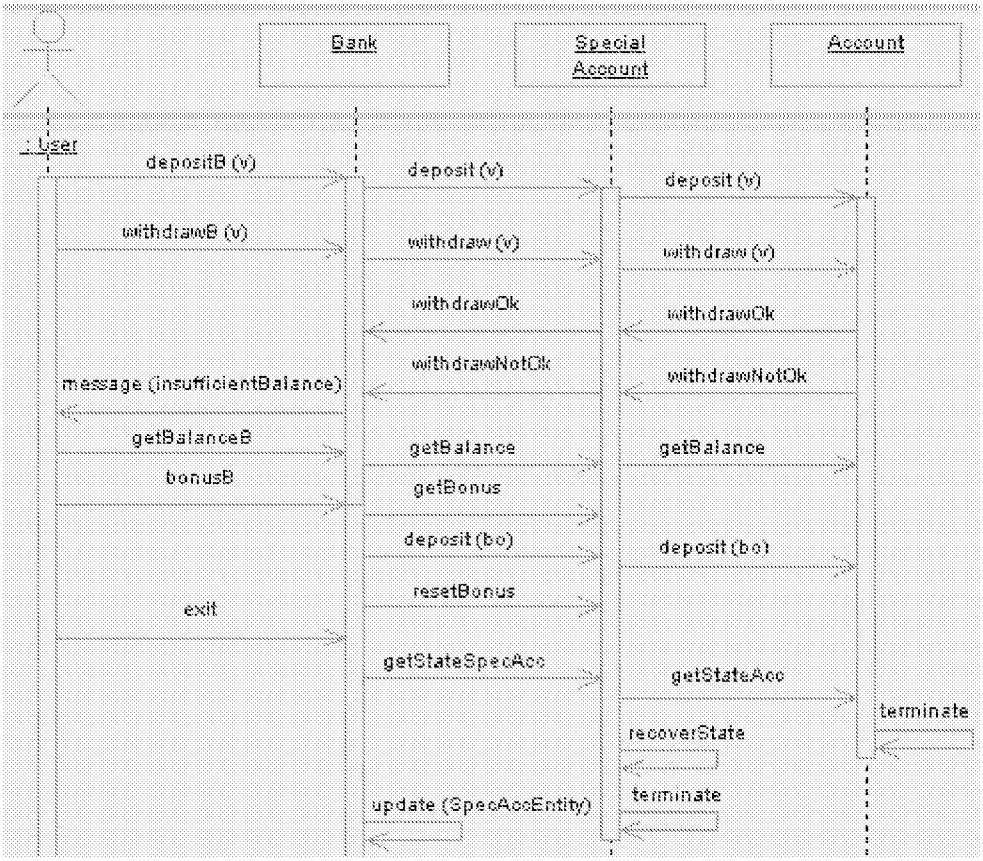


Figura 4.18: exemplo do comportamento dos processos Bank, Special Account e Account

Esta abordagem não utiliza eventos duplicatas, mas em compensação, adiciona eventos no *Process Collection* que não são inerentes a ele. Por exemplo, na Figura 4.18, o processo *Bank*, ao receber a solicitação *bonusB*, requisita *getBonus*, *deposit.bo* e *resetBonus*, que não deveriam fazer parte de *Bank* e sim de *Special Account.*

Especificamos o sistema bancário utilizando essa abordagem (Apêndice B) e usamos a ferramenta FDR que comprovou a ausência de *deadlock* na especificação.

A terceira abordagem utiliza cópia de processos para implementar herança. Ela trabalha da seguinte forma: um processo *SubProcess* que deseja sincronizar com o *Process*, sem o envolvimento do *Process Collection*, deve solicitar a criação de uma cópia do *Process* e, passar, então, a sincronizar com esta cópia. O processo cópia, por sua vez, após realizar os eventos solicitados pelo *SubProcess*, deve repassar os valores do seu estado para o *Process* original e finalizar.

O comportamento dos processos desta abordagem está apresentado na Figura 4.19. O seu diagrama de interação utiliza os seguintes objetos:

- *Process Collection*: processo que possui alguma coleção de entidades (*Entity*) como componente de estado.
- *SubProcess*: processo correspondente à subclasse que determina o processo *Original Process.*

- *Original Process*: processo cuja entidade pertence à coleção de entidades do processo *Process Collection*.

- *Copy Process*: cópia do processo *Original Process*.

Observamos que os eventos herdados e não redefinidos do processo *Original Process* (*event_k1*), quando recebidos pelo processo *SubProcess*, são repassados. Isto porque somente o processo *Original Process* pode realizá-los. No caso dos eventos serem heradados e redefinidos o *SubProcess* realiza a sua parte e delega ao *Process* parte da realização do mesmo. Quando o *SubProcess* deseja sincronizar apenas com o *Process*, então o *SubProcess* solicita ao *Original Process* os valores das variáveis do seu estado para, então, solicitar a criação de uma cópia do *Original Process* com os valores recebidos. O *SubProcess*, então, solicita ao *Copy Process* a realização de diversos eventos (*event_m11*, ..., *event_m1n*). Para finalizar o *Copy Process* é necessário que o *SubProcess* solicite ao *Copy Process* os valores das variáveis de seu estado (*getState*), e ao *Original Process* a atualização das variáveis do estado dele com os valores recebidos do *Copy Process* (*setState*). Além disso, o *Copy Process* necessita receber as mensagens *setState*, *exit*, *getStateProc* e *terminate* para ser finalizado com sucesso. O exemplo ilustrado pela Figura 4.20 mostra que os eventos *withdraw(v)* e *getBalance* são recebidos pelo *Special Account* e repassados para o *Original Account*, porque são eventos próprios dele. Como o evento *deposit(v)* é redefinido, então quando *Special Account* recebe a solicitação (*deposit(v)*), realiza a sua parte e requisita a realização de *deposit(v)* ao processo *Account*. Para que o bônus seja aplicado em *Special Account*, esta recupera o seu bônus (*getBonus*) e passa a se comportar como o processo *BonusProc*. Este processo solicita:

- ao *Original Account*, os valores das variáveis do seu estado;

- a criação do *Copy Account* inicializado com os valores recebidos do *Original Account*;

- ao *Copy Account*, o depósito do valor v e depois os valores das variáveis do seu estado atualizadas, e

- ao *Original Account*, a atualização das variáveis de seu estado.

Os eventos *setState*, *exit*, *getStateAcc* e *terminate* são realizados para que o processo *Copy Account* possa ser finalizado com sucesso. O evento *exit* recebido pelo *Bank* é o responsável pela finalização dos processos *Special Account* e *Original Account*.

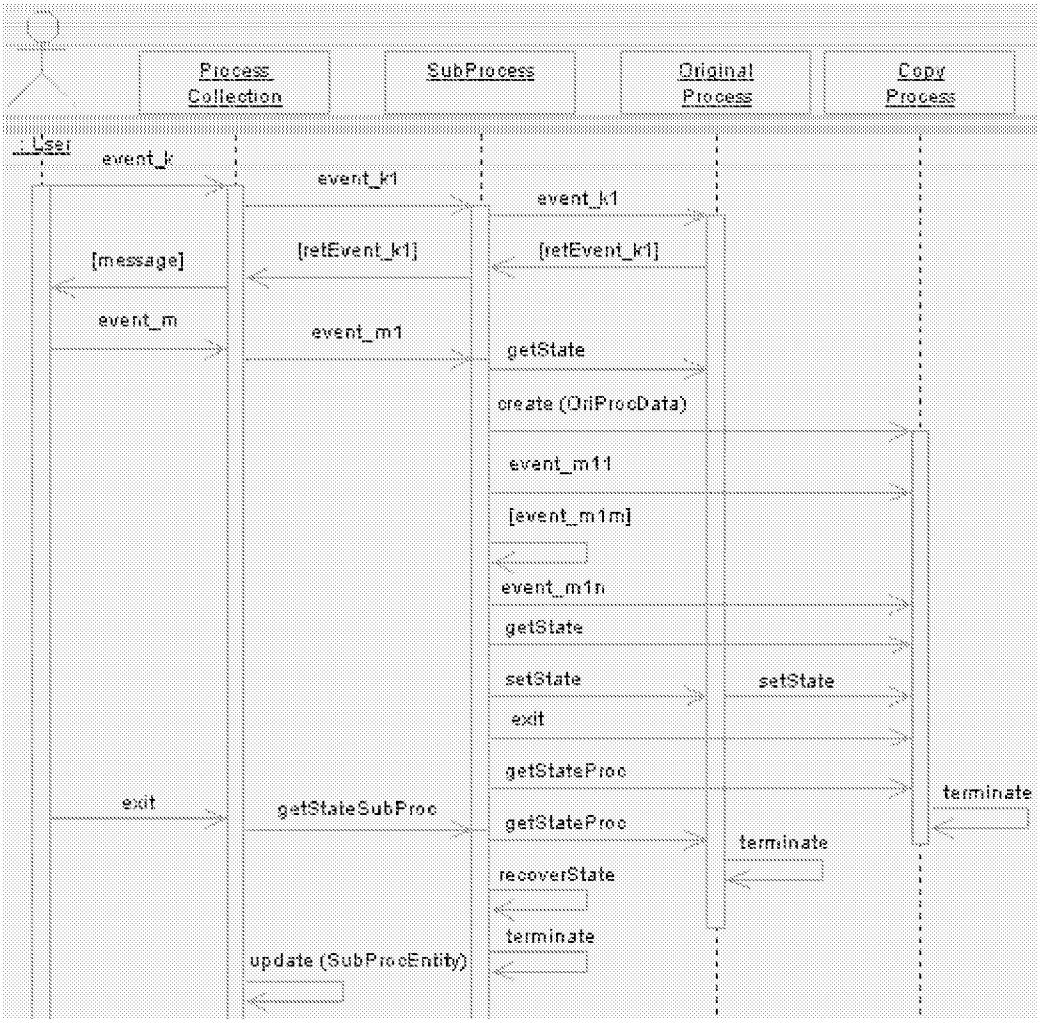


Figura 4.19: comportamento dos processos que utilizam cópia de processos

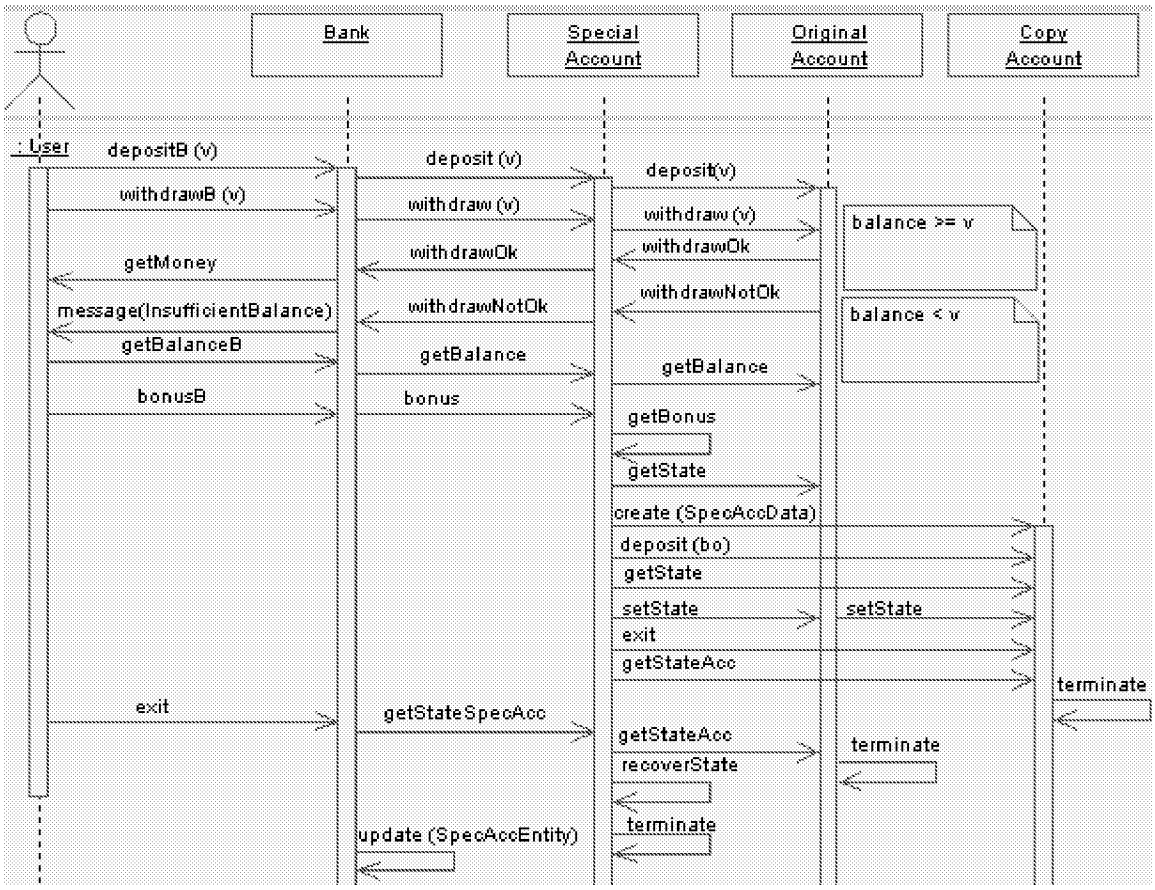


Figura 4.20: exemplo do comportamento de Bank, Special Account, Original Account e Copy Account

Essa abordagem também não utiliza eventos duplicatas, mas por outro lado, cria cópias de objetos da superclasse, ocasionando a adição de novos eventos (*getState, setState*) na superclasse, os quais também são visíveis ao ambiente. Além disso, para finalizar a cópia criada, é preciso adicionar no seu fluxo, logo após o evento *setState*, os eventos *exit*, *getStateAcc* e *terminate*, que geram também uma sobrecarga de eventos na especificação.

Especificamos o mesmo sistema bancário utilizando essa abordagem (Apêndice C) e comprovamos, através da ferramenta FDR, que o sistema está livre de *deadlock*.

4.6 Conclusões

Apresentamos, neste capítulo, uma forma de lidar com conceitos de orientação a objetos em termos da álgebra de processos CSP. Em particular, mostramos que herança, ligação dinâmica, criação e remoção dinâmica de objetos, estruturas de dados polimórficas e processos como valor, podem ser codificados em CSP. Padrões de projetos foram apresentados tanto a nível de especificação em CSP quanto em termos de diagramas de seqüência de UML, usando uma correspondência de um para um em uma abordagem pragmática.

Com o intuito de ilustrar a utilidade prática dos padrões de projeto propostos, aplicamo-os na especificação CSP-OZ do sistema bancário apresentado na Seção 3.3.1. Com esse sistema, pudemos demonstrar todos os padrões de projeto caracterizados neste capítulo.

Além disso, no decorrer da elaboração desses padrões, vislumbramos alternativas de codificação. Em conseqüência, exploramos três maneiras relativamente diferentes de solucionar o mesmo problema, ilustrando as vantagens e desvantagens.

Em trabalhos na área de semântica de linguagens de programação, a definição semântica de uma chamada de método considera dois contextos: o primeiro, quando o método é chamado pelo cliente, e o segundo quando é invocado dentro da classe ou de uma subclasse. Seguindo esta linha de trabalho, optamos pela primeira solução porque os eventos duplicatas estão relacionados a contextos distintos. Por exemplo, temos o contexto cliente (*Process Collection*), que solicita ao *SubProcess* a realização de um determinado evento e1 e o contexto subclasse (*SubProcess*), que solicita à sua superclasse (*Process*) a realização do evento e2, que é uma cópia do evento e1.

Finalmente, além de todas as contribuições apresentadas acima, o uso da abordagem proposta nesse capítulo permite a análise de propriedades de processos em CSP, em particular a ausência de *deadlock*, de sistemas baseados em orientação a objetos e concorrência.

De acordo com os nossos conhecimentos em relação a literatura atual, a presente iniciativa é inédita em termos de propor uma abordagem para modelar e analisar propriedades de sistemas orientados a objetos, utilizando uma álgebra de processos, através da representação dos objetos como processos. Na área de semântica de linguagens de programação orientadas a objetos, existem trabalhos [12, 88], que utilizam álgebra de processos para definir a semântica da linguagem.

Capítulo 5

GEHR: Modelagem em UML-RT/CSP-OZ e Análise em FDR

Este Capítulo apresenta, inicialmente, a arquitetura, em UML-RT*, de um sistema de prontuário eletrônico baseado no modelo* GEHR*. Em seguida, mostra a tradução do componente* EHRS *de* UML-RT *para processos em* CSP-OZ*, utilizando as regras de tradução apresentadas no Capítulo 3. E, por fim, apresenta a implementação, em* CSP_M*, de um processo do componente* EHRS*, utilizando o padrão de projeto apresentado no Capítulo 4; A implementação é utilizada para analisar a especificação em* FDR*.*

5.1 Introdução

Apresentaremos a seguir uma especificação formal de um subconjunto de um sistema de prontuário eletrônico baseado no GEHR, o qual foi descrito no Capítulo 2, juntamente com outros procotolos da área de saúde. Inicialmente, utilizamos diagramas de colaboração em UML-RT para expressar graficamente os processos e suas interações. Em seguida, fazemos uma tradução formal dos diagramas para processos em CSP-OZ. Para finalizar, traduzimos o processo EHRS em CSP-OZ para CSP_M, utilizando a estratégia de verificação de modelos juntamente com o padrão de projeto apresentado no capítulo anterior.

Os requisitos utilizados como base para o processo de formalização foram obtidos a partir de documentos informais, e, portanto, incompletos e ambíguos, disponíveis no endereço eletrônico www.gehr.org.

5.2 Diagramas de Colaboração em UML-RT

Os diagramas de colaboração servem para descrever a arquitetura de sistemas distribuídos, como explicado na Seção 3.2. Nós utilizamos esses diagramas para descrever a arquitetura de um sistema de prontuário eletrônico baseado no GEHR, como apresentado na Figura 5.1. Podemos observar que o sistema possui 3 processos APPLICATION que são independentes uns dos outros, e que requisitam serviços do servidor de registros eletrônicos (EHR_SERVER). Este servidor é formado pelos seguintes componentes:

- ARCHETYPES: responsável pela manutenção de arquétipos. Caso haja substituição de arquétipos, ele solicita ao processo EHRS a troca de todos os dados, que respeitavam as regras do arquétipo anterior, para adotarem as regras do novo arquétipo.

- TIME_ORDER: fornece a data e hora corrente.

- EHRS: responsável pela criação de transações e de versões de transações solicitadas pelos processos APPLICATION. Além disso, cria transações importadas e seleciona transações para serem exportadas, que são requisições do processo EHRS_EXTRACT.

- EHRS_EXTRACT: recebe solicitações de importação de transações de APPLICATION e as repassa ao prontuário eletrônico desejado. Este componente pode receber algum aviso da impossibilidade de importação das transações desejadas ou as próprias transações, que são analisadas e confirmadas pelo médico antes de serem inseridas no prontuário. O EHRS_EXTRACT também recebe solicitações de exportação de transações de outros prontuários, as quais são repassadas para o processo EHRS, que possui o registro completo de pacientes. O EHRS recupera as transações desejadas e as envia para o EHRS_EXTRACT criar o registro de exportação. Este registro é, então, enviado para o prontuário solicitante, e uma cópia do mesmo é guardada pelo prontuário fornecedor.

  Os eventos que possuem AB no final de seu nome são aqueles em que a mensagem vai do prontuário A para o B. Utilizamos essa forma de nomenclatura para facilitar o entendimento da especificação. Como estamos considerando importação

e exportação de transações do prontuário A, isto significa que o processo enviará mensagens com AB no final de seu nome e receberá mensagens com BA.

O processo APPLICATION se comunica com o processo ARCHETYPES quando necessita recuperar determinados arquétipos para fazer as validações dos novos dados, que serão inseridos no registro de paciente através das transações. Ele também se comunica com o processo EHRS para solicitar criação e correção de transações, e com EHRS_EXTRACT para requisitar importação de versões de transações de outro prontuário eletrônico.

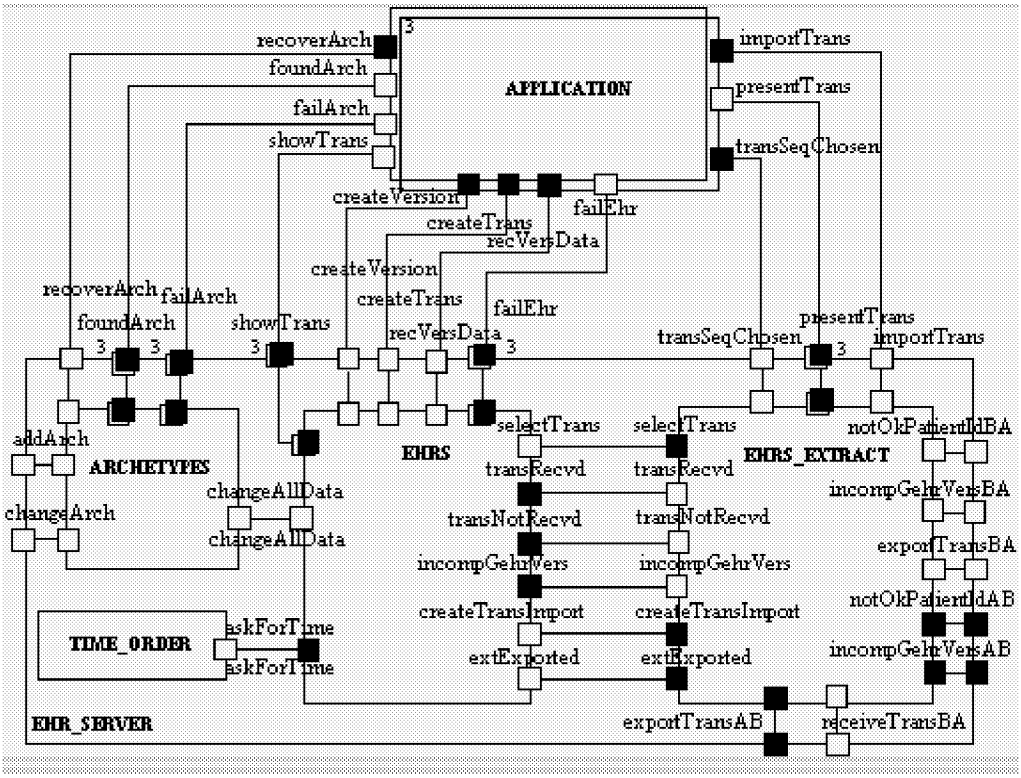


Figura 5.1: Arquitetura de um sistema de prontuário eletrônico baseado no GEHR

Apesar da tradução de diagramas de colaboração em UML-RT para processos CSP-OZ utilizar nomes para portas e conectores, vamos usar apenas os nomes das portas para não comprometer a legibilidade dos diagramas de colaboração. Diante desse fato, não será necessário fazer renomeações de portas para conectores e vice-versa, como está apresentado no processo de tradução da Seção 3.4. O conjunto de equações de processos correspondente à descrição gráfica da arquitetura é construído indutivamente. Os nomes de processos e de parâmetros de processos, para todas as cápsulas básicas, devem ser fixados. Para cada nome de cápsula básica deve existir um processo correspondente em CSP-OZ. De acordo com a função de tradução τ(BC), descrita na Seção 3.4, as cápsulas básicas ARCHETYPES, TIME_ORDER, EHRS e EHRS_EXTRACT são traduzidas para os processos ARCHETYPES($Adr_{foundArch}$,$Adr_{failArch}$), TIME_ORDER, EHRS($Adr_{failEhr}$,$Adr_{showTrans}$) e EHRS_EXTRACT ($Adr_{presentTrans}$), respectivamente. O ARCHETYPES possui como parâmetro dois conjuntos de endereços, $Adr_{foundArch}$ e $Adr_{failArch}$,

porque ele pode se comunicar com um dos três processos APPLICATION através dos eventos *foundArch* e *failArch*. O mesmo acontece com o processo EHRS que se comunica com os processos APPLICATION através dos conjuntos de endereços $Adr_{failEhr}$ e $Adr_{showTrans}$. Para finalizar, o processo EHRS_EXTRACT se comunica com os APPLICATION através do conjunto de endereços $Adr_{presentTrans}$.

O próximo passo é a definição da equação de processo para a cápsula composta EHR_SERVER. Utilizando a função τ(CC)(Seção 3.4), obtemos o seguinte processo:

EHR_SERVER $(Adr_{foundArch}, Adr_{failArch}, Adr_{presentTrans}, Adr_{failEhr}, Adr_{showTrans}) =$
$(((\text{ARCHETYPES } (Adr_{foundArch}, Adr_{failArch}))_{A_{Arch}} \parallel_{A_{Ehr}}$
$\text{EHRS } (Adr_{failEhr}, Adr_{showTrans}))_{A_{Arch} \cup A_{Ehr}} \parallel_{A_{Time}}$
$\text{TIME\_ORDER})_{A_{Arch} \cup A_{Ehr} \cup A_{time}} \parallel_{A_{Ext}}$
$\text{EHRS\_EXTRACT } (Adr_{presentTrans}))) \setminus H_{ehr\_server}$

Os conjuntos A_{Arch}, A_{Ehr}, A_{Time} e A_{Ext} contêm todos os nomes de conectores ligados às cápsulas; neste caso, são iguais aos nomes das portas correspondentes. O conjunto H_{ehr_server} = {*changeAllData, askForTime, selectTrans, incompGehrVers, transNotRecvd, transRecvd, createTransImport, extExported*} possui todos os eventos escondidos de EHR_SERVER.

O próximo passo é a definição da equação de processo para a cápsula múltipla MApplic, que agrega as três instâncias do processo APPLICATION. Utilizando a função τ(MC)(Seção 3.4) obtemos o seguinte processo:

$\text{MApplic} = |||_{in:\{1,2,3\}} \bullet \text{APPLICATION}[c_j \mapsto c_j.in]$

onde $[c_j \mapsto c_j.in]$ representa a renomeação de todos os canais do processo APPLICATION. Por exemplo, *createTrans* é renomeado para *createTrans.in*, *correctTrans* para *correctTrans.in*, e assim por diante.

Finalmente, podemos fornecer a descrição do sistema aplicando mais uma vez o esquema de tradução para cápsulas compostas (τ(CC)):

GERH_SYSTEM = MApplic
$A_{MApplic} \parallel_{A_{ehr\_server}}$
EHR_SERVER ({1,2,3},{1,2,3},{1,2,3},{1,2,3},{1,2,3})

onde os conjuntos $A_{MApplic}$ e A_{ehr_server} possuem os nomes dos conectores que, neste caso, são iguais aos nomes das portas correspondentes, ligados à cápsula múltipla MApplic e a cápsula composta EHR_SERVER, respectivamente. O processo EHR_SERVER possui como parâmetro cinco conjuntos instanciados com {1,2,3}, os quais possuem os endereços das instâncias de APPLICATION. Cada um destes conjuntos está associado a um evento que comunica o processo EHR_SERVER com uma das três instâncias de APPLICATION.

5.3 Especificação em CSP-OZ

A apresentação da especificação formal de todos os componentes descritos na seção anterior torna a documentação extensa para os fins desta dissertação. Por concisão, vamos considerar apenas o processo EHRS, que é um dos principais componentes do EHR_SERVER. Vamos apresentar a sua especificação em CSP-OZ, e em seguida a sua tradução para CSP_M, utilizando a técnica de verificação de modelos estendida.

Antes de apresentar a especificação formal do componente EHRS, vamos apresentar

uma visão estática do mesmo, através do diagrama de classes da Figura 5.2. Os detalhes de cada classe serão apresentados na próxima seção. O diagrama de classes apresentado não faz parte da estratégia de modelagem. Ele tem a finalidade de guiar o leitor na leitura da especificação formal.

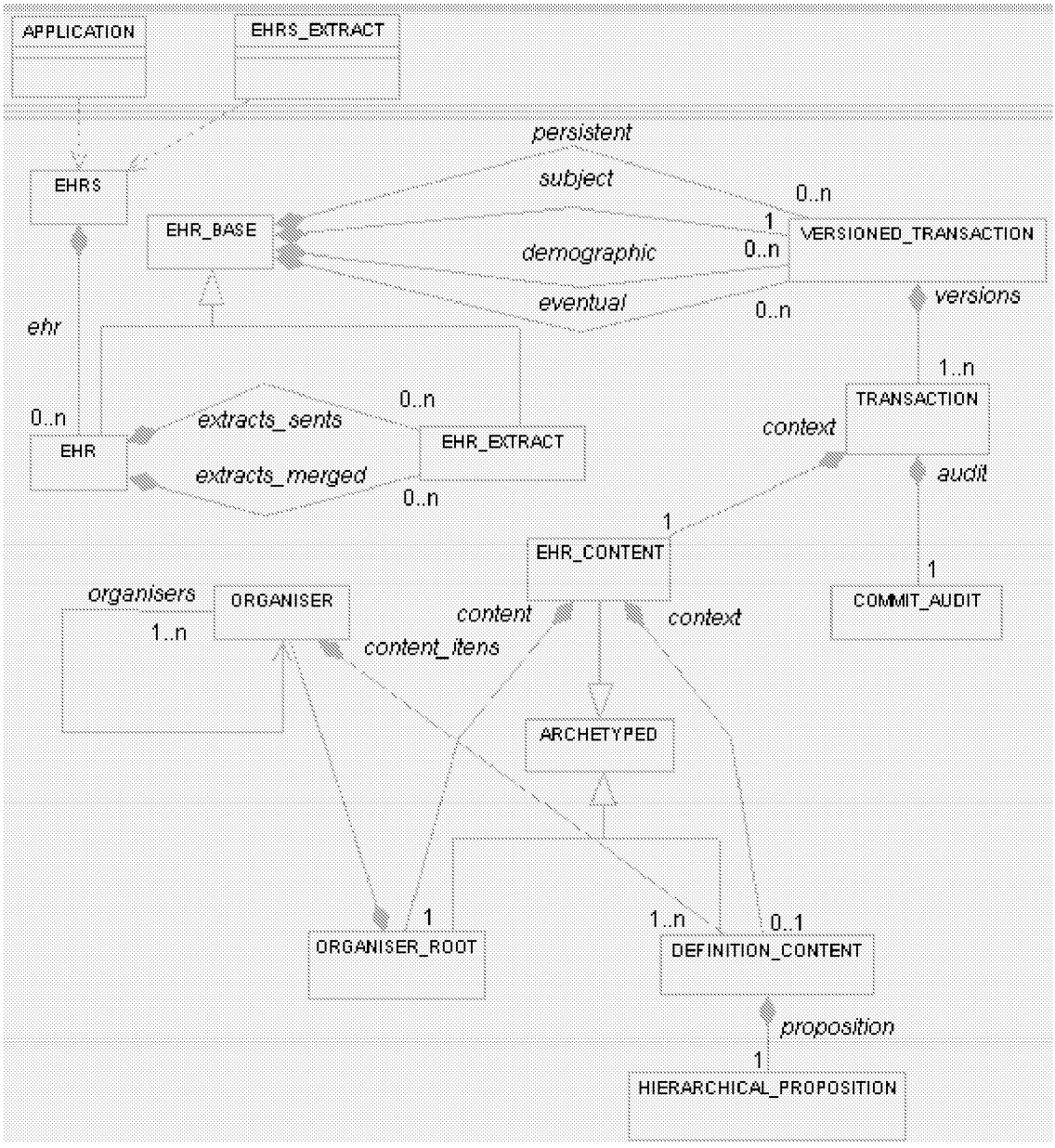


Figura 5.2: Diagrama de classes do componente EHRS

As classes APPLICATION e EHRS_EXTRACT solicitam serviços fornecidos pela classe EHRS, como, por exemplo, criar transações, importar transações, etc. A classe EHRS é composta de registros de paciente, os quais são instâncias da classe EHR.

A superclasse EHR_BASE reúne informações comuns às classes EHR e EHR_EXTRACT. Ela é composta por quatro transações versionadas, as quais estão representadas através das agregações *subject*, *demographic*, *persistent* e *eventual*. A classe EHR é formada

pelas agregações *extracts_sents* (registros exportados) e *extracts_merged* (registros importados), além das herdadas de EHR_BASE. Já a classe EHR_EXTRACT, que modela o registro de importação e exportação, é formada apenas pelas agregações herdadas de EHR_BASE.

A classe VERSIONED_TRANSACTION modela as transações versionadas. Ela é parte da classe EHR_BASE, como comentado acima, e é formada de versões como expressa a associação *versions*. A classe TRANSACTION, a qual modela as versões da transação, é composta pelos tipos EHR_CONTENT e COMMIT_AUDIT.

A superclasse ARCHETYPED reúne informações comuns às classes EHR_CONTENT, ORGANISER_ROOT e DEFINITION_CONTENT. A classe EHR_CONTENT modela o conteúdo das informações do paciente, obtidas através de uma versão. Ela é composta pelas partes ORGANISER_ROOT e DEFINITION_CONTENT. A classe ORGANISER_ROOT é a organizadora de nível topo de uma hierarquia de organizadores (ORGANISER) e a DEFINITION_CONTENT modela um item de conteúdo (dados do paciente), o qual pode estar organizado em diversos formatos (tabela, lista, etc). Estes formatos estão modelados através da classe HIERACHICAL_PROPOSITION.

5.3.1 Tipos de Dados

Vamos começar apresentando o conjunto TOPIC, que possui os assuntos das transações criadas pelos médicos. Estes profissionais interagem com o sistema através de transações, e cada uma delas se refere a algum assunto ou tópico.

TOPIC::= patientIdentity | HCP | HCF | problemList |
familyHistory | contact | testResults | Others

O registro de paciente (EHR) inclui quatro tipos transações: *subject*, cujo tópico pode ser apenas *patientIdentity*, e que contém os dados cadastrais de paciente; *demographic*, que possui transações demográficas, tais como endereços de médicos (HCP) e de hospitais (HCF); *persistent*, que possui transações válidas para toda a vida do paciente, como, por exemplo, a lista de problemas (*problemList*) e a história familiar (*familyHistory*); e *eventual*, que possui transações válidas durante um determinado período, como, por exemplo, resultado de exame (*testResults*) e consulta (*contact*). O conjunto TRANSTYPE contém estes tipos de transações.

TRANSTYPE::= subject | demographic | persistent | eventual

Os motivos de falhas estão representados através do conjunto PURPOSE.

PURPOSE::= RecordRecovered | RecordNotRecovered | WithoutReference

A comunicação entre os sistemas de prontuário eletrônico é feita através de importação e exportação de transações. O conjunto EXT_COMM possui os tipos de comunicação.

EXT_COMM::= imp | exp

A especificação da classe EHRS utiliza uma grande variedade de tipos. Como estamos trabalhando em um nível abstrato, vários desses tipos estão sendo considerados como *given sets* porque não há necessidade de detalhar a sua estrutura interna. Além disso, renomeamos alguns nomes de tipos utilizados no modelo GEHR para não comprometer a legibilidade das classes CSP-OZ. Segue a definição dos *given sets* com os seus significados.

- HCA - conjunto de médicos.
- SRCE - conjunto de identificadores de prontuários.
- GEHR_ARCHID - conjunto de identificadores de arquétipos do sistema GEHR.
- TEXT - conjunto de textos.
- DATE_TIME - conjunto de data e hora.

[HCA, SRCE, DATE_TIME, GEHR_ARCHID, TEXT]

O conjunto EHRID representa os identificadores de registros de paciente.

$$\text{EHRID} == \mathbb{N}$$

Cada elemento do conjunto PERMISS determina o tipo do usuário que tem acesso a uma determinada transação. Segue a descrição dos elementos desse conjunto:

Perm_patient - Paciente
Perm_hcp_legally_responsible - Médico responsável pelo paciente
Perm_hcp_authorising - Médico responsável pela criação da transação
Perm_any_hcp - Qualquer médico
Perm_any_staff - Staff
Perm_anyone - Qualquer pessoa

$$\text{PERMIS} ::= \text{Perm\_patient} \mid \text{Perm\_hcp\_legally\_responsible} \mid \text{Perm\_hcp\_authorising} \mid \text{Perm\_any\_hcp} \mid \text{Perm\_any\_staff} \mid \text{Perm\_anyone}$$

O conjunto PERMISSET contém subconjuntos do conjunto PERMIS, que definem os possíveis tipos de usuários que têm acesso a uma transação.

$$\mid \text{PERMISSET}: \mathbb{P}\ \text{PERMIS}$$

Para facilitar o trabalho dos médicos, algumas organizações de saúde criaram codificadores, como, por exemplo, ICD-10 [43], SNOMED [44], etc, os quais codificam procedimentos médicos, doenças, laudos, materiais, exames, etc. O conjunto CODSYS contém os codificadores utilizados pelos médicos.

$$\text{CODSYS} = \{\text{ICD-10}, \text{ICPC}, \text{SNOMED}, \text{UMLS}\}$$

Quando se cria uma transação, deve-se informar o conjunto de codificadores utilizados na criação do conteúdo clínico. Esse conjunto é um dos elementos do conjunto CODSYSSET.

| CODSYSSET: $\mathbb{P}$ CODSYS

Como existem várias aplicações (APPLICATION) rodando ao mesmo tempo, para acessá-las é necessário o seu endereço. O conjunto APPLIC contém os endereços dessas aplicações.

APPLIC = {1,2,3}

Cada tipo de transação trabalha com um conjunto de conceitos. Estes conjuntos possuem uma grande quantidade de elementos. Por simplicidade, utilizaremos apenas alguns deles. O conjunto Concept_Subject_Set possui um único elemento, o conceito "patientId", que representa a identificação do paciente. O Concept_Demographic_Set contém "address", que representa endereços de profissionais e de entidades de saúde. O Concept_Persistent_Set possui dois conceitos, "bloodPressure", que representa a pressão sanguínea, e "headache", o sintoma dor de cabeça. Finalmente, o Concept_Eventual_Set possui também dois conceitos, "haemogram", que representa o exame hemograma e "consultation", a consulta médica. Segue a descrição destes conjuntos:

Concept_Subject_Set::= patientId
Concept_Demographic_Set::= address
Concept_Persistent_Set::= bloodPressure | headache
Concept_Eventual_Set::= haemogram | consultation

A especificação utiliza vários tipos que nós descrevemos através de esquemas. Segue a definição de cada um. TRANSD reúne dados necessários para a criação de uma nova transação. Ele possui os seguintes elementos: *sr*, identificador do prontuário fonte; *hca*, médico responsável pela criação da transação; *ehr_id*, identificador do registro de paciente; *tt*, tipo de transação; *ac*, conjunto de usuários que possui acesso a leitura das versões da transação; *am*, conjunto de usuários que possui direito de criação de versões da transação; *rs*, motivo da criação da transação; *gv*, versão do GEHR; *cs*, conjunto de codificadores utilizados na transação; *vid*, identificador da transação.

TRANSD

$sr : SRCE$
$hca : HCA$
$ehr\_id : \mathbb{N}$
$tt : TRANSTYPE$
$ac : PERMISSET$
$am : PERMISSET$
$rs : TEXT$
$gv : \mathbb{N}$
$cs : CODSYSSET$
$vid : DATE\_TIME$

VERSD reúne dados para a criação de uma nova versão de transação. Ele possui os seguintes elementos: *sr*, identificador do prontuário fonte; *hca*, médico responsável pela

criação da versão; *ehr_id*, identificador do registro de paciente; *tt*, tipo de transação; *rs*, motivo da criação da versão; *vid*, identificador da transação; *cs*, conjunto de codificadores utilizados nesta versão.

VERSD

$sr : SRCE$
$hca : HCA$
$ehr\_id : \mathbb{N}$
$tt : TRANSTYPE$
$rs : TEXT$
$vid : DATE\_TIME$
$cs : CODSYSSET$

TRANSIMPD reúne dados de todas as transações que se deseja importar. Ele possui os seguintes elementos: *hca*, médico responsável pela criação da versão ou da transação caso não exista no registro de paciente; *ehr_id*, identificador do registro de paciente; *rs*, motivo da criação da versão ou transação; *ex*, registro de importação; *ts*, lista de transações de outro prontuário para serem selecionadas e importadas pelo médico.

TRANSIMPD

$hca : HCA$
$ehr\_id : \mathbb{N}$
$tp : TOPIC$
$rs : TEXT$
$ex : EHR\_EXT$
$ts : \text{seq } VERS\_TRANS$

TRANSIMPORTED reúne dados de cada transação a ser importada. Ele possui os seguintes elementos: *sr*, identificador do prontuário fonte da informação; *tg*, identificador do prontuário receptor da informação; *hca*, médico responsável pela criação da versão ou da transação; *rs*, motivo da criação da transação; *tr*, transação importada.

TRANSIMPORTED

$sr : SRCE$
$tg : SRCE$
$hca : HCA$
$rs : TEXT$
$tr : VERS\_TRANS$

Além dos tipos citados acima, a especificação do processo EHRS utiliza várias classes que são apresentadas logo a seguir. Para facilitar a leitura, apresentamos as principais classes. As outras classes estão definidas no apêndice D.

5.3.2 Transações e Versões de Transações

A classe TRANS modela uma versão de conteúdo de uma transação (VERS_TRANS), que corresponde à ação de um médico entrar com uma nova informação no registro do paciente. A classe VERS_TRANS está definida logo a seguir. Uma nova versão (TRANS) é

gerada a partir da criação ou modificação de uma VERS_TRANS. Ela possui os atributos *content* e *audit*, que representam o conteúdo da versão e a informação de auditoria, respectivamente. As definições das classes `EHR_CONT` e `COMMIT_AUDIT` se encontram no apêndice D.

$$
\begin{array}{|l}
\textit{TRANS}(\textit{cont} : \textit{EHR\_CONT};\ \textit{aud} : \textit{COMMIT\_AUDIT}) \\
\hline
\textit{content} : \textit{EHR\_CONT} \\
\textit{audit} : \textit{COMMIT\_AUDIT} \\
\hline
\textit{INIT} \\
\textit{content} = \textit{cont} \land \textit{audit} = \textit{aud} \\
\hline
\end{array}
$$

A classe VERS_TRANS modela a transação lógica cuja informação existe dentro de versões, que correspondem às diferentes vezes em que a informação foi gerada. Ela é responsável por agrupar e controlar as versões. A classe possui os seguintes atributos:

- *access_rights*: conjunto de tipos de usuários que têm direito a leitura das versões da transação.
- *access_amend*: conjunto de tipos de usuários que têm direito de criar versões na transação.
- *time_created*: data e hora de criação.
- *gehr_version*: versão do GEHR utilizada pelo prontuário.
- *versions*: lista de versões (TRANS).

Esta classe possui um invariante que garante a existência de pelo menos uma versão (TRANS). Além disso, ela possui o evento *addVersion*, que adiciona uma versão na lista.

$$
\begin{array}{|l}
\textit{VERS\_TRANS}(\textit{acc}, \textit{amd} : \mathbb{P}\ \textit{PERMIS};\ \textit{tm} : \textit{DATE\_TIME};\ \textit{gv} : \mathbb{N};\ \textit{v1} : \textit{TRANS}) \\
\textit{method}\ \textit{addVersion} : [\textit{v?} : \textit{TRANS}] \\
\textit{main} = \textit{addVersion?v} \rightarrow \textit{SKIP} \\
\hline
\textit{access\_rights} : \mathbb{P}\ \textit{PERMIS} \\
\textit{access\_amend} : \mathbb{P}\ \textit{PERMIS} \\
\textit{time\_created} : \textit{DATE\_TIME} \\
\textit{gehr\_version} : \mathbb{N} \\
\textit{versions} : \text{seq}\ \textit{TRANS} \\
\hline
\textit{versions} \neq \varnothing \\
\hline
\end{array}
$$

INIT

$access\_rights' = acc \land access\_amend' = amd$

$time\_created' = tm \land gehr\_version' = gv$

$versions' = \langle v1 \rangle$

effect_addVersion

$\Delta(versions)$

$v? : TRANS$

$versions' = versions \frown \langle v? \rangle$

5.3.3 Registros

A classe EHR_BASE é a base comum de informação para as classes EHR e EHR_EXT que serão explicadas mais adiante. Ela possui na sua interface vários canais locais (*local_chan*), os quais são utilizados para a comunicação interna. Estes canais não são visíveis pelo ambiente.

O comportamento de EHR_BASE está descrito através de dois fluxos. Nesta seção vamos apresentar apenas o primeiro porque ele foi escolhido para participar do processo de tradução de CSP-OZ para CSP_M, utilizando a técnica de Fischer junto com o nosso padrão. A definição completa da classe se encontra no apêndice D. O processo recebe a solicitação de criação de uma transação (*createTrans1*), recupera o tipo da transação (*getTransType*), solicita o próximo tempo (*askForTime*) e cria um objeto (c) do tipo COMMIT_AUDIT, que possui as informações de auditoria, como, por exemplo, o médico responsável pelas informações adicionadas na transação. Depois, ele cria um objeto (v) do tipo TRANS (versão da transação), o qual possui como parâmetros os objetos de auditoria (c) e a informação clínica do paciente (d). De acordo com o tipo da transação informado, o processo pode seguir um dos dois fluxos abaixo.

- Se o tipo de transação é igual a demographic ou eventual, ele cria uma transação, a adiciona no registro (*addTransaction*) e finaliza o processo.
- Se o tipo de transação é igual a persistent, ele solicita a recuperação da transação (*recPersisTrans*). Se encontrar a transação, adiciona a versão na transação (*addVersion*), substitui a transação antiga pela nova (*replaceTrans*) e finaliza o processo. Por outro lado, se não encontrar a transação, cria uma nova transação, a adiciona no registro (*addTransaction*) e finaliza o processo.

$$
\begin{array}{l}
EHR\_BASE(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ patId : VERS\_TRANS) \\
local\_chan\ getTransType : [tt? : TRANSTYPE] \\
local\_chan\ replaceTrans : [t : VERS\_TRANS;\ tt : TRANSTYPE] \\
local\_chan\ addTransaction : [t : VERS\_TRANS;\ tt : TRANSTYPE] \\
local\_chan\ recPersisTrans : [d : EHR\_CONT;\ str? : \text{seq}\ VERS\_TRANS] \\
method\ createTrans1 : [d? : EHR\_CONT;\ cn? : TRANSD] \\
chan\ askForTime : [tm? : DATE\_TIME] \\
main = createTrans1?d?cn \rightarrow getTransType?tt \rightarrow askForTime?tm \rightarrow \\
\quad New\ c : COMMIT\_AUDIT(cn.hca, tm, cn.sr, cn.rs, cn.cs) \bullet \\
\quad New\ v : TRANS(d, c) \bullet \\
\quad if\ ((tt = demographic) \lor\ (tt = eventual)) \\
\quad then\ New\ t : VERS\_TRANS(cn.ac, cn.am, tm, cn.gv, v) \bullet \\
\quad\quad addTransaction.t.tt \rightarrow SKIP \\
\quad else\ recPersisTrans.d?str \rightarrow \\
\quad\quad (if\ not\ null(str) \\
\quad\quad then\ (head\ str[|\ \{|\ addVersion\ |\}\ |] \\
\quad\quad\quad\quad (addVersion!v \rightarrow replaceTrans!(head\ str)!tt \rightarrow SKIP)) \\
\quad\quad else\ New\ t : VERS\_TRANS(cn.ac, cn.am, tm, cn.gv, v) \bullet \\
\quad\quad\quad\quad (addTransaction.t.tt \rightarrow SKIP))
\end{array}
$$

A classe EHR_BASE possui os seguintes atributos:

- *ehr_id*: identificador do registro de paciente.
- *hcp_created_by*: médico que criou o registro de paciente.
- *creation_time*: data e hora da criação do registro de paciente.
- *ehr_source*: identificador do prontuário eletrônico.
- *subject*: transação de cadastro de paciente.
- *demographic_entities*: seqüência de transações demográficas (endereços de médicos e instituições de saúde relacionados ao paciente). Na especificação estamos utilizando o nome *dem_entities*.
- *persistent_clinical_transactions*: seqüência de transações clínicas e persistentes do paciente. Na especificação estamos utilizando o nome *pers_clin_trans*.
- *event_clinical_transactions*: seqüência de transações clínicas e eventuais do paciente. Na especificação estamos utilizando o nome *event_clin_trans*.
- *tt*: tipo de transação do registro em que será adicionada a transação.
- *vid*: identificador da transação.

Esta classe possui um invariante que garante que o atributo *subject* só possui uma transação de conceito *patientId*; o *dem_entities* contém somente transações de conceito *address*; o *pers_cli_trans* possui apenas transações de conceitos *bloodPressure* e *headache*; o *event_cli_trans* contém somente transações de conceitos *haemogram* e *consultation*.

$EHR\_BASE(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ patId : VERS\_TRANS)$

$ehr\_id : \mathbb{N}$
$hcp\_created\_by : HCA$
$creation\_time : DATE\_TIME$
$ehr\_source : SRCE$
$subject : VERS\_TRANS$
$dem\_entities : \text{seq}\ VERS\_TRANS$
$pers\_clin\_trans : \text{seq}\ VERS\_TRANS$
$event\_clin\_trans : \text{seq}\ VERS\_TRANS$
$tt : TRANSTYPE$
$vid : DATE\_TIME$

$(\forall t : VERS\_TRANS \mid t = subject \bullet last\ (t.versions).concept \in Concept\_Subject\_Set)$
$(\forall t : ran\ dem\_entities \bullet last\ (t.versions).concept \in Concept\_Demographic\_Set)$
$(\forall t : ran\ pers\_cli\_trans \bullet last\ (t.versions).concept \in Concept\_Persistent\_Set)$
$(\forall t : ran\ event\_cli\_trans \bullet last\ (t.versions).concept \in Concept\_Eventual\_Set)$

INIT

$ehr\_id = id \land hcp\_created\_by = hca \land creation\_time = tm \land ehr\_source = sr$
$subject = patId \land dem\_entities = \varnothing \land pers\_clin\_trans = \varnothing \land event\_clin\_trans = \varnothing$

Segue a descrição de cada operação da classe EHR_BASE.

- *effect_createTrans1*: recebe como parâmetro de entrada os dados do paciente (*d?*) e as informações necessárias para criar a transação desejada (*cn?*). Essa operação atualiza apenas a variável de estado *tt*. O restante dos valores de entrada são utilizados pela parte de CSP para criar a transação (VERS_TRANS).
- *effect_getTransType*: retorna o tipo da transação e não atualiza o estado.
- *effect_addTransaction*: adiciona a transação *t* na lista de transações de tipo *tt*.
- *effect_recPersisTrans*: verifica, na lista de transações persistentes, a existência de uma determinada transação, a partir de informações do paciente (*d*). Se encontrar, retorna a transação numa lista, senão, retorna a lista vazia (*str?*).
- *effect_replaceTrans*: substitui a transação antiga pela nova de acordo com o tipo da transação informado (*tt*).

EHR_BASE$(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ patId : VERS\_TRANS)$

effect_createTrans1

$\Delta(tt)$
$d? : EHR\_CONT$
$cn? : TRANSD$

$tt' = cn?.tt$

effect_getTransType

$\Delta()$
$tt! : TRANSTYPE$

$tt! = tt$

effect_recPersisTrans

$\Delta()$
$d : EHR\_CONT$
$str? : \text{seq}\ VERS\_TRANS$

$(\exists\, t : \text{ran}\ pers\_clin\_trans \mid$
$head\ (t.versions).content.concept = d.concept \land str? =< t >) \lor$
$(\neg\ \exists\, t : \text{ran}\ pers\_clin\_trans \mid$
$head\ (t.versions).content.concept = d?.concept \land str? =<>)$

effect_addTransaction

$\Delta(subject, dem\_entities, pers\_clin\_trans, event\_clin\_trans)$
$t : VERS\_TRANS$
$tt : TRANSTYPE$

$tt = subject \Rightarrow (subject' = t \land dem\_entities' = dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = demographic \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities \frown \langle t \rangle$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = persistent \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans \frown \langle t \rangle \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = eventual \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans \frown \langle t \rangle)$

effect_replaceTrans

$\Delta(subject, dem\_entities, pers\_clin\_trans, event\_clin\_trans)$
$tr : VERS\_TRANS$
$tt : TRANSTYPE$

$tt = subject \Rightarrow (subject' = tr \land dem\_entities' = ehr.dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = demographic \Rightarrow (subject' = subject$
$(\exists\, i : dom\ dem\_entities \mid dem\_entities(i).time\_created = tr.time\_created \bullet$
$dem\_entities' = dem\_entities \oplus \{i \mapsto tr\})$
$pers\_clin\_trans' = pers\_clin\_trans \land event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = persistent \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities$
$(\exists\, i : dom\ pers\_clin\_trans \mid pers\_clin\_trans(i).time\_created = tr.time\_created \bullet$
$pers\_clin\_trans' = pers\_clin\_trans \oplus \{i \mapsto tr\})$
$event\_clin\_trans' = event\_clin\_trans) \lor$
$tt = eventual \Rightarrow (subject' = subject \land dem\_entities' = dem\_entities$
$pers\_clin\_trans' = pers\_clin\_trans$
$(\exists\, i : dom\ event\_clin\_trans \mid event\_clin\_trans(i).time\_created = tr.time\_created \bullet$
$event\_clin\_trans' = event\_clin\_trans \oplus \{i \mapsto tr\})$

As classes EHR e EHR_EXT modelam registro eletrônico de paciente e registro de importação e exportação. Como elas são subclasses de EHR_BASE, então herdam os atributos, operações e comportamento. Por simplificação, não vamos detalhar estas classes. As suas especificações completas estão no apêndice D.

EHR$(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ patId : VERS\_TRANS)$

inherit EHR_BASE
...

EHR_EXT$(id : \mathbb{N};\ hca : HCA;\ tm : DATE\_TIME;\ sr : SRCE;\ tg : SRCE;\ hcatg : HCA;$
$patId : VERS\_TRANS)$
inherit EHR_BASE
...

A classe EHRS é quem possui a coleção de registros de pacientes. Ela é responsável por:

- recuperar registros de pacientes;
- solicitar a criação de transações e versões de transações;
- requisitar a criação de transações importadas;
- solicitar transações para serem exportadas;

Após a declaração dos canais, encontramos o *main* que é formado por uma escolha externa de quatro processos: CREATE_TRANS, CREATE_VERSION, EXPORT_TRANS e IMPORT_TRANS. Por concisão, vamos apresentar apenas o processo CREATE_TRANS. As descrições dos outros processos se encontram no apêndice D.

$$
\begin{array}{l}
EHRS(sqId : \mathrm{seq}\, EHRID;\ sr : SRCE) \\
local\_chan\ notGetEhr, notGetIndObj : [] \\
local\_chan\ getIndObj : [r? : EHRID] \\
local\_chan\ updateEhr, addEhr : [ehr : EHR] \\
local\_chan\ getEhr : [ehr? : EHR] \\
local\_chan\ getTransType : [tt? : TRANSTYPE] \\
chan\ createTrans : [app? : APPLIC;\ d? : EHR\_CONT;\ cn? : TRANSD] \\
chan\ createTrans1 : [d! : EHR\_CONT;\ cn! : TRANSD] \\
chan\ askForTime : [tm? : DATE\_TIME] \\
chan\ failEhr : [app! : APPLIC;\ p! : PURPOSE] \\
... \\
main = CREATE\_TRANS \\
\qquad \Box \\
\qquad CREATE\_VERSION \\
\qquad \Box \\
\qquad EXPORT\_TRANS \\
\qquad \Box \\
\qquad IMPORT\_TRANS
\end{array}
$$

O processo CREATE_TRANS é responsável pela criação de transação. Ele recebe a solicitação de criação de uma transação (*createTrans*), recupera o tipo da transação (*getTransType*) e segue um dos dois fluxos:

1. Recupera o registro do paciente (*getEhr*). Se o tipo da transação for igual a *subject*, o processo avisa que o registro já existe e por isso não é possível criar essa transação (*failEhr*), e volta a se comportar como *main*. Por outro lado, se o tipo da transação for diferente de *subject*, o processo CREATE_TRANS solicita ao processo *ehr* a criação da transação (*createTrans1*), atualiza a sua coleção de registros (*updateEhr*), finaliza o processo, e volta a se comportar como *main*.

2. Não recupera o registro (*notGetEhr*). Se o tipo da transação for igual a *subject*, o processo pode seguir um dos dois caminhos:

   - Consegue um próximo índice disponível (*getIndObj*), solicita o próximo tempo (*askForTime*), cria os objetos necessários para criar o objeto *e* do tipo EHR, adiciona o registro na coleção de registros (*addEhr*), e volta a se comportar como *main*.
   - Não consegue um próximo índice disponível (*notGetIndObj*), avisa que não foi possível criar um registro (*failEhr*), e volta a se comportar como *main*.

   Por outro lado, se o tipo da transação não for igual a *subject*, o processo avisa que não foi possível criar a transação porque o registro do paciente não existe (*failEhr*), e volta a se comportar como *main*.

$$
\begin{array}{l}
\textit{EHRS}(\textit{sqId} : \text{seq}\, \textit{EHRID};\ \textit{sr} : \textit{SRCE}) \\
\textit{CREATE\_TRANS} = \textit{createTrans?app?d?cn} \rightarrow \textit{getTransType?tt} \rightarrow \\
(\textit{getEhr?ehr} \rightarrow \\
\quad \textit{if}\ (\textit{tt} == \textit{subject}) \\
\quad \textit{then}\ \textit{failEhr!app!p} \rightarrow \textit{main} \\
\quad \textit{else} \\
\quad\quad ((\textit{ehr} \\
\quad\quad [|\ \{|\ \textit{createTrans1}\ |\}\ |] \\
\quad\quad (\textit{createTrans1!d!cn} \rightarrow \textit{update.ehr} \rightarrow \textit{SKIP}));\ \textit{main}) \\
\Box \\
\textit{notGetEhr} \rightarrow \\
\quad (\textit{if}\,(\textit{tt} == \textit{subject}) \\
\quad \textit{then}(\textit{getIndObj?ind} \rightarrow \textit{askForTime?tm} \rightarrow \\
\qquad\qquad (\textit{New}\ c : \textit{COMMIT\_AUDIT}(\textit{cn.hca}, \textit{tm}, \textit{cn.sr}, \textit{cn.rs}, \textit{cn.cs}) \bullet \\
\qquad\qquad \textit{New}\ v : \textit{TRANS}(d, c) \bullet \\
\qquad\qquad \textit{New}\ t : \textit{VERS\_TRANS}(\textit{cn.ac}, \textit{cn.am}, \textit{tm}, \textit{cn.gv}, v) \bullet \\
\qquad\qquad \textit{New}\ e : \textit{EHR}(\textit{ind}, \textit{cn.hca}, \textit{tm}, \textit{cn.sr}, t) \bullet \\
\qquad\qquad \textit{addEhr.e} \rightarrow \textit{main}) \\
\qquad\quad \Box \\
\qquad\quad \textit{notGetIndObj} \rightarrow \textit{failEhr!app!p} \rightarrow \textit{main}) \\
\quad \textit{else}\ \textit{failEhr!app!p} \rightarrow \textit{main}))
\end{array}
$$

A classe EHRS possui os seguintes atributos:

- *ehrs*: coleção de registros de paciente.
- *ehr_id*: identificador do registro.
- *app*: endereço da aplicação.
- *tt*: tipo de transação.
- *s*: identificador do prontuário local.
- *p*: motivo da falha.
- *pid*: informações de identificação do paciente.
- *ehrExt*: registro de importação e exportação.
- *comm*: tipo de comunicação (importação ou exportação)

Além dos atributos, a classe possui um invariante que garante a unicidade dos identificadores de registros de paciente.

$EHRS(sqId : \text{seq}\, EHRID;\ sr : SRCE)$

$ehrs : \mathbb{P}\, EHR$
$seqIds : \text{seq}\, EHRID$
$app : APPLIC$
$ehr\_id : \mathbb{N}$
$tt : TRANSTYPE$
$s : SRCE$
$p : PURPOSE$
$pid : EHR\_CONT$
$ehrExt : EHR\_EXT$
$comm : EXT\_COMM$

$\forall e1, e2 : ehrs \bullet e1.ehr\_id = e2.ehr\_id \Rightarrow e1 = e2$

$INIT$

$ehrs = \varnothing$
$seqIds = sqId$
$s = sr$

Segue a descrição de cada operação da classe EHRS:

- *effect_updateEhr*: substitui o registro antigo (e) pelo novo (ehr).
- *effect_getEhr*: recupera o registro do paciente. Esta operação só é habilitada se existir o registro na coleção de registros.
- *effect_notGetEhr*: atualiza o motivo da falha. A operação *notGetEhr* só fica habilitada quando o registro desejado não é encontrado na coleção de registros.
- *effect_failEhr*: gera como saída o motivo da falha.
- *effect_createTrans*: recebe como parâmetros de entrada o identificador da aplicação ($app?$), os dados do paciente ($d?$) e as informações para criar a transação (cn?). Esta operação atualiza as variáveis de estado endereço da aplicação (app), identificador do registro do paciente (ehr_id) e o tipo da transação (tt).
- *effect_getTransType*: recupera o tipo da transação.
- *effect_addEhr*: adiciona o novo registro (ehr) na coleção de registros.
- *effect_getIndObj*: recebe um próximo índice disponível (primeiro da lista) e atualiza a variável de estado *seqIds* com a lista sem o primeiro elemento. Esta operação só fica habilitada enquanto existir elementos na lista de identificadores.
- *effect_notGetIndObj*: A operação *notGetIndObj* atualiza a variável de estado p informando que não há referência disponível. Esta operação só é habilitada quando não exisitr mais elementos na lista de identificadores.

$EHRS(sqId : \text{seq}\, EHRID;\ sr : SRCE)$

effect_updateEhr

$\Delta(ehrs)$
$ehr : EHR$

$(\exists\, e : ehrs \bullet e.ehr\_id = ehr.ehr\_id$
$ehrs' = (ehrs \setminus \{e\}) \cup \{ehr\}$

effect_failEhr

$\Delta()$
$app! : APPLIC$
$p! : PURPOSE$

$app! = app \land p! = p$

enable_getEhr

$\exists\, e : ehrs \bullet e.ehr\_id = ehr\_id$

effect_getEhr

$\Delta(p)$
$ehr? : EHR$

$\exists\, e : ehrs \bullet e.ehr\_id = ehr\_id$
$ehr? = e \land p' = RecordRecovered$

enable_notGetEhr

$\nexists e : ehrs \bullet e.ehr\_id = ehr\_id$

effect_notGetEhr

$\Delta(p)$

$p' = RecordNotRecovered$

effect_createTrans

$\Delta(ehr\_id, tt, app)$
$app? : APPLIC$
$d? : EHR\_CONT$
$cn? : TRANSD$

$ehr\_id' = cn?.ehr\_id \land tt' = cn?.tt \land app' = app?$

effect_getTransType

$\Delta()$
$tt? : TRANSTYPE$

$tt? = tt$

effect_addEhr

$\Delta(ehrs)$
$ehr : EHR$

$ehrs' = ehrs \cup \{ehr\}$

enable_getIndObj

$seqIds \neq <>$

effect_getIndObj

$\Delta(seqIds)$
$ind? : EHRID$

$ind? = head\ seqIds$
$seqIds' = tail\ seqIds$

enable_notGetIndObj

$seqIds = <>$

effect_notGetIndObj

$\Delta(p)$

$p' = WithoutReference$

5.4 Especificação do GEHR Utilizando o Padrão de Projeto

Apresentaremos apenas a implementação do processo CREATE_TRANS, que faz parte do componente EHRS, porque o nosso objetivo é mostrar a utilização do padrão na especificação de sistemas orientados a objetos. Além disso, existem limitações na ferramenta FDR relacionadas à explosão de estados.

Para facilitar a leitura, vamos apresentar a função que cria os processos e os tipos de dados que utilizam o nosso padrão. As definições de todas as funções e de todos os tipos de dados se encontram no apêndice E.

5.4.1 Funções

Para criar processos dinamicamente, utilizamos a função `proc`. Seus parâmetros de entrada são o tipo do processo e os valores de inicialização das variáveis de estado. Observamos que os parâmetros reais desta função determinam o processo que será ativado.

```
proc(TEHRS,(setEhr,seqId,sr)) =
   EHRS(setEhr,seqId,sr)
proc(TEHRBASE,(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)) =
   EHR_BASE(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)
proc(TEHR,(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)) =
   EHR(ehrid,ct,sr,sub,dement,perstr,evettrans,extmerg,extsent)
proc(TVERSTRANS,(ar,aa,tc,gv,vs)) =
   VERS_TRANS(ar,aa,tc,gv,vs)
```

Segue a declaração dos canais utilizados pelo processo `CREATE_TRANS`.

```
channel createTrans:AppDataCn
channel createTrans1:DataCn
channel askForTime:DATE_TIME
channel getEhr:Ehr
channel getTransType:TRANSTYPE
channel addVersion:TRANS
channel updateEhr:Ehr
channel addTransaction:TrTransType
channel recPersisTrans:EHR_CONT.SEQ_VERS_TRANS
channel failEhr:AppPurp
channel replaceTrans:TrTransType
channel notGetEhr
channel getIndObj: EHRID
channel notGetIndObj
channel addEhr:Ehr
channel recoverState:EhrContext
channel getStateEhrBase: EhrBaseContext
channel getStateEhr:Ehr
channel getStateVersTrans:VersTransContext
```

```
channel terminate
channel exit
```

5.4.2 Transações e Versões de Transações

O processo VERS_TRANS, o qual é uma abreviação de VERSIONED_TRANSACTION, modela uma transação. Ele é responsável por adicionar novas versões. Os parâmetros ar, aa, tc, gv e vs são utilizados para inicializar as suas variáveis de estado. Segue a definição de VERS_TRANS.

```
VERS_TRANS(ar,aa,tc,gv,vs) =
```

Na sua definição, várias constantes locais são introduzidas. Elas são apresentadas a seguir. A constante local Ops contém o conjunto de nomes de canais usados pelo processo referente a parte de Z.

```
 let
    Ops = {addVersion,getStateVersTrans,terminate}
```

A constante LocOps contém apenas os nomes dos canais locais. Este processo não possui canais locais.

```
    LocOps = {}
```

O comportamento do processo VERS_TRANS está definido pelo processo main abaixo.

```
    main = addVersion?v -> exit -> getStateVersTrans?st ->
           terminate -> SKIP
```

A constante local CSPOps contém o conjunto de nomes de canais usados pelo processo referente a parte de CSP. Neste caso, os nomes de canais são os mesmos utilizados pelo processo referente a parte de Z.

```
    CSPOps = Ops
```

O estado possui vários atributos, cujos tipos estão descritos no apêndice E. Ele também tem um invariante que garante que a transação deve ter pelo menos uma versão.

```
    state = {(access_rights,access_amend,time_created,gehr_version,
             versions) |access_rights <- PERMISSET,
             access_amend <- PERMISSET,time_created <- DATE_TIME,
             gehr_version <- GEHR_VERSION,
             versions <- SEQ_TRANS,(#versions > 0)}
```

O estado inicial da parte de Z do processo VERS_TRANS é dada pela constante init.

```
    init = {(access_rights',access_amend',time_created',gehr_version',
             versions')|(access_rights',access_amend',time_created',
             gehr_version',versions') <- state,access_rights'== ar,
             access_amend'== aa,time_created' == tc,
             gehr_version' == gv,versions' == vs, (#versions' > 0)}
```

A parte de Z é descrita por um conjunto de funções que são utilizadas pela função `Semantics` (Seção 3.5). Portanto, a seguir apresentamos as equações `in`, `out`, `enable` e `effect` referentes aos canais do processo `VERS_TRANS`. A função `in` aplicada à operação `addVersion` define que os valores de entrada desta operação devem pertencer ao conjunto `TRANS` (abreviação de `TRANSACTION`). Por outro lado, `in` aplicada às operações `getStateVersTrans` e `terminate` determina que elas não possuem valores de entrada.

```
in(addVersion) = TRANS
in(getStateVersTrans) = {{}}
in(terminate) = {{}}
```

A função `out` aplicada à operação `getStateVersTrans` define que os valores de saída devem pertencer ao conjunto `VersTransContext`. Esta função aplicada às operações `addVersion` e `terminate` não geram valores de saída.

```
out(addVersion) = {{}}
out(getStateVersTrans) = VersTransContext
out(terminate) = {{}}
```

A função `enable` habilita e desabilita operações. Neste caso todas as operações estão habilitadas.

```
enable(addVersion)((access_rights,access_amend,time_created,
 gehr_version,versions))= true
enable(getStateVersTrans)((access_rights,access_amend,time_created,
 gehr_version,versions))= true
enable(terminate)((access_rights,access_amend,time_created,
 gehr_version,versions))= true
```

A função `effect` executa a operação habilitada passada como parâmetro. Segue a descrição da função `effect` para cada operação.

A função `effect(addVersion)` adiciona uma versão na lista de versões da transação.

```
effect(addVersion)((access_rights,access_amend,time_created,
       gehr_version,versions),v) =
       {({},(access_rights',access_amend',time_created',
         gehr_version',versions')) |
         (access_rights',access_amend',time_created',
         gehr_version',versions') <- state,
         access_rights' == access_rights,
         access_amend' == access_amend,
         time_created' == time_created,
         gehr_version' == gehr_version,
         versions' == versions ^ <v>, (#versions' > 0)}
```

A função `effect(getStateVersTrans)` recupera o estado da transação e o atribui a variável de saída `o`.

```
effect(getStateVersTrans)((access_rights,access_amend,time_created,
       gehr_version,versions),_) =
       {(o,(access_rights',access_amend',time_created',
         gehr_version',versions')) |
        (access_rights',access_amend',time_created',
         gehr_version',versions') <- state,
        access_rights' == access_rights,
        access_amend' == access_amend,
        time_created' == time_created,
        gehr_version' == gehr_version,
        versions' == versions, o <- VersTransContext,
        o == (access_rights',access_amend',time_created',
              gehr_version',versions'),(#versions' > 0)}
```

A função `effect(terminate)` nem altera o estado e nem gera qualquer saída.

```
effect(terminate)((access_rights,access_amend,time_created,
       gehr_version,versions),_) =
       {({},(access_rights',access_amend',time_created',
        gehr_version',versions')) |
        (access_rights',access_amend',time_created',
        gehr_version',versions') <- state,
        access_rights' == access_rights,
        access_amend' == access_amend,
        time_created' == time_created,
        gehr_version' == gehr_version,
        versions' == versions, (#versions' > 0)}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

5.4.3 Registros

O processo `EHR_BASE` é responsável pela manutenção das transações. Segue a sua definição.

```
EHR_BASE(id,hca,tm,sr,sub,dem,pers,event) =

 let
    Ops = {createTrans1,addTransaction,recPersisTrans,replaceTrans,
           getStateEhrBase,terminate,getTransType}

    LocOps = {getTransType,addTransaction,replaceTrans}
```

O comportamento do processo `EHR_BASE` está definido pelo processo `main` a seguir.

```
main = createTrans1?dcn -> getTransType?tt -> askForTime?tm ->
       (let
          c = (hcaTransd(dcn),tm,srTransd(dcn),rsTransd(dcn),
               csTransd(dcn))
          v = (dataTransd(dcn),c)
         within
            (if ((tt == demographic) or (tt == eventual))
             then(let
                    t = (acTransd(dcn),amTransd(dcn),tm,
                         gvTransd(dcn),v)
                  within(addTransaction!(t,tt) -> exit ->
                         getStateEhrBase?st ->terminate -> SKIP))
             else
                recPersisTrans!(dataTransd(dcn))?str ->
                (if not null(str)
                 then ((let
                          e = head(str)
                          procVersTrans = proc(TVERSTRANS,e)
                          Flow = addVersion!v -> exit ->
                                 getStateVersTrans?t ->
                                 replaceTrans!(t,tt) ->
                                 getStateEhrBase?st ->
                                 terminate -> SKIP
                        within(procVersTrans [|{|addVersion,exit,
                               terminate,getStateVersTrans|}|]
                               Flow)))
                 else
                    (let
                       t = (acTransd(dcn),amTransd(dcn),tm,
                            gvTransd(dcn),v)
                     within(addTransaction!(t,tt) -> exit ->
                            getStateEhrBase?st -> terminate ->
                            SKIP)))))

CSPOps = Ops
```

O estado de EHR_BASE possui vários atributos, cujos tipos estão descritos no apêndice E. Além deles, o estado possui a função checkTransactions, que garante que cada tipo de transação só recebe transações de conceitos específicos. Esta função também está definida no apêndice E.

```
state = {(ehr_id,hcp_created_by,creation_time,ehr_source,subject,
          dem_entities,pers_cli_trans,event_cli_trans) |
          ehr_id <- EHRID, hcp_created_by <- HCA,tt<-TRANSTYPE,
          vid<-DATE_TIME,creation_time <- DATE_TIME,
          ehr_source <- SRCE,subject <- VersTransContext,
          dem_entities <- SEQ_VERS_TRANS,
```

```
pers_cli_trans <- SEQ_VERS_TRANS,
event_cli_trans <- SEQ_VERS_TRANS,
checkTransactions(subject,set(dem_entities),
  set(pers_cli_trans),set(event_cli_trans))}
```

O estado inicial da parte de Z do processo EHR_BASE é dada pela constante init.

```
init = {(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid') |
        (ehr_id',hcp_created_by',creation_time',ehr_source',
        subject',dem_entities',pers_cli_trans',event_cli_trans',
        tt',vid') <- state,ehr_id' == ehrid, hcp_created_by' == hcp,
        creation_time' == ct,ehr_source' == sr, subject' == sub,
        dem_entities' == dement,pers_cli_trans'== perstr,
        event_cli_trans == evettrans,
        checkTransactions(subject',set(dem_entities'),
            set(pers_cli_trans'),set(event_cli_trans'))}
```

Por simplicidade, a parte de Z referente à habilitação de operações e mudanças do estado será apresentada e detalhada para duas operações. Maiores detalhes sobre o restante das operações podem ser encontrados no Apêndice E. A função in aplicada à operação addTransaction define que os valores de entrada devem pertecer ao conjunto TrTransType. A aplicação desta função na operação getStateEhrBase determina que esta operação não possui valores de entrada.

```
in(addTransaction) = TrTransType
in(getStateEhrBase)= {{}}
```

A função out aplicada à operação getStateEhrBase determina que os valores de saída pertencem ao conjunto EhrBaseContext. Por outro lado, a função out aplicada à operação addTransaction não gera valores de saída.

```
out(addTransaction) = {{}}
out(getStateEhrBase)= EhrBaseContext
```

A função enable habilita e desabilita as operações. Neste caso, todas as operações se encontram habilitadas.

```
enable(addTransaction)((ehr_id,hcp_created_by,creation_time,
  ehr_source,subject,dem_entities,pers_cli_trans,event_cli_trans,
  tt,vid))= true
enable(getStateEhrBase)((ehr_id,hcp_created_by,creation_time,
  ehr_source,subject,dem_entities,pers_cli_trans,event_cli_trans,
  tt,vid))= true
```

A função effect executa as operações que estão habilitadas pelo enable. Segue a descrição da função effect para as duas operações.

Para adicionar uma transação no registro de paciente, utiliza-se a função effect aplicada ao parâmetro addTransaction, a qual usa duas outras funções: a ttTrTransType que recupera o tipo da transação na tupla i (parâmetro de entrada), e a trTrTransType que recupera a transação também na tupla i.

```
effect(addTransaction)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),i) =
  if ttTrTransType(i) == subject
  then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
        (ehr_id',hcp_created_by',creation_time',ehr_source',
         subject',dem_entities',pers_cli_trans',event_cli_trans',
         tt',vid') <- state,ehr_id'==ehr_id,
         hcp_created_by'== hcp_created_by,creation_time'==creation_time,
         ehr_source'== ehr_source,subject'== trTrTransType(i),
         dem_entities'== dem_entities,pers_cli_trans' == pers_cli_trans,
         event_cli_trans'==event_cli_trans,tt'==tt, vid'==vid,
         checkTransactions(subject',set(dem_entities'),
                           set(pers_cli_trans'),set(event_cli_trans'))}
  else
    if ttTrTransType(i) == demographic
    then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
          subject',dem_entities',pers_cli_trans',event_cli_trans',
          tt',vid')) |(ehr_id',hcp_created_by',creation_time',
          ehr_source',subject',dem_entities',pers_cli_trans',
          event_cli_trans',tt',vid') <- state,ehr_id'==ehr_id,
          hcp_created_by'==hcp_created_by,creation_time'==creation_time,
          ehr_source'== ehr_source,subject'== subject',
          dem_entities' == dem_entities ^ <trTrTransType(i)>,
          pers_cli_trans' == pers_cli_trans,
          event_cli_trans' == event_cli_trans,tt'==tt, vid'==vid,
          checkTransactions(subject',set(dem_entities'),
                            set(pers_cli_trans'),set(event_cli_trans'))}
    else if ttTrTransType(i) == persistent
         then {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
               subject',dem_entities',pers_cli_trans',event_cli_trans',
               tt',vid')) | (ehr_id',hcp_created_by',creation_time',
               ehr_source',subject',dem_entities',pers_cli_trans',
               event_cli_trans',tt',vid') <- state,
               ehr_id'==ehr_id, hcp_created_by'==hcp_created_by,
               creation_time'==creation_time,ehr_source'== ehr_source,
               subject'== subject', dem_entities' == dem_entities,
               pers_cli_trans' == pers_cli_trans ^ <trTrTransType(i)>,
               event_cli_trans' == event_cli_trans,tt'==tt,vid'==vid,
               checkTransactions(subject',set(dem_entities'),
                           set(pers_cli_trans'),set(event_cli_trans'))}
         else  {({},(ehr_id',hcp_created_by',creation_time',ehr_source',
                subject',dem_entities',pers_cli_trans',event_cli_trans',
                tt',vid')) |(ehr_id',hcp_created_by',creation_time',
                 ehr_source',subject',dem_entities',pers_cli_trans',
                 event_cli_trans',tt',vid') <- state, ehr_id'==ehr_id,
```

```
                hcp_created_by'==hcp_created_by,
                creation_time' == creation_time,ehr_source'== ehr_source,
                subject'== subject',dem_entities' == dem_entities,
                pers_cli_trans' == pers_cli_trans,
                event_cli_trans' == event_cli_trans ^ <trTrTransType(i)>,
                tt'== tt,vid'==vid,
                checkTransactions(subject',set(dem_entities'),
                          set(pers_cli_trans'),set(event_cli_trans'))}
```

A tupla de valores que representa o estado do processo `EHR_BASE` é recuperada e atribuída a variável de saída `o` através da função `effect(getStateEhrBase)`.

```
effect(getStateEhrBase)((ehr_id,hcp_created_by,creation_time,ehr_source,
  subject,dem_entities,pers_cli_trans,event_cli_trans,tt,vid),_) =
  {(o,(ehr_id',hcp_created_by',creation_time',ehr_source',subject',
  dem_entities',pers_cli_trans',event_cli_trans',tt',vid')) |
  (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
  dem_entities',pers_cli_trans',event_cli_trans',tt',vid') <- state,
  ehr_id'== ehr_id, hcp_created_by'== hcp_created_by,
  creation_time' == creation_time, ehr_source'== ehr_source,
  subject'== subject',dem_entities'== dem_entities,
  pers_cli_trans' == pers_cli_trans, tt'==tt, vid'==vid,
  event_cli_trans' == event_cli_trans, o <- EhrBaseContext,
  o == (ehr_id',hcp_created_by',creation_time',ehr_source',subject',
        dem_entities',pers_cli_trans',event_cli_trans',tt',vid'),
  checkTransactions(subject',set(dem_entities'),set(pers_cli_trans'),
                    set(event_cli_trans'))}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

A classe `EHR` é uma subclasse da classe `EHR_BASE`. Ela herda todos atributos, operações e comportamento da sua superclasse. Segue a sua definição.

```
EHR(ehrid,hcp,ct,sr,sub,dem,perstr,eventtr,extmerg,extsent) =
  let
    Ops = {recPersisTrans,replaceTrans,recoverState,terminate}

    LocOps = {recoverState}
```

O processo `main` é formado pelo paralelismo dos processos `procEhrBase` e `Flow`. O `Flow` possui apenas os eventos específicos de `EHR`; os herdados não aparecem nele porque o processo `EHR_BASE` é quem os realiza. Como estamos considerando apenas o processo `CREATE_TRANS`, não adicionamos no `Flow` eventos específicos de `EHR` porque eles estão relacionados a importação e exportação de transações.

```
main =
 let
   e = (ehrid,hcp,ct,sr,sub,dem,perstr,eventtr)
   procEhrBase = proc(TEHRBASE,e)
   Flow = exit -> getStateEhrBase?st1 -> recoverState?st ->
          getStateEhr!makeContext(st1,st) -> terminate -> SKIP
 within(procEhrBase [|{|getStateEhrBase,terminate,exit|}|] Flow)

CSPOps = Ops
```

O estado possui dois elementos, cujos tipos estão descritos no apêndice E.

```
state = {(extracts_merged,extracts_sents) |
          extracts_merged  <- SEQ_EHR_EXT,
          extracts_sents <- SEQ_EHR_EXT}
```

O estado inicial da parte de Z do processo `EHR` é dado pela constante `init`.

```
init = {(extracts_merged',extracts_sents')|
        (extracts_merged',extracts_sents') <- state,
         extracts_merged' == extmerg,
         extracts_sents' == extsent}
```

A função `in` aplicada nas operações `recoverState` e `terminate` determina que elas não possuem valores de entrada.

```
in(recoverState) = {{}}
in(terminate) = {{}}
```

A função `out` aplicada à operação `recoverState` determina que os valores de saída pertencem ao conjunto `EhrContext`. Esta função aplicada as outras operações não geram valores de saída.

```
out(recoverState) = EhrContext
out(terminate) = {{}}
```

A função enable habilita e desabilita as operações. Neste caso, todas elas estão habilitadas.

```
enable(recoverState)((extracts_merged,extracts_sents))= true
enable(terminate)((extracts_merged,extracts_sents))= true
```

A função effect executa as operações habilitadas. Segue a descrição da função `effect` aplicada a cada operação.

A função `effect(recoverState)` recupera o estado e o atribui a variável de saída `o`.

```
effect(recoverState)((extracts_merged,extracts_sents),_) =
 {(o,(extracts_merged',extracts_sents')) |
  (extracts_merged',extracts_sents') <- state,
   extracts_merged' == extracts_merged,
   extracts_sents' == extracts_sents,
   o <- EhrContext, o == (extracts_merged',extracts_sents')}
```

A função `effect(terminate)` não altera o estado e nem gera qualquer saída.

```
effect(terminate)((extracts_merged,extracts_sents),_) =
 {({},(extracts_merged',extracts_sents')) |
  (extracts_merged',extracts_sents') <- state,
  extracts_merged' == extracts_merged,
  extracts_sents' == extracts_sents}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo `EHRS` é responsável pela manutenção de registros de pacientes. No nosso caso, ele realiza apenas a inclusão de transações, porque só estamos tratanto o processo `CREATE_TRANS`.

```
EHRS(setEhr,seqId,s) =
 let
    Ops = {createTrans, getTransType, getEhr,notGetEhr,
           addEhr, updateEhr, getIndObj,notGetIndObj,failEhr}

    LocOps = {getTransType,getEhr,updateEhr}
```

O processo `main` possui apenas o processo `CREATE_TRANS`, porque o escolhemos para aplicar o padrão de projeto. Segue a descrição do processo `CREATE_TRANS`.

```
main = CREATE_TRANS
CREATE_TRANS = createTrans?appdcn  -> getTransType?tt ->
     (getEhr?ehr ->
      if (tt = subject)
      then failEhr?appp -> main
      else
        ((let
           e = ehr
           procEhr = proc(TEHR,e)
           Flow = createTrans1!patientData(appdcn) ->
                  exit -> getStateEhr?st ->updateEhr.st -> SKIP
       within (procEhr [|{|createTrans1,getStateEhr,exit|}|] Flow));
              main)

     []
     notGetEhr ->
       (if (tt == subject)
        then (getIndObj?ind -> askForTime?tm ->
              (let
                 c = (hcaTransd(appdcn),tm,srTransd(appdcn),
                       rsTransd(appdcn),csTransd(appdcn))
                 v = (dataTransd(appdcn),c)
                 t = (acTransd(appdcn),amTransd(appdcn),
```

```
                    tm,gvTransd(appdcn),v)
                e = (ind,hcaTransd(appdcn),tm,srTransd(appdcn),
                    t,<>,<>,<>,<>,<>)
               within(addEhr.e -> main))
          []
          notGetIndObj -> failEhr?appp -> main)
    else failEhr?appp -> main))

CSPOps = Ops
```

O conjunto SeqIds é resultado da aplicação da função FSeq, a qual cria um conjunto de seqüências cujos elementos têm o tipo dado pelo primeiro argumento e cujo tamanho máximo é dado pelo segundo argumento. A definição da função FSeq se encontra no apêndice E.

```
SeqIds = FSeq(EHRID,sizeSeq)
```

O estado de EHRS possui vários atributos, cujos tipos estão descritos no apêndice E. Além dos atributos, o estado possui a função checkEhrs que garante a unicidade dos identificadores de registros de paciente. Esta função caracteriza o invariante do processo EHRS.

```
state = {(ehrs,seqId,app,ehrid,tt,sr,p)|
          ehrs <- Set(Ehr),seqId <- SeqIds, app <- APPLIC,
          ehrid <- EHRID, tt <- TRANSTYPE,sr <- SRCE,
          p <- PURPOSE,checkEhrs(ehrs,seqId)}
```

O estado inicial da parte de Z do processo EHRS é dado pela constante init.

```
init = {(ehrs',seqId',app',ehrid',tt',sr',p') |
        (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
        ehrs' == setEhr, seqId'== sqId,
        sr'== s, checkEhrs(ehrs',seqId')}
```

Para facilitar a leitura, vamos apresentar a parte de Z referente à habilitação de operações e mudanças do estado para apenas três operações. O restante das operações estão definidas no Apêndice E. A função in define o conjunto de elementos de entrada para cada operação.

```
in(getEhr) = {{}}
in(addEhr) = Ehr
in(updateEhr) = Ehr
```

A função out define o conjunto de elementos de saída para cada operação.

```
out(getEhr) = Ehr
out(addEhr) = {{}}
out(updateEhr) = {{}}
```

A função `enable` habilita as operações que tiverem o predicado verdadeiro. Vamos comentar apenas a operação `getEhr` porque as demais possuem o predicado verdadeiro. A função `enable(getEhr)` só habilita a operação `getEhr` se encontrar o registro de paciente na coleção de registros. A função `ehrIdent` recupera o identificador do registro, o qual é um dos elementos do estado de `EHR`.

```
enable(getEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) =
  not empty({e | e <- ehrs,ehrIdent(e) == ehrid})
enable(addEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) = true
enable(updateEhr)((ehrs,seqId,app,ehrid,tt,sr,p)) = true
```

A função `effect` executa as operações que foram habilitadas pela função `enable`. No corpo de cada `effect`, encontramos a função `checkEhrs`, que garante que, após a realização da operação, os identificadores dos registros de paciente continuam sendo únicos. Segue a descrição da aplicação da função `effect` para cada operação.

Para recuperar o registro de paciente e atribuí-lo à variável de saída `o`, utiliza-se a função `effect(getEhr)`. A função `pick` retorna o elemento de um conjunto unitário. Esta função é própria da ferramenta FDR.

```
effect(getEhr)((ehrs,seqId,app,ehrid,tt,sr,p),_) =
    {(o,(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== ehrs, seqId'== seqId, app'== app, ehrid'== ehrid,
     sr'== sr,tt'== tt,p'== p, o <- Ehr,
     o == pick({e | e <- ehrs,ehrIdent(e) == ehrid'}),
     checkEhrs(ehrs',seqId')}
```

Para adicionar um registro na coleção de registros, utiliza-se a função `effect(addEhr)`.

```
effect(addEhr)((ehrs,seqId,app,ehrid,tt,sr,p),i) =
    {({},(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== union(ehrs,{i}), seqId'== seqId, app'== app,
     ehrid'== ehrid,tt'== tt, sr'== sr,p'== p,
     checkEhrs(ehrs',seqId')}
```

A substituição do registro antigo do paciente pelo atual (i) é realizada pela função `effect(updateEhr)`. Os outros atributos permanecem inalterados.

```
effect(updateEhr)((ehrs,seqId,app,ehrid,tt,sr,p),i) =
    {({},(ehrs',seqId',app',ehrid',tt',sr',p')) |
     (ehrs',seqId',app',ehrid',tt',sr',p') <- state,
     ehrs'== union(diff({ e | e <- ehrs, seqId'== seqId,
     ehrIdent(e) == ehrIdent(i)},ehrs),{i}),
     app'== app, ehrid'== ehrid, tt'== tt,sr'== sr,
     p'== p,checkEhrs(ehrs',seqId')}
```

A função `Semantics` tem sua definição e explicação na Seção 3.5

```
within Semantics(Ops,in,out,enable,effect,init,CSPOps,LocOps,main,event)
```

O processo SYSTEM define a arquitetura do sistema. Ele cria o processo EHRS através da função proc. Essa função recebe dois parâmetros: o primeiro é o tipo do processo a ser criado, e o segundo, os valores para a inicialização das variáveis de estado do processo EHRS.

```
SYSTEM =
  let
    ehrsState = ({},<1,2>,1)
    procEhrs = proc(TEHRS,ehrsState)
  within (procEhrs)
```

5.5 Análise em FDR e Conclusão

Primeiramente, utilizamos a ferramenta FDR para verificar a propriedade ausência de *deadlock* no sistema GEHR acima. Infelizmente, fazendo uso da tecnologia atual não foi possível realizar a análise do sistema diretamente, porque ele é grande demais. Na tentativa de conseguir realizar a análise, recorremos a algumas técnicas descritas em [60], para reduzir o estado do sistema, mas isso não foi suficiente, devido ao fato do sistema comunicar estruturas de dados complexas (objetos como valor).

Para viabilizar a análise do sistema, foi necessário restringir as comunicações, de forma *ad hoc*, para conjuntos viáveis de serem analisados. Essa análise inicial não apresentou *deadlock*, porém esse resultado não nos possibilita concluir que o sistema original também seja ausente de *deadlock*; a quantidade de dados a comunicar atua diretamente sobre a relação de refinamento, podendo mudar o resultado da análise de acordo com mudanças nos tipos usados nas comunicações. A análise do sistema original ainda depende de trabalhos na linha de simplificação de comunicações complexas, como, por exemplo, o de Mota e Sampaio [61]. Entretanto, como nossa principal contribuição não é análise de propridedades, mas sim no padrão de projeto, tal restrição é ortogonal aos nossos resultados.