Universidade Federal de Pernambuco

Centro de Informática

Graduação em Engenharia da Computação

2007.1

VERIFICANDO A APLICABILIDADE DE UMA ABORDAGEM DE GERAÇÃO DE ESPECIFICAÇÃO FORMAL A PARTIR DE DOCUMENTOS DE REQUISITOS

Adriano José Oliveira Gomes

TRABALHO DE GRADUAÇÃO

Orientador: Alexandre Cabral Mota

Recife

27 de agosto de 2007

Aos meus pais

AGRADECIMENTOS

A Deus, antes e acima de tudo, a Quem eu tudo devo e nunca seria capaz de agradecer suficientemente.

A meus pais, agentes proporcionadores da minha formação.

A Isis, minha noiva, pela enorme compreensão e apoio.

A meu orientador, Alexandre, pelos importantes e duradouros ensinamentos.

Aos meus amigos de curso, em especial, Farley, Vitor, Turah, Joabe, Paulo, Alan, Adelmário e Rodrigo, pela ajuda e companheirismo.

Aos meus mestres no CIn, pelo conhecimento adquirido.

E a todos, que contribuíram direta ou indiretamente para a conclusão deste trabalho, meus sinceros agradecimentos.

ABSTRACT

As requirements written in natural language are hard to process, it is difficult to generate other artifacts from them. To avoid this, an alternative is to use a formal language to describe requirements, which assures absense of ambiguity. As a example, the process algebra CSP (Communicating Sequential Processes) is a formal language to model behavioral aspects and it is straightforward to extract other artifacts from its models.

Furthermore, as formal methods are not widespread in industry, using them in a hidden way is a current trend. A very practical way to do that is creating a controlled natural language (a subset of English) which has a one-to-one mapping to a formal language.

In the present work we intend to investigate whether a previous approach designed to create formal specifications from controlled natural descriptions in the context of mobile phones is also applicable to a real case study of a subway, showing advantages and disadvantages.

After generating the formal specifications, we check some classical properties as well as compare it to its corresponding final implementation also described in CSP as well as the possibility of adaptations and extensions of the applied approach.

Keywords: Requirements Specification, Use Case, Formal Models, CSP, Controlled Natural Language.

RESUMO

No processo de desenvolvimento de sistemas, a especificação de requisitos é extremamente suscetível a erros, visto que os requisitos são definidos em linguagem natural, como o inglês. Este tipo de linguagem pode gerar ambigüidade na descrição dos requisitos ou ser de difícil entendimento, comprometendo uma correta implementação do sistema.

Uma alternativa para garantir a ausência de ambigüidade é usar linguagem formal para descrever os requisitos. Um exemplo de uma tal linguagem é CSP (Communicating Sequential Processes). A álgebra de processos de CSP modela aspectos comportamentais de sistemas e é usada de forma direta na extração de outros artefatos a partir de sua especificação.

Entretanto, como linguagens formais não são muito difundidas na indústria, uma tendência atual é utilizá-las de forma escondida. Uma maneira muito prática de se fazer isso é criando uma CNL (Controlled Natural Language), um sub-conjunto do Inglês que tem um mapeamento um-para-um para uma linguagem formal.

No presente trabalho nós pretendemos investigar se uma abordagem anterior, projetada para gerar especificações formais a partir de descrições naturais controladas no contexto de telefones móveis, também é aplicável a um estudo de caso real de um metrô; mostrando vantagens e desvantagens.

Após a geração das especificações formais, nós checamos algumas propriedades clássicas como também comparamo-las com sua implementação final correspondente também descrita em CSP como também a possibilidade de adaptações e extensões à abordagem aplicada.

Palavras chaves: Especificação de Requisitos, Casos de Uso, Modelos Formais, CSP, Linguagem Natural Controlada.

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

Os requisitos são o ponto de partida para o desenvolvimento de um software e conseqüentemente precisam ser tratados de forma especial a fim produzir artefatos coerentes e de alta qualidade. A fase de especificação de requisitos caracteriza-se como uma das principais fases do ciclo de desenvolvimento de software e requer o total entendimento do problema e sua correta especificação.

O uso de linguagem natural como inglês/ português é o meio mais ágil de se trabalhar em etapas iniciais do ciclo de desenvolvimento, por ser simples e flexível para se especificar um sistema, e ser a forma de comunicação e interação entre clientes e contratados.

Entretanto, este tipo de linguagem pode ser de difícil entendimento, capaz de gerar ambigüidades na descrição dos requisitos e comprometer uma correta implementação do sistema. Além disso, requisitos escritos usando linguagem natural são de difícil processamento, dificultando a geração de outros artefatos a partir dos mesmos.

O uso dos modelos formais, que são uma maneira abstrata de especificar os sistemas computadorizados, apresenta-se como uma alternativa para garantir artefatos coerentes e bem especificados. Os benefícios a respeito do uso dessa notação abstrata, antes de começar a execução de sistema, não estão relacionados somente a uma compreensão melhor do problema. O que se torna cada vez mais evidente é que o uso da representação formal abstrata combinada com o refinamento dos modelos pode mesmo promover o decréscimo do tempo de implementação do sistema. Uma das aplicações possíveis seria a geração automática do código de fonte a partir dos modelos formais [1]. A fase de teste também poderia ser positivamente atingida pelo uso dos modelos para a geração de casos de testes [2].

Há diversas maneiras de se especificar formalmente sistemas. Algumas linguagens de especificação são mais flexíveis do que outras. As linguagens formais de especificação usam aproximações matemáticas para validar propriedades da especificação. Os exemplos de linguagens formais são Z [8], CSP [9], e Circus [10].

Uma vez que um projeto tenha documentos de requisitos precisos é possível criar uma especificação formal a fim validar propriedades desejáveis do sistema. E como especificação formal de um sistema também favorece uma compreensão mais profunda do problema a ser modelado, conseqüentemente reduz erros nesta fase inicial do projeto, tornando bem mais simples os possíveis ajustes que o modelo venha a sofrer. Com base nisto, a médio ou longo prazo, os custos com a manutenção do software tendem a serem reduzidos [3].

A especificação de requisitos é uma atividade complexa e cara. Todo o erro durante sua definição pode ocasionar problemas em fases subseqüentes do desenvolvimento. A presença de requisitos temporários e instáveis é uma situação muito comum na maioria dos projetos. Manter os requisitos coerentes com os artefatos da execução e do teste mostrou ser uma tarefa difícil.

Apesar da existência dos benefícios formais da prova, a incidência de uso de métodos formais num projeto real é pequena. Este fato está relacionado à uma típica aversão por parte das equipes de desenvolvimento em lidar com requisitos formais e à pouca demanda de pessoal altamente especializado.

Além disso, a extração de especificação formal a partir de requisitos pode ser uma tarefa complexa. Os engenheiros de software são geralmente os responsáveis por esta tarefa. Eles usam de sua própria criatividade para especificar um modelo formal. Este é um processo manual e a especificação final pode não cobrir todos os requisitos especificados ou conter inconsistências a respeito. Além disso, pode ser mal entendido pelo coordenador ou até mesmo escrito de maneira ambígua ou ilegível. Dessa forma, este processo manual introduz a possibilidade de criação de um modelo formal problemático.

Devido a fraca aceitação sobre as linguagens formais e seus métodos atuais de implementação serem muitas vezes complexos e trabalhosos, a possibilidade de ocultação dessa etapa de tratamento formal seria de grande utilidade. O cenário ideal seria a existência de documentos de requisitos que pudessem automaticamente ser processado por um programa que gerasse o modelo formal.

Entretanto, o processamento de linguagens naturais pode ser impraticável se a compreensão textual dos requisitos depender do conhecimento prévio sobre o domínio da aplicação. Esta falta de informação traz a possibilidade da não compreensão dos requisitos [6] por um programa, ou pessoa. Conseqüentemente, técnicas de processamento de linguagem natural devem ser aplicadas a fim validar a semântica dos requisitos. Este é um grande desafio no campo de estudo da lingüística.

Uma alternativa é o uso de alguma linguagem mais simples para descrever requisitos ao invés da linguagem natural. Estas linguagens são chamadas de Linguagens Naturais Controladas (CNL) [7]. Possuem uma gramática menor e restrita impedindo que o escritor introduza sentenças ambíguas e não uniformes.

As CNL impactam não somente na estrutura do requisito, mas também na maneira de como devem ser escritos. Viabializam o uso de técnicas e ferramentas [4] que propõem um arranjo melhor dos requisitos, fazendo-os simples e diretos e tornando suas ambigüidades uniformes e categorizadas. Com isso, é possível se obter um mapeamento simples e mais direto desses requisitos para especificações formais.

Essa abordagem já foi aplicada em projetos de sistemas para celulares [5]. A abordagem detalhada em [5] utiliza *templates* de especificação de casos de uso e a Linguagem Natural Controlada para descrever os requisitos. Os *templates* de casos de uso asseguram a estruturação correta do documento de requisitos e a CNL garante a exatidão da gramática do texto que especifica o comportamento do sistema. A partir dessa estruturação dos requisitos foi possível definir uma estratégia de geração automática de uma especificação formal da aplicação em questão. Essa solução apresentou diversos benefícios para o sistema em questão, resta verficarmos ser pode ser extensível para sistemas com outros contextos de aplicação.

Sistema com controle via hardware e software podem conter as características complexas, que incluem trocar simultânea de comportamento e de mensagem. Especificar requisitos para sistemas desta natureza pode ser uma tarefa árdua, especialmente se for necessário validar o comportamento capturado. Pode ser bastante desgastante analisar componentes simultâneos, funcionando em paralelo, e validar seu comportamento.

Este trabalho é um esforço inicial para geração de uma especificação formal de um sistema de controle de portas de um metrô [17], mediante a aplicação da abordagem anterior. A linguagem de álgebra de processamento paralelo (CSP) gerada pode ser muito útil para validar os requisitos do sistema, usando particularmente o verificador modelo do FDR [18].

Muitas vezes, a construção de um modelo formal é uma tarefa que envolve tomada de decisão. Há diversas soluções possíveis para o mesmo problema e é necessário encontrar a mais apropriada. Conseqüentemente, a solução não pode ser simplesmente obtida a partir de rotinas de um programa. Informações adicionais fazem-se necessárias para criar automaticamente uma especificação formal. Assim, adaptações sobre essas técnicas e ferramentas mediante o contexto do sistema abordado são necessárias para que a

solução seja eficiente o suficiente para extrair todos os detalhes e características do sistema envolvido, bem como possa gerar um resultado que seja o mais apropriado para o sistema.

Estes são alguns dos tópicos previstos para este trabalho que envolve a engenharia de requisitos, processamento de linguagem natural, e conceitos, técnicas e ferramentas de métodos formais.

1.1 CONTEXTO E OBJETIVOS

O contexto deste trabalho é fruto de uma cooperação de pesquisa entre o CIn/UFPE e uma empresa de São Paulo responsável por um sistema do complexo ferroviário do metrô de Santiago no Chile, estando relacionado mais especificamente à um sistema de controle geral de portas (CGP) de um metrô. O sistema executará as funções de abertura e fechamento de portas, além da execução de funções periféricas como troca de informações com o sistema de interface homem-máquina do trem, sinalizações e indicações.

Dessa forma, exploraremos a especificação de requisitos desse sistema e verificaremos se a CNL proposta e seu respectivo modelo formal gerado reflete-se neste domínio. Para isso, consideramos os casos de uso especificados no sistema utilizando uma visão por componentes.

Este trabalho será parte inicial de um conjunto de trabalhos que procuram fechar o ciclo de verificação de propriedades de um sistema. De um lado teremos uma abordagem pelo "fluxo natural" partindo de requisitos – aplicação de uma estratégia de geração automática de modelagem formal a partir de requisitos –, que é foco desse trabalho, e do outro, uma abordagem pós-implementação – extração do modelo formal após implementação partindo do código fonte [36].

A figura a seguir (Figura 1.1) ilustra a seqüência de atividades a serem realizadas para a implementação dos trabalhos conjuntos. As atividades sombreadas representam o escopo desse trabalho. Existe ainda um outro trabalho que busca realizar um estudo sobre o uso das ferramentas de análise de propriedades de sistemas a partir do código implementado (Model Checking [37]), a fim de relacionar quais seriam as ferramentas mais aplicáveis para o nosso caso [30].

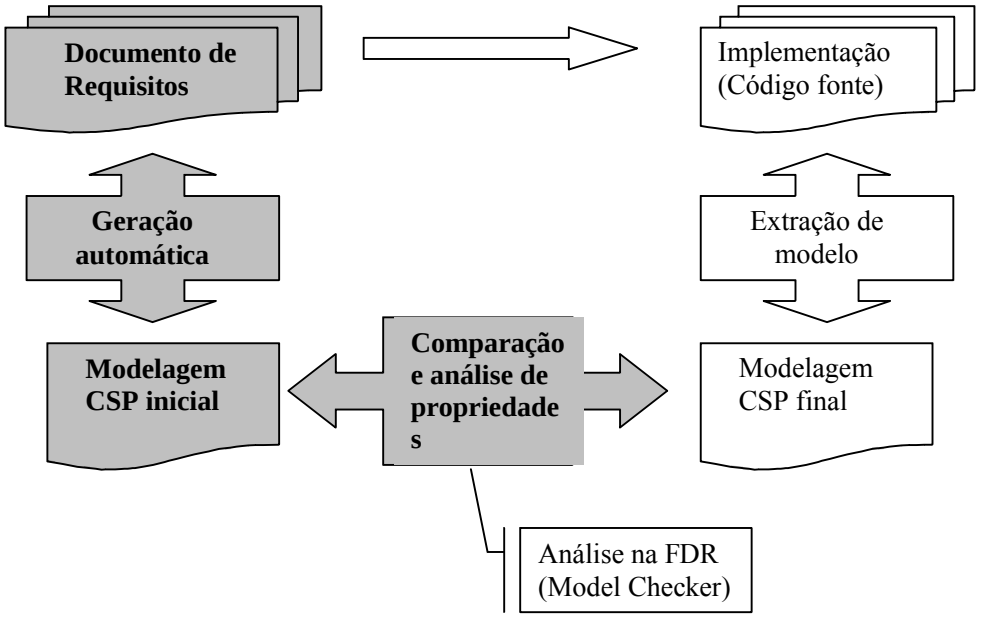

Figura 1.1 Contexto geral de trabalhos

Propomos, neste trabalho, a aplicação e avaliação de uma abordagem para extração de um modelo formal a partir de requisitos em linguagem natural apresentada em [5] a fim de indentificar alguma inconsistência e sugerir adaptações necessárias à abordagem em relação ao sistema em estudo. O modelo formal gerado é extraído através da aplicação sucessiva de regras de mapeamento de CNL para CSP.

Em seguida, realizaremos verificações sobre as propriedades do sistema a partir da especificação formal gerada, a fim de averiguar se as características do sistema em análise foram validadas. Inclusive, será feita uma comparação entre nossa verificação e os resultados obtidos em [36].

Dessa forma, apresentaremos os benefícios e possíveis problemas dessa estratégia em relação ao sistema em análise, de forma que ela possa posteriormente suprir as carências encontradas e fornecer facilidades para criação e validação de documentos.

1.1.1 Contribuições do Trabalho

De forma resumida, as contribuições desse trabalho são:

- Geração de uma especificação em CSP para o sistema CGP do Metrô
- Identificação de problemas e inconsistências presentes na especificação gerada em relação ao domínio do sistema

- Propomos um conjunto de adaptações à estratégia de extração para suprir alguns dos problemas encontrados.
- Análise e validação das propriedades do sistema: deadlock, livelock e não-determinismo.

Analisando a abordagem adotada, verificou-se que em se tratando do seu modelo e forma de extração da especificação ela possui uma característica comprometedora para nossa aplicação.

Assim, este trabalho está dirigido aos problemas relacionados à criação e validação de requisitos de um sistema específico (CGP - Metrô), analisando maneiras de melhorá-lo e automatizar a geração de modelos formais. Tais soluções podem ser estendidas futuramente para melhorar qualquer tipo de especificação de sistema, reduzindo custos do projeto e tempo de desenvolvimento e fornecendo artefatos de boa qualidade.

1.2 VISÃO GERAL

Este trabalho foi estruturado da seguinte maneira: no próximo capítulo é dada uma visão geral sobre a linguagem formal CSP expondo seus conceitos básicos. No capítulo 3, apresentamos a Linguagem Natural Controlada (CNL) com uma gramática fixa usada para representar as etapas dos casos de uso, que é fundamental para a abordagem proposta.

No Capítulo 4, apresentamos a abordagem adotada para a geração do modelo em CSP a partir dos documentos correspondentes escritos em CNL do inglês. Todas as etapas de seu processo são detalhadas e justificadas para utilização em nosso estudo de caso.

A contribuição deste trabalho se dá com o desenvolvimento de um estudo de caso, o qual é mostrado no Capítulo 5. Nele, fornecemos uma investigação para a concepção da abordagem e solução do estudo de caso.

Expomos as regras de mapeamento de CNL para CSP e o processo de aplicação de cada etapa da estratégia sobre o sistema em estudo. Em seguida, apresentamos os problemas ou dificuldades encontradas na geração do modelo em CSP e verificação das propriedades do sistema. Detalharemos também os resultados obtidos após a análise da estratégia.

Por fim, no Capítulo 6, serão apresentadas as conclusões sobre o trabalho. Também serão discutidos trabalhos futuros e relacionados.

CAPÍTULO 2

CSP

CSP [19, 20] surgiu na década de 1970 através do trabalho de C. A. Hoare [19]. Ele introduziu a idéia de processos com variáveis locais que interagem apenas através de trocas de mensagens. Em sua essência, aquela versão era apenas uma linguagem para programação concorrente e não tinha uma semântica matematicamente definida.

Nos anos posteriores, Hoare conseguiu evoluiu sua idéia e apresentou a versão teórica de CSP. Desde então, a teoria de CSP apresentou apenas pequenas mudanças. A motivação para a criação dessas mudanças é a concepção de ferramentas de análise e verificação automáticas. O texto definitivo para a teoria de CSP é apresentado no trabalho de A. W. Roscoe [20].

CSP (Communicating Sequential Processes) é uma linguagem formal utilizada para modelar o comportamento de sistemas concorrentes e distribuídos. Uma forma de entender CSP é imaginar um sistema como uma composição de unidades comportamentais independentes (subsistemas, componentes ou simplesmente rotinas de processamento) que se comunicam entre si e com o ambiente que os cerca [19]. Cada uma destas unidades independentes pode ser formada por unidades menores, combinadas por algum padrão de interação específico. Consideramos o ambiente como todo agente externo que pode interagir com o sistema, como os seus usuários ou outros sistemas.

CSP permite a descrição de sistemas em termos de processos/ componentes que se operam independentemente, e interagem uns com os outros através de comunicação por meio de mensagens.

As interações entre processos diferentes, e a maneira que cada processo se comunica com seu ambiente são descritos usando vários operadores algébricos de processos. Usando esta solução algébrica, as descrições de processos complexos podem ser facilmente construídas a partir de alguns elementos primitivos.

Alguns dos construtores e operadores de CSP são o *datatype* (datatype D), o *event* (channel a: D), o *prefix* (a → P), *escolha determinística* (P □ Q), *escolha não-determinística* (P ⊓ Q), *interleaving* (P ||| Q), *a composição paralela* (P | [s] | Q, onde s é o

conjunto de eventos em que P e Q sincronizam), e o *hiding* (P \ s, onde s é o conjunto dos eventos a ser escondido). Estes construtores e operadores são detalhados em seguida.

2.1 CONCEITOS BÁSICOS

2.1.1 Processo

Os processos são as entidades básicas que capturam um comportamento. Cada processo pode ser definido através de equações que, por questões de modularidade, podem ser definidas como um conjunto de processos. Além de denotar os módulos de um sistema, o nome de um processo pode denotar o estado de um processo.

O comportamento de um processo de CSP é descrito em termos de eventos, que são operações imediatas, como ABRIR ou FECHAR que pode transmitir informações. Um processo primitivo pode ser compreendido como uma representação de um comportamento básico. Há dois processos primitivos em CSP: *STOP* e *SKIP*. STOP é o processo que não se comunica com nada. Ele é usado para descrever a falha de um sistema, assim como uma situação de deadlock. O SKIP é o processo que indica o término com sucesso do comportamento de um processo. Na figura abaixo, detalhamos os operadores algébricos e os processos primitivos de CSP usados para compor a equação de processos complexos.

$$
\begin{array}{lll}
P(s) & ::= & \boldsymbol{STOP} \\
 & | & \boldsymbol{SKIP} \\
 & | & a \rightarrow P(s) \qquad (prefixo) \\
 & | & P(s) \qquad (recursao) \\
 & | & g \;\&\; P(s) \qquad (escolha\ condicional) \\
 & | & P(s) \;\Box\; P(s) \qquad (escolha\ externa) \\
 & | & P(s) \sqcap P(s) \qquad (escolha\ interna) \\
 & | & P(s) \backslash \mathrm{C} \qquad (internalizacao) \\
 & | & P(s)[\mathrm{R}] \qquad (renomeacao) \\
 & | & P(s) \;;\; P(s) \qquad (composicao\ sequencial) \\
 & | & P(s) \;\triangle\; P(s) \qquad (interrupcao) \\
 & | & P(s) \;||\; P(s) \qquad (paralelismo) \\
 & & C
\end{array}
$$

Figura 2.1 Equações de processos

Na literatura ainda existem outros processos, como DIV e RUN, estes não são considerados primitivos. O processo DIV representa um estado de livelock, e pode ser simulado através de um processo que executa ações internas indefinidamente. Estas ações não são percebidas por outros processos ou pelo ambiente. Sua definição é basicamente DIV = DIV. O processo RUN representa um processo que aceita sincronizar com qualquer evento em qualquer instante de tempo, e volta a se comportar como RUN novamente.

2.1.2 Datatype

Sempre que alguma informação necessita ser transmitida é necessário definir seu tipo e valores possíveis a partir do datatype. Conseqüentemente, os datatypes complexos podem ser definidos e usados nas definições de canais (eventos que transferem dados) para especificar a comunicação dos canais.

2.1.3 Comunicação

A comunicação em CSP significa três coisas: interação, pois dois ou mais processos interagem através da comunicação; observação, pois só podemos observar o comportamento dos processos através da sua comunicação; e sincronização, pois dois processos que estão executando paralelamente sincronizam suas execuções através da comunicação. Por convenção, o nome da comunicação é escrito utilizando-se letras minúsculas.

Uma comunicação pode ser:

- **Evento** – Um evento é a menor unidade em uma especificação CSP e representa a ocorrência de um fato relevante para entender o comportamento do sistema. Todos os eventos de uma especificação pertencem a um alfabeto (§), e este define o nível de abstração da especificação. Os eventos descrevem as interações mais simples entre os processos. Para um processo RELOGIO, poderíamos definir os eventos tic e tac, que indicam a passagem do tempo.
- **Canal** – Os canais diferem-se dos eventos devido ao fato de transmitirem dados. Os tipos dos dados transmitidos devem estar de acordo com o tipo do canal. Em um sistema que especifica um banco, um processo CAIXA poderia informar o valor debitado da conta de um cliente ao processo SERVIDOR através de um canal

debitar, por exemplo. Além de informar a ação do débito, a comunicação informa o valor do débito.

2.1.4 Alfabeto

O alfabeto de uma especificação é a união de todas as comunicações presentes nas definições de todos os processos.

2.2 OPERADORES

2.2.1 Prefixo

O prefixo é a operação mais simples que envolve um processo. Define um acoplamento de um processo em um evento. O operador de prefixo sempre possui um evento do lado esquerdo e um processo do lado direito. Dessa forma. o comportamento do processo é como o processo sufixado. Se x é um evento e P um processo, (x $\rightarrow$ P) representa o processo que espera indefinidamente por x, e quando x ocorre, o processo comporta-se então como P.

Graças ao recurso de composição de processos em CSP é possível criar um processo com vários prefixos em seqüência, como no exemplo a seguir. Nestes casos o escopo dos parâmetros de entrada se estende pelos eventos subseqüentes. Este operador ainda pode ser usado para modelar processos recursivos.

$$\textbf{channel}\ tick, tack, abrir, fechar$$

$$Relogio = tick \rightarrow tack \rightarrow tick \rightarrow tack \rightarrow \textbf{STOP}$$
$$Porta = abrir \rightarrow fechar \rightarrow \textbf{SKIP}$$

Figura 2.2 Operador Prefixo

2.2.2 Recursão

A Recursão em CSP é a habilidade de um processo de incorporar um comportamento repetitivo. O operador ($\rightarrow$), apresentado na seção anterior, pode ser usado

para modelar processos recursivos. O comportamento dos processos (P = x $\rightarrow$ P) é a repetição indefinida do evento x.

A recursão é útil para definir um processo através de uma única equação, mas também é útil para definir processos que possuem recursão mútua entre si. Por exemplo,

$$P = up \rightarrow Q$$
$$Q = down \rightarrow P$$

Se uma equação recursiva é prefixada por um evento, então é chamada recursão guardada. Tal classe de recursão é de grande importância em CSP, haja vista que previne a ocorrência de livelock.

Uma alternativa para definir a recursão é através do operador μ. Por exemplo, o processo Clock apresentado anteriormente poderia ser representado como:

$$Clock = \mu X \bullet tick \rightarrow tack \rightarrow X$$

2.2.3 Composição Sequencial

O operador de composição seqüencial *";"* permite que dois processos sejam executados segundo uma ordem de precedência. O segundo processo é iniciado após o término com sucesso do primeiro processo. Por exemplo, o processo

$$Q; R$$

comporta-se inicialmente como o processo Q. Após o término com sucesso de Q (identificado quando este passa a comportar-se como SKIP) o processo Q; R passa a comportar-se como R.

Diferente do operador de prefixo, que permite eventos consecutivos compartilharem o escopo de uma mesma variável, o operador de composição seqüencial não permite a extensão doescopo das variáveis do primeiro processo para o segundo. Assim, na composição seqüencial

$$(a?x \rightarrow SKIP); P$$

a variável x não será percebida pelo processo P.

2.2.4 Escolha Interna e Externa

Para representar um comportamento determinístico alternativo, podemos usar o operador □ (escolha externa) e para um não-determinístico, fazemos uso do operador ⊓ (escolha interna). O primeiro fornece ao ambiente o controle sobre a escolha das opções de comportamento. Enquanto que no segundo, o ambiente não tem nenhuma influência sobre a seleção dos comportamentos. Assim, esses dois operadores modelam bifurcações e desvios nos processos.

A escolha externa, ou escolha generalizada possui como argumentos dois processos, por exemplo, $P \square Q$. Este processo oferece ao ambiente a escolha entre os primeiros eventos de P e Q, que devem ser diferentes. Após isso, o processo assumirá o comportamento do processo escolhido. Por exemplo:

$$(a \rightarrow P \square b \rightarrow Q)$$

O processo tenta comunicar os eventos iniciais a e b. Caso o ambiente aceite comunicar a, o processo passa a se comportar como P. Caso o evento b seja aceito pelo ambiente, o processo se comporta como Q.

O comportamento do operador de escolha interna é definido através da escolha pelo próprio processo entre os eventos iniciais de P e Q. Como a escolha não depende do ambiente do sistema, mas apenas do processo $P \sqcap Q$, ela se dá internamente, ou seja a escolha entre eles é definida de forma não-determinística.

A escolha externa se comporta como a interna quando os eventos iniciais dos processos são iguais. Neste caso quem decide qual evento deve ocorrer é o próprio processo, sem a interferência do ambiente, assumindo um comportamento não-determinístico.

Uma diferença importante entre os dois operadores aqui descritos é a questão da obrigatoriedade da comunicação. Em uma escolha externa a comunicação é obrigatória se é oferecido ao ambiente apenas o evento inicial de um dos processos, já na escolha interna,

é possível rejeitar qualquer um dos eventos. Na escolha interna, apenas é obrigatória a comunicação se o ambiente oferecer ambos os eventos.

2.2.5 Escolha Condicional

Além das escolhas apresentadas na seção anterior, CSP possui ainda escolhas condicionais baseadas em variáveis introduzidas através de parâmetros de processos ou comunicações de entrada. Assim, estas variáveis podem ser utilizadas para determinar o comportamento dos processos, através de expressões lógicas.

if (b) then P else STOP

O operador de escolha condicional *if-then-else* se comporta como nas demais linguagens de especificação e programação. Também podemos encontrar o operador de guarda *b* & *P*, uma notação concisa e elegante da escolha condicional if *b* then *P* else STOP. Seu significado é: se a condição *b* não for satisfeita então o comportamento como um todo é bloqueado (STOP).

2.2.6 Composição Paralela

Até aqui os processos descritos têm representado ações seqüenciais. Mesmo os processos que oferecem alternativas de execução (externa ou interna) determinam que apenas um fluxo de execução seja escolhido. Através do paralelismo é possível executar mais de um processo simultaneamente, havendo comunicação entre eles.

Quando dois processos são postos na execução simultânea, na maioria das vezes, o desejo é que um interaja com o outro. As interações podem ser vistas como eventos que requerem a participação simultânea de ambos os processos. Se P e Q são processos com o mesmo alfabeto,

$$P \,||\, Q$$

representa um processo em que P e Q devem ser sincronizados em todos os eventos. Assim um evento x ocorre somente quando ambos os processos estão prontos para o aceitar. O processo

$$P \; [|X|] \; Q$$

sincroniza P e Q no evento do conjunto X. P e Q podem interagir independentemente com o ambiente através dos eventos fora do conjunto X. O Processo

$$P \; ||| \; Q$$

permite que P e Q executem simultaneamente sem sincronização entre eles. Cada evento, oferecido a uma intercalação de dois processos, ocorre somente em um deles. Se ambos estiverem prontos para aceitar esse evento, a escolha entre os processos é não-determinística.

2.3 MODELOS SEMÂNTICOS DE CSP

CSP possui um conjunto de leis algébricas que permitem provar equivalências semânticas (através de refinamentos) entre processos sintaticamente diferentes. Algumas destas leis podem ser usadas para reescrever a definição de processos, tornado-os mais simples ou atendendo algum padrão estrutural, sem, no entanto, mudar o comportamento do processo original. A leis algébricas também permitem a verificação com rigor matemático de propriedades clássicas de sistemas concorrentes e distribuídos, como o determinismo e a ausência de deadlock ou livelock.

CSP pode ser visto sob 3 estilos semânticos diferentes: denotacional, operacional e algébrico. A escolha do estilo está intimamente relacionada ao propósito de seu uso, pois cada um possui um tratamento matemático diferente. Neste trabalho introduzimos o denotacional porque ele é usado para definir a relação de refinamento de CSP com os projetos relacionados. A semântica denotacional de CSP é usualmente definida em termos dos modelos de *traces*, *falhas e divergências*. A seguir, descreveremos cada um destes modelos.

2.3.1 Modelo de Traces

O modelo de traces é definido como um conjunto de seqüência de eventos possíveis que uma especificação pode executar. Um trace de uma especificação define uma

seqüência de eventos finita possível que um processo executou até eles. Entretanto, não define o conjunto de traces que uma especificação não pode executar. Conseqüentemente, os processos P □ Q e P ⊓ Q são equivalentes no modelo dos traços. Este modelo é útil para verificar até que ponto um padrão comportamental é atendido.

Para qualquer processo P, o conjunto de traces(P) deve obedecer às seguintes propriedades:

- traces(P) sempre contém a seqüência vazia (<>).
- Se s^t pertence a traces(P), então s também pertence.

Para ilustrar este modelo, considere o processo S definido por

$$S = a \text{ -> } b \text{ -> } c \text{ -> } STOP$$

ele é representado no modelo de traces como

$$traces(S) = \{ <> , < a > , < a , b > , < a, b, c> \}$$

Os traces de um processo são obtidos a partir de um conjunto de regras, que definem como obter os traces de processos simples (como SKIP e STOP) diretamente e traces de processos compostos (como P [] Q) em termos dos traces de seus componentes (P e Q). A seguir são apresentadas algumas destas regras:

1. traces(STOP) = { < > }
2. traces(a -> P) = { < > } U { < a > ^ s | s □ traces(P) }
3. traces(P |~| Q) = traces(P) U traces(Q)
4. traces(P [] Q) = traces(P) U traces(Q)

Vale salientar que o modelo de traces não serve para verificar determinismo de um processo. Ele apenas serve para explicar o que o processo pode fazer. Por isso, as regras para o tratamento dos operadores de escolha interna (|~|) e de escolha externa ([]) produzem a mesma saída.

2.3.2 Modelo de Falhas

Este modelo é mais poderoso que o anterior, em termos de investigação de propriedades: além de indicar o que o processo pode fazer, ele indica onde os processos falham. Assim, este modelo permite demonstrar que o processo P □ Q não é semanticamente equivalente ao processo P ⊓ Q, se considerarmos o conjunto de eventos que podem não ser aceitos. Através do modelo de Falhas também é possível verificar se um processo é determinístico ou não. Um processo é dito determinístico se ele não se comporta diferentemente a partir da mesma situação inicial.

Uma falha é um par (*s*, *X*), onde *s 2* traces(*P*) e *X 2* refusals(*P*/*s*) (refusals(*P*) é o conjunto dos eventos recusados por *P*). failures(*P*) é o conjunto de todas as falhas de *P*. As falhas de um processo em CSP são definidos de forma composicional, em termos dos operadores. Como exemplo, apresentamos as falhas de alguns operadores básicos:

1. failures(STOP) = { (< >, X) | X ⊆ $\Sigma^{\surd}$}
2. failures(SKIP) = { (< >, X) | X ⊆ Σ } U { (< >, X) | X ⊆ $\Sigma^{\surd}$}
3. failures (P |~| Q) = failures (P) U failures (Q)

2.3.3 Modelo de Falhas e Divergências

O modelo de falhas inclui os modelos de traces e falhas. Ele permite a investigação mais completa dos comportamentos de um processo. Além de também permitir investigar o que um processo pode fazer e identificar as falhas do processo, ele permite investigar as divergências do processo. Um processo diverge quando ele está realizando eventos invisíveis ao ambiente. O ambiente não consegue diferenciar se o processo parou ou divergiu, pois não consegue enxergar os eventos.

Quando um processo diverge, é assumido que ele pode recusar qualquer evento, rejeitar qualquer evento e sempre divergir mesmo após voltar a se comunicar com o ambiente. Então o conjunto divergences(P) contém os traces s nos quais P pode divergir e as suas extensões (s^t). Este conjunto também pode ser obtido através de regras. Algumas delas são:

1. divergences(STOP) = { }

2. divergences(SKIP) = { }
3. divergences (P |~| Q) = traces(P) U traces(Q)
4. divergences (P [] Q) = traces(P) U traces(Q)

O conjunto de falhas é estendido para capturar a idéia de um processo poder falhar após ter divergido. O conjunto estendido é definido por:

$$failures_{\perp}(P) = failures(P) \cup \{ (s, X) \mid s \in divergences(P)\}$$

A representação de um processo no modelo de falhas e divergências consiste na tupla:

$$(failures_{\perp}(P) , divergences(P))$$

Este modelo é o único que descreve completamente o comportamento de um processo.

A ilustração a seguir, mostra a relação das expressões usadas para construir os conjuntos que representam cada um dos modelos.

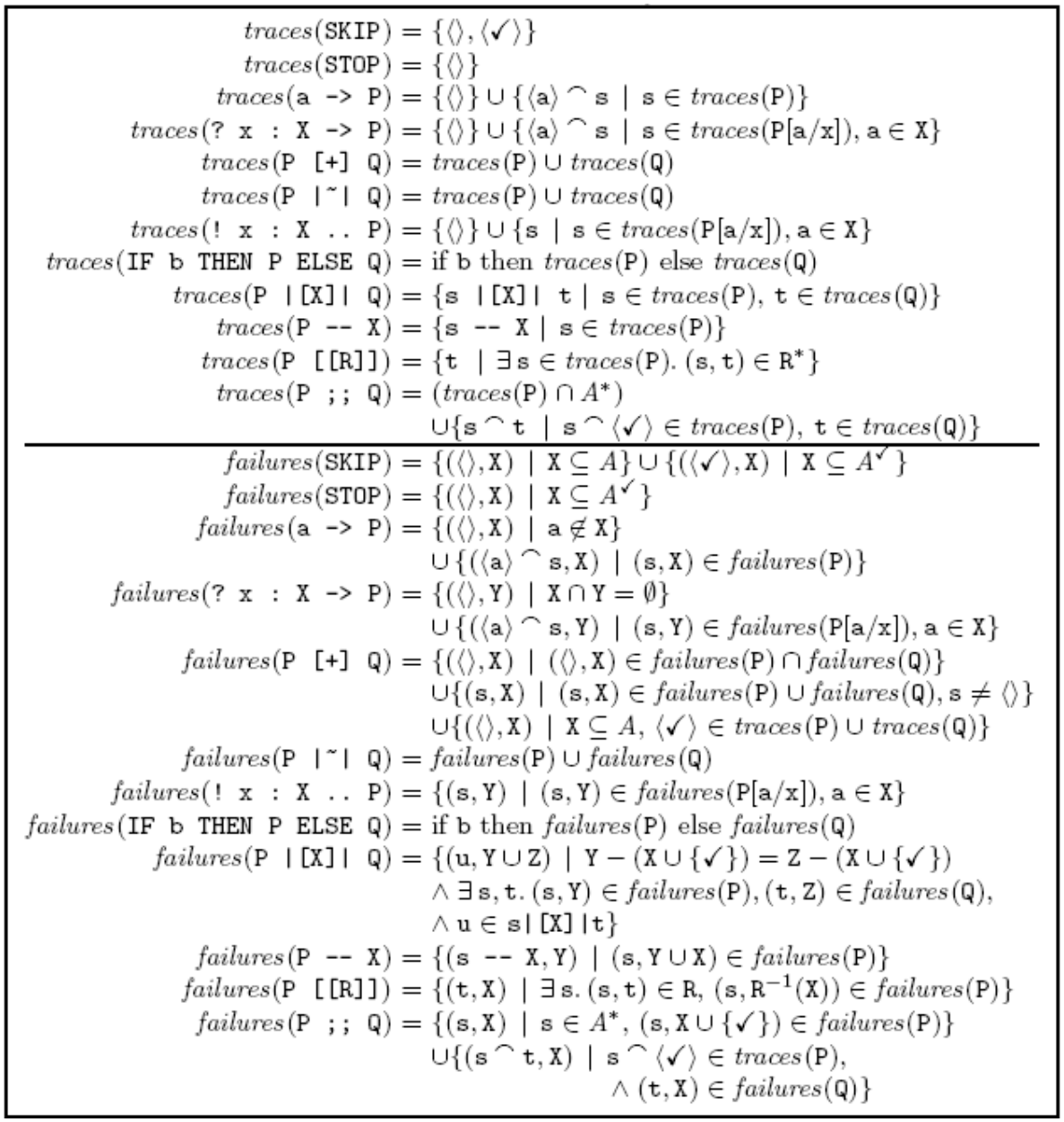

Figura 2.3 Tabela de cláusulas semânticas

2.4 FERRAMENTAS

A ferramenta mais conhecida para prova de refinamentos sobre processos CSP é o FDR (Failures and Divergences Refinament) [24]. O ProBE [25] é outra ferramenta útil para animar modelos CSP. Ambas lêem especificações CSP descritas em uma linguagem funcional chamada CSPM [24], que é um acrônimo para machine readable CSP. Esta linguagem tem sintaxe adaptada para representar as construções de CSP, com pequenas variações. Os canais de comunicação, por exemplo, devem ser explicitamente declarados, assim como o tipo dos dados que estes podem transmitir em seus eventos. É possível

inclusive definir canais multidimensionais para transmitir mais de um dado simultaneamente.

Através do ProBE, ferramentas animadora, o usuário pode fazer o papel de ambiente e escolher um dos eventos que o processo está tentando comunicar a cada passo. Na verdade, ela permite inclusive que o usuário resolva as escolhas internas. Afigura abaixo ilustra um exemplo de animação da ferramenta.

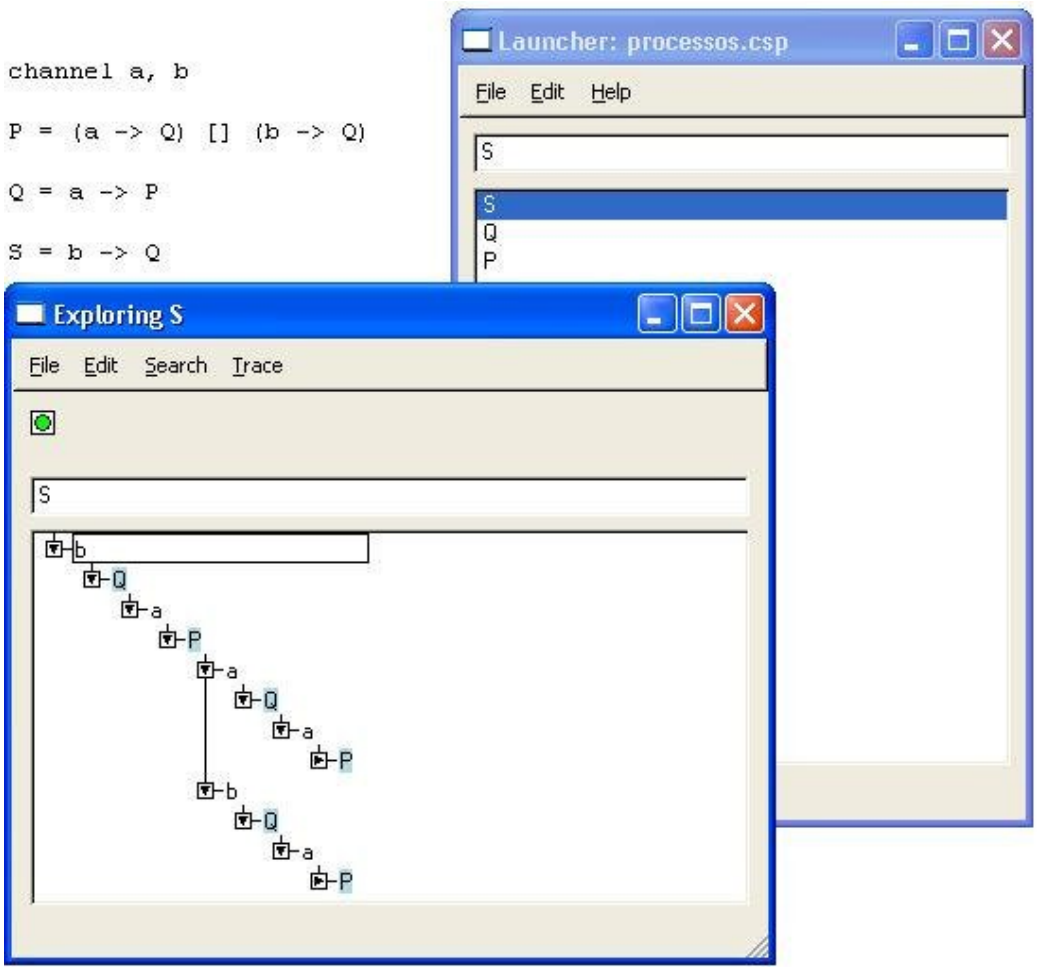

Figura 2.4 Ferramenta ProBE

A ferramenta FDR possui um outro propósito. Com ela o usuário pode testar afirmações sobre processos, como equivalências entre processos e existência de deadlocks, livelocks ou não-determinismo.

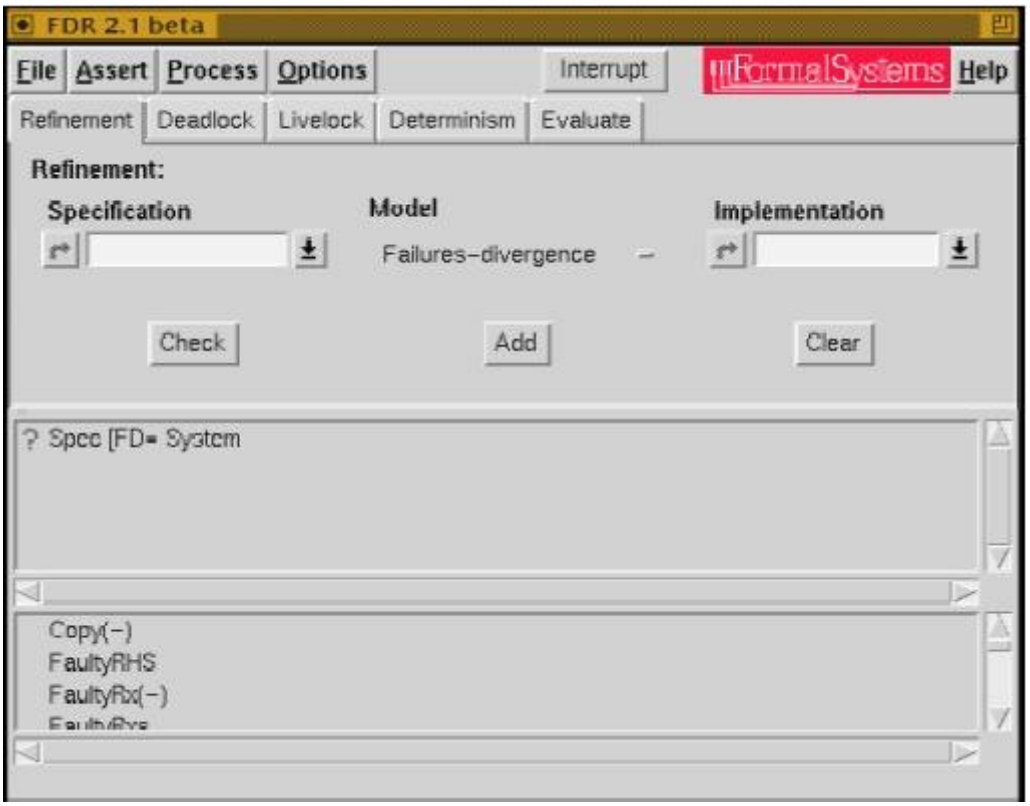

Figura 2.5 Ferramenta FDR

CAPÍTULO 3

LINGUAGEM NATURAL CONTROLADA

Detalharemos agora, o uso da Linguagem Natural Controlada (CNL), que pode ser vista como uma versão restringida e processável do inglês, sendo usada para escrever as etapas do caso de uso, tornando possível à realização de validações e transformações.

A gramática de CNL é basicamente um sub-conjunto da gramática inglesa. A construção de suas sentenças contém verbos, termos, e modificadores específicos relativos ao domínio da aplicação. A construção das frases é centrada no verbo. Os termos e os modificadores do domínio são combinados a fim fazer regras temáticas em torno do verbo [27]. Esta estratégia é detalhada em [26] onde isto foi usado para traduzir sentenças dos casos de teste em construções de CSP.

As seguintes seções descrevem as bases de conhecimentos usadas para armazenar os vocábulos envolvidos na definição da CNL. Os exemplos ilustrados já se encontram no domínio do sistema de controle de portas do metrô. Em nosso trabalho, estas bases de conhecimento são usadas por nossa proposta a fim conseguir a tradução das sentenças dos casos de uso da aplicação em estudo em construções de CSP.

3.1 SÍMBOLOS LÉXICOS

Os símbolos léxicos representam os vocábulos que podem aparecer nas sentenças de uma CNL. Cada um vocábulo pode ser um verbo, um termo, ou um modificador. As seguintes subseções descrevem cada destes vocábulos.

3.1.1 Verbo

Um verbo é usado para definir uma ação realizada pelo usuário, para dar uma descrição do estado do sistema e da resposta do sistema, ou para especificar uma mensagem. As ações são descritas com comandos imperativos, tais como uma indicação ao usuário ou para o componente realizar alguma operação.

```
<verb>
   <name>          </name>
   <third-person> </third-person>
   <past>          </past>
   <participle>    </participle>
   <gerund>        </gerund>
</verb>
```

Figura 3.1 Esquema para definição de verbo

A figura é a estrutura de XML [28] usada para definir verbos em CNL. A definição do verbo contém as formas possíveis do verbo, tais como o presente ou passado. O tag *name* contém o verbo na forma infinitiva. A forma no infinitivo é usada em construções de sentenças no presente. Como a forma da terceira pessoa do singular pode variar, o tag *third-person* define-a. O tag *gerund* contém a forma do verbo no gerúndio (- ing). Além disso, o tag *past* define a forma do verbo no passado e o tag *participle* sua forma no particípio. A figura a seguir contém alguns exemplos de definições de verbo.

```
<verb>
        <term>start</term>
        <past>started</past>
        <participle>started</participle>
        <gerund>starting</gerund> <thirdperson/>
</verb>
<verb>
        <term>press</term>
        <past>pressed</past>
        <participle>pressed</participle>
        <gerund>pressing</gerund> <thirdperson/>
</verb>
```

Figura 3.2 Exemplo de definição de verbo

3.1.2 Termo

Um termo é um elemento, ou entidade, do domínio da aplicação. Pode ser simplesmente um substantivo ou um substantivo combinado com os adjetivos ou com outros substantivos. Ele é visto como um objeto do domínio da aplicação que seja manipulado de algum modo pelo usuário ou por componentes durante todo a execução do caso de uso.

```
<noun>
      <term></term>
      <plural></plural>
      <model></model>
      <class></class>
</noun>
```

Figura 3.3 Esquema para definição de termo

A figura é a estrutura de XML usada para definir um termo. O tag *term* é o próprio nome do termo. Define a forma do termo no singular. O tag *plural* contém o a forma do plural do termo. Além disso, o tag *model* contém a representação do código em CSP do termo. Finalmente, o tag *class* define a classe da Ontologia a qual ele pertence, determinando como o termo se relaciona com outros termos. Esta representação é usada para definir valores do datatype em CSP. A figura abaixo possui alguns exemplos de definições de termo.

```
<noun>
      <term>kloan key</term>
      <plural>kloan keys</plural>
      <model>KLOAN_KEY</model>
      <class>key</class>
</noun>
<noun>
      <term>train speed</term>
      <plural/>
      <model>TRAIN_SPEED</model>
      <class>variable_item</class>
</noun>
```

Figura 3.4 Exemplos de definição de termo

Como pode ser visto na figura, o termo *kloan key*, por exemplo, tem o plural definido como *kloan keys* e pertence à classe key da Ontologia. Conseqüentemente, *kloan key* é tratado como uma key pelos verbos que se referem a esta classe. Finalmente, seu código de CSP é definido como KLOAN_KEY.

3.1.3 Modificador

Um modificador pode ser qualquer coisa que qualifica um termo, tal como um adjetivo ou um advérbio. Pode ser até mesmo um substantivo, uma vez que os substantivos

podem detalhar características dos termos. Abaixo temos a estrutura em XML [28] usada para definir os modificadores.

```
<modifier>
   <name>          </name>
   <position>      </position>
   <precedence>    </precedence>
   <number>        </number>
   <article>       </article>
   <model>         </model>
</modifier>
```

Figura 3.5 Esquema para definição de modificador

Mais uma vez, o tag *name* especifica o nome do modificador. O tag *position* define se o modificador vai estar antes ou depois do termo. O tag *precedence* especifica a ordem da prioridade entre os modificadores. O tag *number* é usado para determinar se o modificador concorda com um termo no singular ou plural. Já o tag *article* define se o modificador pode ser precedido por um artigo definitivo ou indefinido. Finalmente, o tag *model* contém a representação do código em CSP do modificador, assim ele é usado para definir valores do datatype em CSP.

```
<modifier>
        <term>all</term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>ALL</model>
</modifier>
<modifier>
        <term>less than <int/></term>
        <position>before</position>
        <precedence>0</precedence>
        <numberinflection>plural</numberinflection>
        <article>no</article>
        <model>LESS_THAN.Int</model>
</modifier>
<modifier>
        <term>with <int/> seconds</term>
        <position>after</position>
        <precedence>0</precedence>
        <numberinflection>singular</numberinflection>
        <article>no</article>
        <model>WITH_N_SECONDS.Int</model>
</modifier>
```

Figura 3.6 Exemplos de definição de modificadores

A figura anterior ilustra três definições do modificador. O modificador *with <int/> seconds* é usado junto com um inteiro e uma palavra fixa. Para compreender melhor este modificador, considere como exemplo a sentença *Start the temporization flag with 10 seconds*. O número 10 é o inteiro que é seguido com a palavra *seconds* que é o outro item requerido pela construção do modificador.

3.2 ONTOLOGIA

Conforme já mencionado anteriormente, cada domínio da aplicação tem elementos e entidades específicos. São criados termos e agrupados em classes de acordo com suas características. Estas classes podem também ser relacionada por herança. Uma Ontologia define as classes gramaticais e suas hierarquias [5].

```
<ontology>
    <class>
       <description>Generic Class</description>
       <name>Object</name>
       <code>object</code>
       <subclasses>
          <class>
             <description>Represents a generic value</description>
             <name>Value</name>
             <code>value</code>
             <subclasses>
                <class>
                   <description>Represents a state value, e. g.,
                               "enabled", "ON", "active". </description>
                   <name>State Value</name>
                   <code>state_value</code>
                   <subclasses />
                </class>
                <class>
                   <description>Represents a value used to fill a field,
                            e. g., "speed".</description>
                   <name>Field Value</name>
                   <code>field_value</code>
                   <subclasses />
                </class>

                <class>
                   <description>Represents a position value, e. g., "left", "right".</description>

                   <name>Position Value</name>

                   <code>position_value</code>

                   <subclasses />
                </class>
             </subclasses>
          </class>
...
```

Figura 3.7 Fragmento de uma Ontologia

A figura ilustra um fragmento pequeno da Ontologia que define o *object*, o *value*, o *state value*, o *field value*, e o *position value*. As classes *state value*, *field value*, e *position value* herdam a classe *value*, que por sua vez herda a classe *object*. Na figura 3.4, o termo *kloan key* é uma key devido ao fato que pertence à classe key da Ontologia. Esta classe restringe a maneira com que os termos são combinados com os verbos para evitar sentenças inconsistentes nos casos de uso.

3.3 CASE FRAME

Um case frame define a relação entre verbos e termos, mais especificamente define como as classes da ontologia são combinadas com as definições complementares dos verbos. Cada case frame determina como um verbo pode ser usado para instanciar uma sentença. O formalismo da gramática do case [26] é usado a fim definir como os verbos são associados com os termos, que podem ser detalhados por modificadores. Cada termo faz o papel sobre uma determinada regra temática em torno do verbo; pode ser, por exemplo, um agente ou um tema da sentença. Cada case frame pode também ser associado a mais de um verbo, neste caso, todos assumem o mesmo significado (sinônimos).

```
<frame>
  <description></description>
  <name></name>
  <verb-list>
    <verb></verb>
    <verb></verb>
  </verb-list>
  <roles>
    <role mandatory=" ">agent</role>
    <role mandatory=" ">theme</role>
    <role mandatory=" ">from-value</role>
    <role mandatory=" ">to-value</role>
    <role mandatory=" ">from-loc</role>
    <role mandatory=" ">to-loc</role>
    <role mandatory=" ">at-loc</role>
    <role mandatory=" ">instrument</role>
  </roles>
</frame>
```

Figura 3.8 Esquema de definição de um case frame

A figura representa a estrutura em XML [28] usada para definir case frames. O tag de descrição serve para dá uma explanação breve sobre como os case frames podem ser usados, muitas vezes incluindo exemplos. O tag *name* identifica o case frame. O tag

verblist contém um conjunto dos tags de verbos que se referem aos verbos relacionados a este case frame. Os verbos desta lista devem ter o mesmo significado semântico. Conseqüentemente, possuem os mesmos argumentos definidos nas tags *role*. Cada tag *role* determina um argumento possível do verbo e seu tipo. Possui o atributo imperativo que determina se o argumento é obrigatório ou não. A seguir, seguem os possíveis tipos de *roles*:

- agent: um termo que execute a ação do verbo (agente ativo).
- theme: um termo que sofra a ação do verbo (agente passivo).
- from-value: um termo que determina o valor passado do *theme*.
- to-value: um termo que determine o valor futuro do *theme*.
- from-loc: um termo que determine a posição passado do *theme*.
- to-loc: um termo que determine a posição futura do *theme*.
- at-loc: um termo que determine a posição atual do *theme*.
- instrument: um termo que determine a maneira com que o agente executa a ação do verbo.

Um exemplo de uma definição de case frame é mostrado a seguir. O *SelectItem* representa o case frame definido pelos verbos select e choose. O agente e o tema são imperativos, assim necessita ser especificado quando estes verbos são usados. O *from-loc* é opcional. Dessa forma, não é necessário especificar este argumento.

```
<frame>
   <description>Set the value of an item.
   Example: Set the selector key position to right</description>
   <name>SetItem</name>
   <verblist>
      <verb>set</verb>
      <verb>check</verb>
   </verblist>
   <roles>
      <role mandatory="True">agent</role>

      <role mandatory="True">theme</role>

      <role mandatory="false">to-value</role>
   </roles>
</frame>
```

Figura 3.9 Exemplo de definição de um case frame

Para ilustrar como os verbos são associados com seus argumentos, alguns exemplos de sentenças de CNL são mostrados na tabela abaixo.

Close the door with signalling.	Close **<theme> <instrument>**
Press the closing button	Press **<theme>**
Return to initial operation.	Return **<to-loc>**
All doors **are** opened.	**<agent>** is **<to-value>**

Tabela 3.1 Exemplos de sentenças em CNL.

3.4 CASE FRAME RESTRICTIONS

O case frame restriction define a relação entre os argumentos do verbo e as classes da Ontologia. O argumento de cada verbo pertence a uma classe da Ontologia a fim restringir na maneira com que as frases são escritas. Isto minimiza a possibilidade de escrever sentenças semanticamente erradas. A seguinte figura mostra a estrutura em XML usada para definir um case frame restriction.

```
<frame>
  <name></name>
  <restrictions>
    <restriction name=" ">
      <class role=" "> </class>
    </restriction>
    <restriction name=" ">
      <class role=" "> </class>
    </restriction>
  </restrictions>
</frame>
```

Figura 3.10 Esquema de definição de um case frame restriction

O tag *frame* define uma restrição do case frame e sua identificação é fornecida pelo tag *name*. O esquema contém tags *restriction* que envolvem todas as restrições possíveis. Cada tag *restriction* contém um nome do atributo para identificá-lo e uma lista de tags *class*. Cada tag *class* define a classe da Ontologia associada ao argumento *role* do verbo. As figuras a seguir contêm a definição do case frame SetItem para os verbos *set* e *check* (Figura 3.11), e sua respectiva restrição de case frame (Figura 3.12).

```
<frame>
   <description>Set the value of an item.
   Example: Set the selector key position to right</description>
   <name>SetItem</name>
   <verblist>
      <verb>set</verb>
      <verb>check</verb>
   </verblist>
   <roles>
      <role mandatory="True">agent</role>

      <role mandatory="True">theme</role>

      <role mandatory="false">to-value</role>
   </roles>
</frame>
```

Figura 3.11 Definição do case frame SetItem

```
<frame>
   <name>SetItem</name>
   <restrictions>
      <restriction name="DTSET_FIELDVALUE_FIELD">
         <class role="to-value">field_value</class>
         <class role="theme">field</class>
      </restriction>
      <restriction name="DTSET_STATEVALUE_ITEM">
         <class role="to-value">state_value</class>
         <class role="theme">item</class>
      </restriction>
       <restriction name="DTSET_STATEVALUE_KEY">
         <class role="to-value">state_value</class>
         <class role="theme">key</class>
      </restriction>
      <restriction name="DTSET_POSITIONVALUE_KEY">
         <class role="to-value">position_value</class>
         <class role="theme">key</class>
      </restriction>
      <restriction name="DTSET_ITEM">
         <class role="theme">item</class>
      </restriction>
   </restrictions>
</frame>
```

Figura 3.12 Definição do case frame restriction SetItem

Como pôde ser observado, o case frame SetItem contém os roles: *agent*, *theme* e *to-value*. Assim, para aquelas três regras, há cinco restrições definidas.

As quatro primeiras restrições restringem os argumentos *theme* e *to-value*, e o quinto restringe somente o argumento *theme*, uma vez que o argumento *to-value* não é obrigatório. Cada restrição tem um nome, que será usado para definir um datatype em CSP. Para conciliar a definição da restrição, é necessária a associação de cada role a uma classe da ontologia. Esta associação restringe os argumentos do verbo. Por exemplo, a restrição DTSET_FIELDVALUE_FIELD define que o *theme* é um termo de um campo da classe da Ontologia e o argumento do *to-value* pertence ao campo *field_value* da classe.

CAPÍTULO 4

GERAÇÃO DO MODELO DE CSP

Este capítulo descreve a estratégia de automação utilizada para traduzir especificações de requisitos do sistema CGP do metrô escritas numa CNL em modelos de CSP correspondente aos casos de uso desse sistema na visão de componente. A estratégia de conversão é fundamentada em [5], a qual mostra com sucesso a utilização dessa abordagem para uma aplicação de celular. O fato de essa estratégia ser adaptável para diferentes tipos sistemas tornou interessante nossa adoção. Assim, ela será utilizada para a análise do nosso sistema alvo (capítulo 5).

Como apresentado na figura abaixo, para a geração do modelo formal de um sistema faz-se necessário realizar diversas atividades. O processo começa com a análise dos documentos de requisitos. Estes documentos são escritos geralmente em uma maneira informal, determinando o que o sistema deve fazer, mas não como. Estas especificações não são escritas e não são estruturadas em uma maneira fixa; definem apenas as necessidades do cliente.

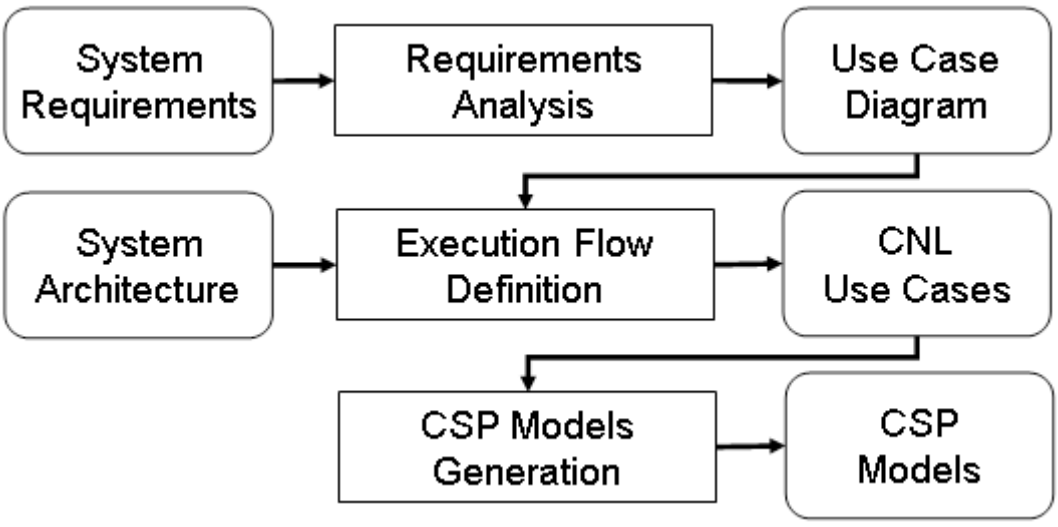

Figura 4.1 Passos da Solução

Após este nível de abstração, mediante a análise e entendimento do sistema, é criada a Gramática em Linguagem Natural Controlada a partir do Inglês. Ela servirá para a

definição dos casos de uso (próximo passo) implementados em templates numa visão por componentes.

Uma vez criado os casos de uso através de templates e da gramática em CNL, é possível gerar especificações formais a partir deles. Particularmente, nosso alvo é um modelo na álgebra processos de CSP. Assim como foi feito em [5], o modelo a ser gerado deve seguir a sintaxe de CSPm que é uma representação textual de CSP. Essa é a sintaxe padrão aceitada por ferramentas de CSP como o FDR.

4.1 GERAÇÃO DA GRAMÁTICA EM CNL

A geração da gramática em CNL é baseada no contexto do sistema e trata-se basicamente de um subconjunto da gramática inglesa. A construção das sentenças contém verbos, termos, e modificadores específicos do domínio.

Os itens de um caso de uso (ação do usuário, estado do sistema, resposta de sistema, e mensagem) são escritos nessa língua natural controlada (CNL) com uma gramática fixa, definida pela base de conhecimento. As sentenças dos casos de uso devem aderir à gramática de CNL, logo, os designers têm que saber a gramática de CNL.

O uso de CNL além de tornar os casos de usos legíveis e uniformes, também traz a possibilidade de processá-los a fim gerar construções de CSP [26].

O detalhamento do processo de geração da gramática já foi mencionado no capítulo anterior, onde ilustramos alguns conceitos usados na nossa estratégia.

4.2 CRIAÇÃO DOS CASOS DE USO

Após o nível inicial de abstração expostos nos documentos de requisitos, os requisitos são analisados e os casos de uso do sistema são definidos. A fim conseguir a cobertura total dos requisitos, os casos de uso são definidos depois que os requisitos são agrupados por similaridade.

Os casos de uso capturam o comportamento do sistema possibilitando diversos níveis de abstrações. Dependendo da necessidade do desenvolvedor, eles podem ser criados para diferentes propósitos.

Nesta seção, nós apresentamos os templates da especificação do caso de uso na perspectiva de componentes do sistema. Esses templates definem os fluxos da execução que determinam a interação entre o usuário e o sistema. A língua natural controlada (CNL),

versão processável do inglês, é usada para escrever as etapas do caso de uso fazendo possível realizar validações e transformações.

4.2.1 Template do Caso de Uso na Visão de Componentes

A visão de casos de uso por componentes especifica o comportamento do sistema baseado na interação do usuário com componentes do sistema. Nesta visão, o sistema é decomposto em componentes que concorrentemente processam os pedidos do usuário e comunicam entre si. A figura a seguir apresenta o template desse caso de uso. Seus campos contêm os comentários que explicam como o template deve ser preenchido. Os itens subseqüentes explicam este template em detalhes.

Feature <id>

UC <id> - <Use Case Name>

Main Flow

From Steps: <from where this flow starts (step ids)>
To Step: <to where this flow goes (step id)>

Step Id	Sender	Message	System State	Receiver
<Step id>	<Component that sends message>	<Message sent from sender component to receiver component>	<system state related to the specified message>	<Component that receives message>

Alternative Flows

From Steps: <from where this flow starts (step ids)>
To Step: <to where this flow goes (step id)>

Step Id	Sender	Message	System State	Receiver
<Step id>	<Component that sends message>	<Message sent from sender component to receiver component>	<system state related to the specified message>	<Component that receives message>

Exception Flows

From Steps: <from where this flow starts (step ids)>
To Step: <to where this flow goes (step id)>

Step Id	Sender	Message	System State	Receiver
<Step id>	<Component that sends message>	<Message sent from sender component to receiver component>	<system state related to the specified message>	<Component that receives message>

Figura 4.2 Template de caso de uso na visão de componente.

- *Features*: Os casos de uso são agrupados inicialmente para formar a uma característica. Cada característica contém um número de identificação. Este agrupamento é conveniente para outras finalidades da organização, mas não é obrigatória sua aplicação no template proposto para o caso de uso.
- *Execution Flow*: Geralmente um caso de uso pode ser usado para especificar cenários diferentes, dependendo das entradas do usuário e das ações. Assim, cada fluxo de execução representa um trajeto possível (uma seqüência das etapas) de que o usuário ou sistema pode fazer. Os seguintes itens descrevem os componentes do fluxo da execução.
- *Set*: A tupla (sender, message, system state, receiver) é chamada de etapa. Cada etapa é identificada através de um identificador e descreve a ação de envio de uma mensagem por parte de um componente remente; dependendo da natureza do sistema pode ser esta mensagem pode ser tão simples quanto pressionando alguma tecla ou informar uma operação mais complexa, tal como imprimir um relatório. O estado do sistema é uma condição na configuração de sistema real imediatamente antes da mensagem ser recebida. Assim, pode ser uma condição de status atual da configuração (instalação) ou da memória da aplicação. O receptor é o componente que vai processar a mensagem baseando-se no estado atual do sistema.
- *FlowTypes*: Os fluxos de execução são categorizados como fluxos principal, alternativos ou de exceção. Geralmente, os fluxos principais de execução representam o trajeto feliz dos casos de uso, que é uma seqüência das etapas onde tudo trabalha como esperado. Os fluxos alternativos de execução representam uma situação bem escolhida. Durante a execução do fluxo principal, pode ser possível executar uma ação diferente da que foi especificada no fluxo principal e continuar a execução do caso de uso através de um trajeto diferente. Os fluxos de exceção de execução especificam os cenários de erro causados por dados de entrada inválidos ou por estados críticos do sistema. Os fluxos da alternativos e de exceção são relacionados estritamente às condições do estado do sistema. O sistema pode responder de maneira diferente para as mesmas ações de usuários.
- *Referências entre fluxos de execução*: Há umas situações em que um componente pode escolher entre trajetos diferentes. Quando isto acontece é necessário definir um fluxo da execução para cada trajeto. Cada fluxo da execução tem um ponto de começo, ou o estado inicial, e um estado final. O ponto de começo é representado

pelo do campo *From Steps* e do estado final pelo *To Step*. O *From Steps* pode aceitar mais de um valor, significar que este fluxo pode ser acionado por fontes diferentes. Quando isto acontece, o fluxo de execução especificado pode ser executado depois que uma das etapas é executado. Além disso, o *To Step* referência somente uma etapa do fluxo de execução, fazendo possível o reuso de etapas de fluxo de execução ou até mesmo, definir laços.

O uso de especificações de caso de uso para definir os requisitos do sistema serve para projetar o comportamento do sistema através de seqüências de etapas. O uso de tal estratégia fornece meios para definir as possíveis ações do usuário e a respectiva resposta do sistema. A seqüência de tais pares é fundamental para determinar o comportamento do sistema. Além disso, é possível também descrever casos no qual o sistema pode se comportar de maneira diferente da esperada; estes casos são modelados através da execução dos fluxos alternativos ou de exceções.

4.3 TRADUÇÃO PARA CSP

Na seção anterior mostramos que as sentenças dos casos de uso são escritas de acordo com a gramática de CNL, que é definida por bases de conhecimento. As seguintes subseções definem o alfabeto de CSP e os métodos usados para traduzir as sentenças de CNL em eventos de CSP, baseados neste alfabeto. Por fim é mostrada a abordagem de mapeamentos dos casos de uso em processos.

4.3.1 Alfabeto de CSP

Usando as bases de conhecimento de CNL como uma gramática e uma definição de dicionário é possível definir os *datatypes* e os *channels* de CSP. Estes elementos de CSP definem o alfabeto do modelo de CSP; relacionando-os à gramática de CNL. Os verbos, os termos, e os modificadores são traduzidos em possíveis *eventos* e *datatypes* de CSP que são usados mais tarde gerar o modelo de CSP. Para compreender a criação do alfabeto de CSP, uma parte dele será apresentada a seguir.

Mapeamento de Classes de Ontologia em Datatype de CSP

Para cada classe de Ontologia, há um datatype específico de CSP associado. Como pode ser visto em (Figura 4.3 e 4.4), as classes de Ontologia são mapeadas em datatypes de CSP. Os elementos do datatype referem-se às subclasses de uma classe maior. Assim, torna-se claro ver que o mesmo nível da hierarquia estabelecida na Ontologia pode ser definido nos datatypes de CSP.

```
<datatype class="value">
        <label/>
        <name>Value</name>
</datatype>

<datatype class="state_value">
        <label>STATEVALUE</label>
        <name>StateValue</name>
</datatype>

<datatype class="field_value">
        <label>FIELDVALUE</label>
        <name>FieldValue</name>
</datatype>
```

Figura 4.3 Classe de Ontologia.

```
datatype Value =
                DTSTATEVALUE.StateValue
              | DTFIELDVALUE.FieldValue

datatype StateValue =  state1 | state2 | ...

datatype FieldValue =  field1 | field2 | ...
```

Figura 4.4 Definição dos datatypes de CSP.

O datatype *Value*, considerando na figura, é um datatype de CSP que pode ter um *StateValue*, ou um *FieldValue.*

Mapeamento de Case Frame em Channel

As regras temáticas dos case frames são representadas como parâmetros de *channels* em CSP. Para cada regra e seu conjunto de restrições, que são definidas pelas case frame restrictions, os datatypes definidos são usados como parâmetros dos channels.

Assim, o channel de CSP que define o case frame mostrado em (Figura 3.11) pode receber três parâmetros, que especificam a regra temática: *agent*, *theme* e *to-value*. Os channels de CSP recebem tuplas em dois argumentos. O primeiro argumento é a regra temática e o segundo é o conjunto dos modificadores ({} em CSP).

As figuras a seguir (Figuras 4.5 e 4.6) mostram como um channel de CSP é definido. O conjunto de channel de CSP foi definido baseado no case frame ilustrado em (Figura 3.11) e pode receber dois tipos dos parâmetros: para uma ou duas tuplas.

```
<channel>
  <name>set</name>
  <case-grammar>SetItem</case-grammar>
  <datatype>DTSet</datatype>
<channel>
```

Figura 4.5 Definição do channel a partir do exemplo em CNL

```
channel Set : DTSet
datatype DTSet =
              DTSET_ITEM.(Item, Set(Modifier))
            | DTSET_ITEM.(Item, Set(Modifier)).(Item, Set(Modifier))
```

Figura 4.6 Definição do channel em CSP

Usando a definição em XML (case frame), o channel em CSP é gerado. O datatype DTSet é criado automaticamente depois de avaliar os case frame restrictions. Vale observar que os nomes das restrições são conservados na definição do channel.

4.3.2 Geração de Eventos de CSP

Na seção anterior, as bases de conhecimento em CNL são empregadas na definição do alfabeto de CSP. Usando este alfabeto é possível traduzir cada sentença dos templates de casos de uso nos eventos de CSP, que são usados para definir modelos de uso dos casos de uso.

A próxima figura apresenta as sentenças em CNL que estão de acordo com o case frame SetItem. A estrutura da sentença *Set the selector key to enabled* é baseada no case frame restirction DTSET_STATEVALUE_KEY. Esta restrição permite que o verbo ajuste-se para aceitar dois argumentos: o valor SELECTOR KEY da classe *key* e o valor ENABLED da classe *state_value*. A estrutura das sentenças pode ainda ser caracterizada

por um conjunto de modificadores; na sentença *Set all doors to opened*, o *item* DOORS é mudado pelo modificador ALL.

```
-- Set the selector key to enabled
   set.DTSET_STATEVALUE_KEY.(ENABLED_STATE_VALUE, {}).(SELECTOR_KEY, {})
-- Set all doors to opened
   set.DTSET_STATEVALUE_ITEM.(OPENED_STATE_VALUE, {}).(DOOR_ITEM, {ALL})
-- Set the selector key position to right
   set.DTSET_POSITIONVALUE_FIELD.(RIGHT_POSITION_VALUE, {}).(SELECTOR_KEY_POSITION, {})
```

Figura 4.7 Exemplo de sentenças em CNL e suas traduções em CSP

Após essas etapas de mapeamento, resta a geração do modelo de CSP a partir dos templates dos casos de uso para finalizar a especificação em CSP do sistema. A seção a seguir explica a estratégia usada para gerar os modelos na visão de componente.

4.3.3 Geração do modelo de CSP na visão de componente

A geração do modelo de componente é realizada com a tradução de cada caso de uso. Cada caso de uso deve conter pelo menos um fluxo da execução, que é o fluxo principal; os fluxos alternativos e de exceção não são obrigatórios. Como explicado na seção 4.2, cada fluxo contém uma lista de etapas, e cada uma destas etapas é traçada a um processo de CSP. O processo é definido pela identificação da etapa, que é um identificador único entre todas as etapas. O corpo do processo é definido pelos eventos gerados pela comunicação dos componentes, pelo estado do sistema, e dos campos da mensagem do sistema. Cada um destes campos é descrito com uma ou mais sentenças no formato em CNL.

Pelo fato de cada etapa dos fluxos da execução terem uma identificação única, é possível usar sua identificação para definir os nomes dos processos. Cada um destes processos do caso de uso é composto seqüencialmente com o processo do sistema. Esta composição define que uma vez que a execução do caso de uso é terminada (SKIP) é possível começar uma execução nova do sistema do começo; com um outro caso de uso.

Na seguinte figura é definido o datatype *ComponentElement* que é um possível componente para essa visão. Há também o nametype de *ComponentView*, que é a composição de dois ComponentElement, o remetente e o receptor na troca da mensagem.

O canal setComp é um exemplo de uma definição channel; todos eventos restantes são definidos em uma maneira similar.

```
include "CSP_HEADER_COMPONENT.csp"

datatype ComponentElement = USER | TRAIN_COMMAND_APP |
                            TRAIN_CONTROLLER | TRAIN_LINE_COMMAND | TIMER

nametype ComponentView = ComponentElement.ComponentElement

channel setComp : ComponentView.DTSet
```

Figura 4.8 Parte do alfabeto do modelo de componente em CSP

Cada processo é definido como a escolha externa de CSP entre todas as primeiras etapas do fluxo de execução com o fluxo cujo campo *From Step* foi definido como START. Geralmente, cada um dos casos de uso tem somente um ponto de início; conseqüentemente, cada componente tem sub-processos para cada caso de uso.

A tarefa final é ligar completamente estes processos. Esta ligação é definida dependendo da posição da etapa. Se houver um fluxo alternativo ou de exceção no caso de uso, as etapas referenciadas no campo *From Step* são usadas para identificar de onde o fluxo alternativo ou de exceção se origina. O efeito resultante no processo do componente é a adição do operador de escolha externa de CSP, após a etapa especificada no campo From Step, referenciando assim a etapa do fluxo do alternativo ou de exceção à primeira.

Na figura a seguir, no fim do fluxo principal do processo TRAIN_COMMAND_APP, há uma referência aos processos TRAIN_COMMAND_APP_UC_01_8M e TRAIN_COMMAND_APP_UC_01_1A_1 a fim de liga o fluxo principal e aos fluxos alternativos e de exceção.

```
TRAIN_COMMAND_APP_P =
    TRAIN_COMMAND_APP_UC_01  [] TRAIN_COMMAND_APP_UC_02

-- Scenario Case: Execute the command "open" for all doors of a determined position

TRAIN_COMMAND_APP_UC_01 =
    -- Message: Set the selector key to enabled.
    setComp.USER.TRAIN_COMMAND_APP.DTSET_STATEVALUE_KEY.(ENABLED_STATE_VALUE, {}).
                                (SELECTOR_KEY, {}) ->
    -- Message: Set the selector key position to right
    setComp.USER.TRAIN_COMMAND_APP.DTSET_POSITIONVALUE_VARIABLEITEM.(RIGHT_POSITION_VALUE, {}).
                                (SELECTOR_KEY_POSITION, {}) ->
    -- Message: Select the "preparation option" from kloan key
    selectComp.USER.TRAIN_COMMAND_APP.DTSEL_STATEVALUE_KEY.(PREPARATION_STATE_VALUE, {OPTION}).
                                    (KLOAN_KEY, {}) ->
    -- Message: Start the open doors action
    startComp.TRAIN_COMMAND_APP.TRAIN_CONTROLLER.DTSTA_ACTION.(OPEN_ACTION, {})->
    -- Message: Check the train speed.
    checkComp.TRAIN_CONTROLLER.TRAIN_COMMAND_APP.DTCHE_VARIABLEITEM.(TRAIN_SPEED, {}) ->
    (TRAIN_COMMAND_APP_UC_01_9M [] TRAIN_COMMAND_APP_UC_01_1E)

TRAIN_COMMAND_APP_UC_01_9M =
    --Message: The train speed is less than 6.
    isstateComp.TRAIN_COMMAND_APP.TRAIN_CONTROLLER.DTISS_VARIABLEITEM.(TRAIN_SPEED, {LESS_THAN.6}) ->
    --Message: All doors are opened.
    isstateComp.TRAIN_CONTROLLER.TRAIN_COMMAND_APP.DTISS_ITEM_STATEVALUE.(DOOR,{ALL}).(OPENED_STATE_VALUE, {}) ->
    -- Message:  The led of respective operation position is lighted.
    lightComp.TRAIN_COMMAND_APP.USER.DTLIG_SIGNALLING_ITEM.(LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
    TRAIN_COMMAND_APP_P [] TRAIN_COMMAND_APP_UC_01_1A_1

TRAIN_COMMAND_APP_UC_01_1A_1 =
    -- Message: Change the selector key position from right to left.
    changeComp.USER.TRAIN_COMMAND_APP.DTCHA_POSITIONVALUE_VARIABLEITEM_POSITIONVALUE.(RIGHT_POSITION_VALUE, {}).
                                    (SELECTOR_KEY_POSITION, {}).(LEFT_POSITION_VALUE, {}) ->
    -- Message: Stop the open doors action
    stopComp.TRAIN_COMMAND_APP.TRAIN_CONTROLLER.DTSTO_ACTION.(CLOSE_ACTION, {})->
    -- Message: Choose the "unprepared option" from kloan key
    chooseComp.TRAIN_CONTROLLER.TRAIN_COMMAND_APP.DTCHO_STATEVALUE_KEY.(UNPREPARED_STATE_VALUE, {OPTION}).
                                    (KLOAN_KEY, {}) ->
    -- Message: The led of respective operation position is extinguished.
    extinguishComp.TRAIN_COMMAND_APP.USER.DTEXT_SIGNALLING_ITEM.(LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
    TRAIN_COMMAND_APP_P

TRAIN_COMMAND_APP_UC_01_1E =
    -- Message: The train speed is is greater than or equals to 6.
    isstateComp.TRAIN_COMMAND_APP.TRAIN_CONTROLLER.DTISS_VARIABLEITEM.
            (TRAIN_SPEED, {GREATER_THAN_OR_EQUALS.6}) ->
    -- Message: Choose the "unprepared option" from kloan key.
    chooseComp.TRAIN_CONTROLLER.TRAIN_COMMAND_APP.DTCHO_STATEVALUE_KEY.(UNPREPARED_STATE_VALUE, {OPTION})
                                                .(KLOAN_KEY, {}) ->
    -- Message: The led of respective operation position is extinguished.
    extinguishComp.TRAIN_COMMAND_APP.USER.DTEXT_SIGNALLING_ITEM.(LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
    TRAIN_COMMAND_APP_P
```

4.9 Exemplo do modelo de componente em CSP

Observe que os campos *From Step* e *To Step* são de extrema importância à construção modelo. Eles determinam quando o fluxo começa e termina, e as ligações entre as etapas. Lembrando que o campo *From Step* pode conter mais de uma etapa.

Se num fluxo o campo *From Step* for preenchido com a palavra chave START, o sistema pode iniciar a partir deste fluxo. Além disso, o processo deve parar, depois que a última etapa deste fluxo for executada, se campo *To Step* estiver preenchido com END. Isto é representado em CSP como o processo SKIP.

4.4 CONSIDERAÇÕES

Uma observação importante diz respeito às referências entre os fluxos de execução de um mesmo caso de uso ou casos de usos diferentes. Esta relação é originada pelos campos *From Step* e *To Step* e não se equivalem a uma generalização de UML, nem a uma inclusão ou extensão de caso de uso [29]. É uma maneira nova de relacionar a seqüência das etapas permitindo o seu reuso. Quando um fluxo é definido, ele pode partir de uma etapa do meio de um outro fluxo reusando as etapas precedentes desse fluxo. Similarmente, este fluxo pode terminar e referenciar algum outro fluxo ou até mesmo definir recursões. O uso de tal flexibilidade pode ser uma boa prática para escrever os casos de uso que encapsulam uma única funcionalidade, mas deve ser feito de forma cuidadosa, uma vez que a definição de demasiadas referências no meio dos fluxos pode tornar o modelo da especificação do caso de uso algo difícil de compreender.

Durante a geração do modelo, o operador de escolha externa de CSP é usado para permitir a execução do caso de uso escolhido pelo usuário. Neste caso, o usuário tem claramente a escolha entre executar um determinado caso de uso selecionado. Entretanto, o uso do operador de escolha externa de CSP nos fluxos alternativos ou de exceção parece ser uma tarefa subjetiva, pois a execução de um fluxo alternativo ou de exceção é habilitada por uma combinação de fatores (ação do usuário e estado do sistema), neste caso, o operador de escolha interna de CSP pode ser utilizado. Em todo o caso, como somente o modelo de traces está sendo abordado na estratégia para fiz de refinamento, a presença de não-determinismo no modelo é irrelevante.

O modelo na visão de usuário é muito mais simples de se gerar do que o modelo na visão de componente. Isto se deve ao fato dele não promover o paralelismo em sua definição. Além disso, o template do caso de uso na visão do usuário é mais simples de se usar. Por outro lado, usar o template de componente, além de desenvolver o conhecimento sobre arquitetura através do designer, requer uma visão abstrata do sistema; podendo ser visto como um conjunto de componentes. Entretanto, não é garantida a eficácia dessa estratégia para o sistema abordado em nosso trabalho. Na verdade, o que será feito no próximo capítulo é uma análise dessa abordagem a fim de se avaliar se ela pode ser aplicável ao sistema em estudo.

4.5 FERRAMENTAS

Para a automatizar o processo de geração de modelos de CSP um conjunto de ferramentas demonstradas em [5] pode ser utilizado.

A primeira ferramenta, NLP, é uma aplicação em Java que gera automaticamente a documentação da gramática de CNL [26] a partir das bases de conhecimento apresentadas. A gramática de CNL é gerada em páginas do HTML possibilitando aprender a sintaxe de CNL navegando sobre as definições da gramática.

A ferramenta *Use Case Validator* é um plugin do Microsoft Word 2003. Ela é usada para validar automaticamente as sentenças do caso de uso e relatando todas as inconsistências encontradas. Ela assegura que os casos de uso estejam escritos de acordo com template do caso de uso e a sintaxe de CNL. O MS Word 2003 é capaz estruturar o índice dos casos de uso com as definições dos schemas de XML. O plugin processa as sentenças do caso de uso para encontrar as inconsistências (frases não de acordo com a gramática de CNL).

Um módulo da NLP pode ser executado para automatizar a tradução dos casos de uso na visão por componentes em modelos de CSP. A aplicação lê o caso de uso a partir de um arquivo do Word 2003, verifica seu índice e gera os modelos do componente. o módulo NLP também é usado para obter os eventos de CSP a partir das sentenças de CNL. O uso da ferramenta de geração do modelo por si só executa a estratégia apresentada neste capítulo; estrutura os eventos de CSP, que são gerados eficazmente em processos para definir o modelo formal do sistema.

CAPÍTULO 5

ESTUDO DE CASO

A estratégia adotada nesse trabalho para geração de modelo formal a partir de requisitos foi devidamente apresentada e detalhada nos capítulos anteriores. Agora, focaremos nossos esforços em relatar a tentativa de aplicar essa estratégia a um sistema específico. No nosso caso, trata-se do sistema de Controle Geral de Portas (CGP) de Metrô utilizado para operar no projeto Santiago Linha 2, Chile.

A aplicação de todo o método científico em um ambiente real de projeto é fundamental para verificar a eficácia e os possíveis benefícios da estratégia. Além disso, a introdução de técnicas novas em um processo de desenvolvimento de software, tal como esse do Metrô, necessita ser analisada para estimar o tempo, a aplicação e a viabilidade num escopo pequeno, antes que a solução possa extensamente ser empregada.

Cada atividade que compõe a abordagem utilizada foi inserida em nosso estudo de caso. Alguns passos foram possíveis de serem implementados, mas impasses e deficiências foram encontrados. Assim, neste capítulo, apresentaremos as analises feitas sobre a aplicabilidade da estratégia utilizada. Nela, detalharemos os avanços obtidos, bem como problemas que ocorreram e quando possível, foram apresentados os possíveis caminhos ou soluções para as questões levantadas.

5.1 DESCRIÇÃO DO SISTEMA

O sistema será implementado usando a linguagem C e faz parte do equipamento de Controle Geral de Portas AeS-0617, armazenado e processado por um microcontrolador da linha PIC da família 18F da Microchip. A intenção é que o sistema entre em operação no projeto Santiago Linha 2, Chile.

O software, que é um componente da CGP (Controladora Geral de Portas), ele define a partir das entradas existentes a possibilidade de abertura das portas (momento e lado de abertura) e comando de periféricos do trem, além de intertravamentos de segurança com outros equipamentos. Além do trabalho mecânico de abrir/ fechar portas, o sistema também será responsável por emitir sinalizações (sonoras e luminosas) para seus operadores.

5.2 ESPECIFICAÇÃO DO SISTEMA

5.2.1 Característica

O software executará as funções de abertura e fechamento de portas (considerando as condições de intertravamentos com fatores de segurança impostos), além da execução de funções periféricas como troca de informações com o sistema de interface homem-máquina do trem, sinalizações e indicações.

O sistema se encarrega de, automaticamente, executar os intertravamentos necessários com as condições seguras de abertura e fechamento de portas, considerando inclusive intertravamentos entre o sistema de portas e de tração.

Esse software possui interface com o operador do trem pela cabine de comando, com os funcionários de manutenção através de conectores nas CGP que podem ser conectados à laptops providos do software de manutenção fornecido pela AeS, interfaces com o sistema TIMS (Train Information Monitoring System, sistema de monitoramento de informações para o condutor por tela presente nas cabines de condução) e com as os demais equipamentos do SCP (Sistema de Controle de Portas).

A CGP deve executar todos os intertravamentos necessários, entre os comandos do operador e as demais variáveis presentes no trem, de modo a não permitir ao operador executar certas manobras em condições que colocariam em risco a integridade dos passageiros.

O software não autoriza o operador a abrir portas com o trem em velocidade maior do que 6 Km/h.

O software possui modo de manutenção, onde sua segurança é temporariamente desabilitada, sendo os riscos envolvidos durante a manutenção responsabilidade do funcionário de manutenção envolvido.

5.2.2 Casos de Uso Abordados

As descrições de casos de uso abaixo foram extraídas do documento de requisitos do sistema SRS-0617-1 Controle Geral de Portas [17]:

- **Abrir portas**: Caso o trem estiver com chave seletora de lado habilitada e chave de abertura “KLOAN” selecionada em modo de preparação será habilitado um flag de

temporização de 10s. Caso a velocidade estiver abaixo de 6Km/h ou diminua de 6Km/h dentro desse intervalo e todas as condições anteriores permaneçam inalteradas, será executada abertura de portas no respectivo lado selecionado e o flag de temporização zerado. Caso o operador selecione a chave "KLOAN" em modo de despreparação dentro do período de 10s e antes do trem atingir velocidade inferior a 6Km/h o flag também será zerado. Caso o operador mude a posição da chave de seleção de lado de abertura após a abertura de portas, estas se fecharão sem a necessidade de comando.

- **Fecha portas (prioritária em relação à abertura)**: caso o trem estiver com chave seletora de lado habilitada e o operador pressionar o botão de fechamento de portas correspondente ao lado da operação e mantê-lo pressionado até o fim da operação, as portas se fecharão. Caso o botão for solto antes do fim da operação, o fechamento será interrompido. Caso o botão volte a ser pressionado em 7s, as portas se fecham sem sinalização de início de fechamento. Caso os 7s tenham sido ultrapassados, todo o ciclo de fechamento deverá ser reiniciado. Caso o operador mude a posição da chave de seleção de lado de abertura após a abertura de portas, estas se fecharão sem a necessidade de comando.

5.3 APLICAÇÃO E ANÁLISE DA ABORDAGEM

Detalharemos agora, o processo de aplicação da abordagem proposta para o sistema de Controle Geral de Portas (CGP). Como já foi mensionado, o processo consiste em 4 etapas:

- Análise dos requisitos e características do sistema
- Geração da Gramática em CNL
- Criação dos casos de Uso através do Template
- Tradução para CSP

Cada etapa foi seguida de maneira crítica e sempre voltada ao sistema em análise com objetivo de verificar em que pontos a abordagem se aplica ou não ao caso em estudo. A figura a seguir ilustra uma visão geral das atividades para a geração do modelo em CSP:

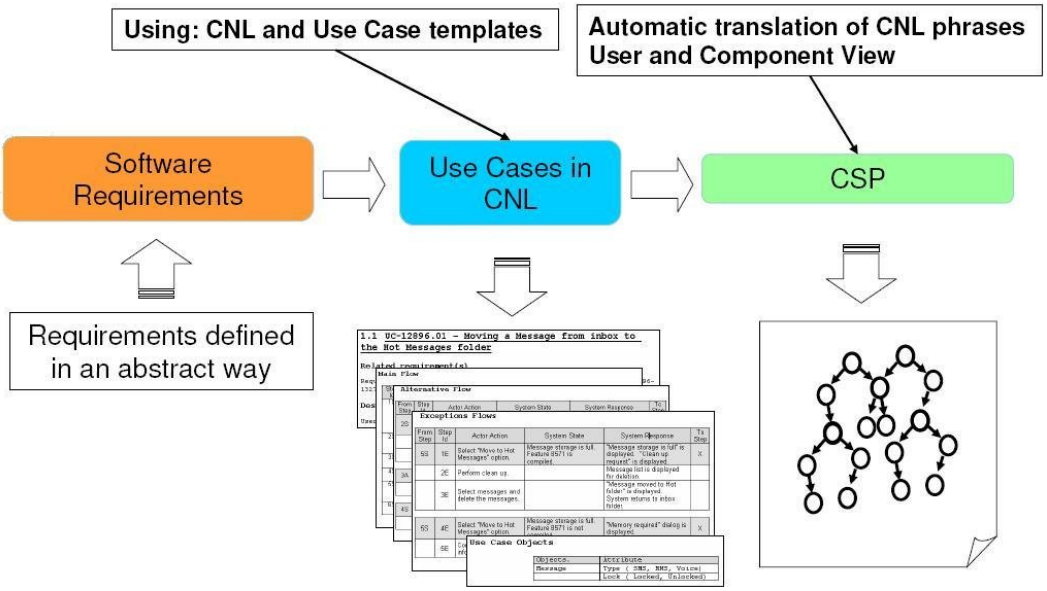

Figura 5.1 Visão Geral Processos para a geração do Modelo em CSP

5.3.1 Análise dos Requisitos e Características do Sistema

O processo se inicia com a análise dos requisitos do sistema. Em particular, tivemos dificuldades em compreender bem esses requisitos, pois os artefatos obtidos, os quais continham as informações sobre o sistema em questão, eram limitados. Entretanto, procuramos compreender ao máximo seu contexto e intenção de funcionamento. Além disso, como a equipe do projeto situa-se em outra cidade, a comunicação e entrevistas para extrair mais informações também foram prejudicadas.

Entretanto, o princípio básico de funcionamento dos casos de uso explorados foi consideravelmente esclarecido, de forma que sua implementação e análise foram feitas de maneira satisfatória.

Por se tratar de um sistema crítico, acreditamos que essas especificações devem ser melhoradas a fim de conter um nível de detalhamento maior e mais esclarecedor. Devem considerar também os padrões de qualidade de especificação e desenvolvimento para a melhoria do projeto.

5.3.2 Geração da Gramática em CNL

A geração da gramática em CNL foi baseada no contexto do sistema. Criamos basicamente um subconjunto limitado da gramática inglesa. A construção das sentenças contém verbos, termos, e modificadores específicos do domínio.

Por se tratar de um conjunto de estruturas em XML, a gramática foi implementada através de um editor para tal linguagem, a saber, o Context [34]. A figura a seguir ilustra alguns dos elementos pertencentes a gramática do CGP.

```
<noun>
  <term>right</term>
  <plural/>
  <model>RIGHT_POSITION_VALUE</model>
  <class>position_value</class>
</noun>

<noun>
  <term>train speed</term>
  <plural/>
  <model>TRAIN_SPEED</model>
  <class>variable_item</class>
</noun>

<verb>
  <term>light</term>
  <past>lighted</past>
  <participle>lighted</participle>
  <gerund>lighting</gerund> <thirdperson/>
</verb>

<frame>
  <description>Change an item from value1 to value2.
          Example: Change the selector key position
          from right to left</description>
  <name>ChangeItem</name>
  <verblist>
    <verb>change</verb>
  </verblist>
  <roles>
    <role mandatory="True">agent</role>
    <role mandatory="True">theme</role>
    <role mandatory="false">from-value</role>
   <role mandatory="false">to-value</role>
  </roles>
</frame>
```

```
<modifier>
  <term>less than <int/></term>
  <position>before</position>
  <precedence>0</precedence>
  <numberinflection>plural</numberinflection>
  <article>no</article>
  <model>LESS_THAN.Int</model>
</modifier>


  <frame>
        <name>SelectItem</name>
        <restrictions>
          <restriction name="DTSEL_ITEM">
            <class role="theme">item</class>
          </restriction>
          <restriction name="DTSEL_POSITION_VALUE">
            <class role="theme">position_value</class>
          </restriction>
           <restriction name="DTSEL_STATE_VALUE">
            <class role="theme">state_value</class>
          </restriction>
          <restriction name="DTSEL_STATE_VALUE_KEY">
            <class role="theme">state_value</class>
            <class role="from-loc">key</class>
          </restriction>
          <restriction name="DTSEL_KEY">
            <class role="theme">key</class>
          </restriction>
          <restriction name="DTSEL_ITEM_APPLICATION">
            <class role="theme">item</class>
            <class role="from-loc">application</class>
          </restriction>
        </restrictions>
      </frame>
```

Figura 5.2 Fragmentos da gramática do sistema

A geração da gramática é um importante passo pois além de restringir as sentenças dos casos de uso, ela também já possibilita boa parte das traduções para a geração do modelo de CSP. Assim, após essa etapas, seremos capazes de definir o Alfabeto de CSP.

Uma dificuldade encontrada foi na manutenção da gramática em CSP. Pois na medida em que os casos de usos eram implementados através do template, constantemente fazia-se necessário reajustar a gramática para que a definição dos casos de usos pudesse fluir de maneira satisfatória.

5.3.3 Criação dos Casos de Uso através de templates

A definição dos casos de uso na visão de componentes envolve o conhecimento prévio dos requisitos da aplicação e das suas definições de arquitetura, tais como ambiente e padrões de projeto. Nessa fase, alguns designers decidem por implementar primeiramente

os casos de uso a partir de uma visão de usuário para um melhor entendimento. Após isso, eles evoluem o modelo para uma visão por componentes [5].

No nosso caso, a implementação dos casos de uso na visão de usuário foi descartada, pois tratamos com um sistema onde sua essência é a comunicação entre hardwares. Ou seja, a visão de componentes é mais nítida, sendo simples identificar quais componentes executam as tarefas do sistema.

A próxima figura apresenta um exemplo do uso do template de caso de uso na visão de componente, e como sua construção foi feita. O exemplo já é escrito em CNL e cobre o domínio da aplicação do metrô (CGP) - o caso de uso escolhido foi o *Abrir Porta.* Nele, os componentes comunicam-se concorrentemente entre si e de forma simultanêa. Os itens dos casos de uso (ação do usuário, estado do sistema, resposta de sistema, e mensagem) foram escritos em CNL e já aderem à gramática.

Feature – 01

Use Cases

UC 01 – Execute the command "open" for all doors of a determined position

Main Flow

From Step: START
To Step: END

Step	Sender	Message	System State	Receiver
UC_01_1M	User	Set the selector key to enabled		Train Command App
UC_01_2M	User	Set the selector key position to right	Selector key is enabled	Train Command App
UC_01_3M	User	Select the "preparation option" from kloan key	Selector key position is right Selector key is enabled	Train Command App
UC_01_4M	Train Command App	Start the open doors action	Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Controller
UC_01_5M	Train Controller	Start the temporization flag with 10 seconds	Kloan key option is "preparation" Selector key position is right Selector key is enabled	Timer
UC_01_6M*	Train Controller	Wait for 10 seconds	Temporization flag is active Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Controller
UC_01_7M	Timer	Temporization flag is inactive	Temporization flag is active Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Controller
UC_01_8M	Train Controller	Check the train speed	Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Command App
UC_01_9M	Train Command App	The train speed is less than 6	Train speed is less then 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Controller
UC_01_10M	Train Controller	Execute the open doors action	Train speed is less then 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Command Line
UC_01_11M	Train Command Line	All doors are opened	Train speed is less the 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Controller
UC_01_12M	Train Controller	All doors are opened	The doors are opened Train speed is less the 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Command App
UC_01_13M	Train Command App	The led of respective operation position is lighted	The doors are opened Train speed is less the 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	User

Alternative Flows

From Step: UC_01_6M
To Step: END

Step	Sender	Message	System State	Receiver
UC_01_1A_1	User	Change the selector key position from right to left	Temporization flag is active Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Command App
UC_01_1A_2	Train Command App	Stop the open doors action	Selector key position is left Temporization flag is active Kloan key option is "preparation" Selector key is enabled	Train Controller
UC_01_1A_3	Train Controller	Reset the temporization flag	Selector key position is left Temporization flag is active Kloan key option is "preparation" Selector key is enabled	Timer
UC_01_1A_4	Train Controller	Choose the "unprepared option" from kloan key	Temporization flag is inactive Selector key position is left Kloan key option is "preparation" Selector key is enabled	Train Command App
UC_01_1A_5	Train Command App	The led of respective operation position is extinguished	Kloan key option is "unprepared" Temporization flag is inactive Selector key position is left Selector key is enabled	User

Exception Flows

From Step: UC_01_8M
To Step: END

Step	Sender	Message	System State	Receiver
UC_01_1E	Train Command App	The train speed is greater than or equals to 6	Train speed is greater than or equals to 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Controller
UC_01_2E	Train Controller	Cancel the open doors action	Train speed is greater than or equals to 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Command Line
UC_01_3E	Train Command Line	All doors are closed	Train speed is greater than or equals to 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Controller
UC_01_4E	Train Controller	Choose the "unprepared option" from kloan key	The doors are closed Train speed is greater than or equals to 6 Temporization flag is inactive Kloan key option is "preparation" Selector key position is right Selector key is enabled	Train Command App
UC_01_5E	Train Command App	The led of respective operation position is extinguished	Kloan key option is "unprepared" The doors are closed Train speed is greater than or equals to 6 Temporization flag is inactive Selector key position is right Selector key is enabled	User

Figura 5.3 Parte do caso de uso Abrir Porta.

Na figura há um fluxo principal, um alternativo e um de exceção. A execução do fluxo principal pode ser desviada a um trajeto de exceção após etapa 7M, quando a Train Controller emite uma mensagem ao componente Train Command App. Aqui, a mensagem seguinte a ser trocada depende do estado atual do sistema. Nesse exemplo, o estado da velocidade do trem (menor que 6 ou não) determina a mensagem seguinte a ser trocada entre os componentes. Note que a etapa 1E do fluxo da exceção está ativada após etapa 7M, quando a circunstância falhar. O campo To Step, no fluxo de exceção, indica que depois que o fluxo da execução ocorrer, a execução do caso de uso termina (END).

Normalmente os casos de uso descrevem as funcionalidades do sistema sem revelar a estrutura interna do sistema. Entretanto, o exemplo proposto quebra esta convenção e é

usado na verdade para detalhar os casos de uso na visão de usuário, que segue a idéia regular do caso de uso; criação de uma relação entre o ator e o sistema.

Na visão de componente é necessário definir o componente que está invocando uma ação e quem está fornecendo o serviço. É um processo da troca da mensagem composto por um remetente, por um receptor e por uma mensagem. O usuário é visto aqui como um componente, e pode emitir ou receber mensagens para outros componentes, respectivamente. Um componente pode também enviar uma mensagem para si mesmo. Estas particularidades permitem a definição de cenários simultâneos, que é uma exigência não-funcional. Assim, componentes podem compartilhar recursos e mensagens de troca, algo que não é possível em modelos de caso de uso comuns.

A definição das chamadas dentro dos casos de uso pode ser tarefa complexa. O caso de uso *Fechar Portas*, por exemplo, contém além do fluxo principal, cinco fluxos alternativos e 1 de exceção. Assim pode ser que os casos de uso definidos induzam o modelo do sistema a deadlocks ou livelocks. Felizmente, estas propriedades podem ser verificadas, usando ferramentas tais como FDR [18], e os deadlocks e livelocks podem ser impedidos.

Analisando um pouco mais o caso de uso gerado a partir do template, nota-se que esse sistema é caracterizado essencialmente em manter um conjunto considerável de informações sobre seu estado. Para verificar isso, basta olhar a coluna *System State*. Essa característica será analisada melhor na seção seguinte. Até esse ponto, a estratégia adotada cobre bem as características do sistema em análise.

5.3.4 Tradução para CSP

A etapa de tradução se iniciou pela geração dos *datatypes* e dos *channels* de CSP. Estes elementos de CSP relacionam-se com a gramática de CNL a fim de se obter o alfabeto de CSP.

Conforme mencionamos no capítulo anterior, os verbos, os termos, e os modificadores são traduzidos em possíveis *eventos* e *datatypes* de CSP que são usados mais tarde gerar o modelo de CSP.

A tradução se deu de forma automática, de modo que cada classe da ontologia tornou-se um datatype de CSP, mapeamos os cases frame da CNL em *channel* de CSP, considerando também as restrições que geravam novos datatypes. Assim como no capítulo anterior, a figura a seguir também ilustra alguns dos elementos de CSP gerado.

```
datatype Item =
 VARIABLEITEM.VariableItem
 | FIELD.Field
 | KEY.Key
 | SIGNALLING.Signalling

datatype VariableItem =
 TRAIN_SPEED
 | SELECTOR_KEY_POSITION

datatype Field =
 TEMPORIZATION_FLAG_FIELD
 | COMMAND_FIELD

datatype Key =
 CLOSING_BUTTON_KEY
 | SELECTOR_KEY
 | KLOAN_KEY

datatype Signalling =
  DISPLAY
  | LED

datatype Modifier =
 ALL
 | OPTION
 | LESS_THAN.Int
 | GREATER_THAN_OR_EQUALS_TO.Int
 | WITH_N_SECONDS.Int
 | INTEGER.Int
 | END_OF.Field
 | WITHOUT.Item

...

channel set : DTSet
datatype DTSet =
 DTSET_FIELDVALUE_FIELD.(FieldValue, Set(Modifier)).(Field, Set(Modifier))
 | DTSET_STATEVALUE_ITEM.(StateValue, Set(Modifier)).(Item, Set(Modifier))
 | DTSET_STATEVALUE_KEY.(StateValue, Set(Modifier)).(Key, Set(Modifier))
 | DTSET_POSITIONVALUE_VARIABLEITEM.(StateValue, Set(Modifier)).(VariableItem, Set(Modifier))

channel change : DTChange
datatype DTChange =
 DTCHA_VARIABLEITEM.(VariableItem, Set(Modifier))
 | DTCHA_POSITIONVALUE_VARIABLEITEM_POSITIONVALUE.(PositionValue, Set(Modifier)).
 (VariableItem, Set(Modifier)).(PositionValue, Set(Modifier))
 | DTCHA_FIELD.(Field, Set(Modifier))
 | DTCHA_APPLICATION.(Application, Set(Modifier))
 | DTCHA_ITEM.(Item, Set(Modifier))

...
```

Figura 5.4 Parte do alfabeto de CSP gerado

De posse do alfabeto, iniciamos a tradução de cada sentença dos templates de casos de uso em eventos de CSP.

Uma dificuldade encontrada para realizar a tradução para CSP foi devido a não utilização da ferramenta responsável por validar as sentenças em CNL dos casos de uso. Assim esse processo se apresentou um pouco trabalhoso, pois foi feito por métodos de tentativas até se obter o resultado esperado.

Por fim, temos a geração do modelo de componentes em CSP. Este modelo foi gerado a partir dos casos de usos especificados nos templates.

Cada processo foi definido como uma seqüência das mensagens comunicadas para outro componente. A próxima figura está ilustrando somente o processo de USER_P, que agrupa todas as mensagens enviadas e recebidas pelo componente USER e os outros componentes apresentados no caso de uso. Os processos restantes, relacionados com as mensagens dos outros componentes do caso de uso, são estruturados na mesma maneira que o USER_P. Observe a troca mensagens do processo USER_P com o processo do componente TRAIN_COMMAND_APP previamente apresentado.

Para cada etapa do caso de uso há dois eventos de CSP. Cada evento de CSP mostra os componentes remetentes e receptores envolvidos em sua troca de mensagens. Ainda na figura, *Set the selector key to enabled* tem sua notação em CSP no processo USER_UC_01 definida como:

setComp.USER.TRAIN_COMMAND_APP.DTSET_STATEVALUE_KEY.(ENABLED_STATE_VALUE, {}). (SELECTOR_KEY, {}).

Entre o nome do canal (setComp) e o nome da restrição (*dtset_statevalue_key*) vêm os componentes: remetente (USER) e receptor (TRAIN_COMMAND_APP).

```
 include "CSP_HEADER_COMPONENT.csp"

   SubSystem1_events = union(User_Channels, Train_Command_App_Channels)
   SubSystem1 = USER_P [ User_Channels || Train_Command_App_Channels] TRAIN_COMMAND_APP_P

   SubSystem2_events = union(SubSystem1_events, Train_Controller_Channels)
   SubSystem2 = SubSystem1 [SubSystem1_events || Train_Controller_Channels] TRAIN_CONTROLLER_P

...

USER_P = USER_UC_01 [] USER_UC_02

   -- Scenario Case: Execute the command "open" for all doors of a determined position

   USER_UC_01 =
       -- Message: Set the selector key to enabled.
       setComp.USER.TRAIN_COMMAND_APP.DTSET_STATEVALUE_KEY.(ENABLED_STATE_VALUE, {}).(SELECTOR_KEY, {}) ->
       -- Message: Set the selector key position to right
       setComp.USER.TRAIN_COMMAND_APP.DTSET_POSITIONVALUE_VARIABLEITEM.(RIGHT_POSITION_VALUE, {}).
                                    (SELECTOR_KEY_POSITION, {}) ->
       -- Message: Select the "preparation option" from kloan key
       selectComp.USER.TRAIN_COMMAND_APP.DTSEL_STATEVALUE_KEY.(PREPARATION_STATE_VALUE, {OPTION}).
                                        (KLOAN_KEY, {}) ->
       (USER_UC_01_13M  [] USER_UC_01_5E)

   USER_UC_01_13M =
       -- Message:  The led of respective operation position is lighted.
       lightComp.TRAIN_COMMAND_APP.USER.DTLIG_SIGNALLING_ITEM.(LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
       USER_P [] USER_UC_01_1A_1

   USER_UC_01_1A_1 =
       -- Message: Change the selector key position from right to left.
       changeComp.USER.TRAIN_COMMAND_APP.DTCHA_POSITIONVALUE_VARIABLEITEM_POSITIONVALUE.
                 (RIGHT_POSITION_VALUE, {}).(SELECTOR_KEY_POSITION, {}).(LEFT_POSITION_VALUE, {}) ->
       -- Message: The led of respective operation position is extinguished.
       extinguishComp.TRAIN_COMMAND_APP.USER.DTEXT_SIGNALLING_ITEM.
                     (LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
       USER_P

   USER_UC_01_5E =
       -- Message: The led of respective operation position is extinguished.
       extinguishComp.TRAIN_COMMAND_APP.USER.DTEXT_SIGNALLING_ITEM.
                     (LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
       USER_P
...
```

5.5 Parte do processo do componente USER_P em CSP

Durante a geração do modelo, o operador de escolha externa de CSP é usado para permitir a execução do caso de uso escolhido pelo usuário. Neste caso, o usuário claramente tem a escolha de execução sobre um determinado caso de uso selecionado.

Analisando a abordagem adotada, nota-se que em se tratando do seu modelo e forma de tradução dos casos de uso ela possui uma característica comprometedora para nossa aplicação. A execução dos casos de uso gerados são essencialmente seqüenciais, apesar dos componentes do sistema serem executados em paralelo. Na verdade, apenas os componentes do caso de uso ficam em paralelo.

Essa característica é evidenciada quando verificamos que cada caso de uso deve terminar para que novos casos de uso possam começar (Figura 5.6).

```
USER_P =
    USER_UC_01
    []
    USER_UC_02
    [] ...
```

5.6 Definição do processo USER_P em CSP

Dessa forma, como modelar um sistema em que a execução de um caso de uso pode ser interrompida por outro? Observe que no CGP, o caso de uso *FecharPorta* tem prioridade sobre o *AbrirPorta*. Ou seja, o *FecharPorta* pode ser executado durante o andamento do caso de uso *AbrirPorta*. Sendo assim, para nossa aplicação essa abordagem se mostra incompleta, devendo assim ser adaptada para que assim possamos obter o modelo formal completo em CSP para o sistema.

Outro problema consequente a essa análise é que se um caso de uso não tiver um ponto de término, deve chamar outro caso de uso de modo que o modelo possa conter laços recursivos entre casos de uso. Assim, além de também incapacitar o usuário de executar outros casos de uso, isso também faz com que o sistema apresente um estado de livelock.

Uma possível solução a ser oferecida seria propor a execução desses casos de usos num paralelismo sincronizado em eventos (eventos responsáveis pela interrupção) como é mostrado abaixo:

```
USER_P =  USER_UC_01 [{|User_Interrupt_Channels|}] USER_UC_02 ...
```

5.7 Redefinição do processo USER_P em CSP

Para isso, seria necessário replicar os canais que fossem usados por mais de um caso de uso. Por exemplo, o canal *changeComp* deveria ser replicado e identificado com o caso de uso que o invocou para que ele só possa sincronizar com o respectivo caso de uso do outro componente. Evitando assim, que processos diferentes disputem pela sincronização do canal.

```
-- Message: Change the selector key position from right to left.
changeComp.USER.TRAIN_COMMAND_APP.DTCHA_POSITIONVALUE_VARIABLEITEM_POSITIONVALUE.
(RIGHT_POSITION_VALUE, {}).(SELECTOR_KEY_POSITION, {}).(LEFT_POSITION_VALUE, {})

changeComp.UC_01.USER.TRAIN_COMMAND_APP.DTCHA_POSITIONVALUE_VARIABLEITEM_POSITIONVALUE.
(RIGHT_POSITION_VALUE, {}).(SELECTOR_KEY_POSITION, {}).(LEFT_POSITION_VALUE, {}) ->

changeComp.UC_02.USER.TRAIN_COMMAND_APP.DTCHA_POSITIONVALUE_VARIABLEITEM_POSITIONVALUE.
(RIGHT_POSITION_VALUE, {}).(SELECTOR_KEY_POSITION, {}).(LEFT_POSITION_VALUE, {}) ->
```

Figura 5.8 Redefinição do evento de USER_P em CSP

Outra adaptação seria a inclusão do processo responsável pela interrupção de um caso de uso juntamente com os outros sub-processo situados no external choice de cada sub-processo do caso de uso a ser interrompido. Por exemplo, se o caso de uso *Fechar Portas* pode interromper o caso de uso *AbrirPorta*, então os sub-processos do Processo USER_P que contém uma "bifurcarção" de fluxo – representada pelos external choice – deve conter mais um fluxo possível inserido juntamente com os outros.

```
USER_P = USER_UC_01 [{|cancelInterrupt.UC_02.UC_01|}] USER_UC_02

   -- Scenario Case: Execute the command "open" for all doors of a determined position

   USER_UC_01 =
       -- Message: Set the selector key to enabled.
       setComp.USER.TRAIN_COMMAND_APP.DTSET_STATEVALUE_KEY.(ENABLED_STATE_VALUE, {}).
                                       (SELECTOR_KEY, {}) ->
       -- Message: Set the selector key position to right
       setComp.USER.TRAIN_COMMAND_APP.DTSET_POSITIONVALUE_VARIABLEITEM.(RIGHT_POSITION_VALUE, {}).
                                       (SELECTOR_KEY_POSITION, {}) ->
       -- Message: Select the "preparation option" from kloan key
       selectComp.USER.TRAIN_COMMAND_APP.DTSEL_STATEVALUE_KEY.(PREPARATION_STATE_VALUE, {OPTION}).
                                          (KLOAN_KEY, {}) ->
       (USER_UC_01_13M  [] USER_UC_01_5E [] USER_UC_01_CANCEL_INTERRUPT )

   USER_UC_01_13M =
       -- Message:  The led of respective operation position is lighted.
       lightComp.TRAIN_COMMAND_APP.USER.DTLIG_SIGNALLING_ITEM.(LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
       (USER_P [] USER_UC_01_1A_1 [] USER_UC_01_CANCEL_INTERRUPT )

   USER_UC_01_1A_1 =
       -- Message: Change the selector key position from right to left.
       changeComp.USER.TRAIN_COMMAND_APP.DTCHA_POSITIONVALUE_VARIABLEITEM_POSITIONVALUE.
                  (RIGHT_POSITION_VALUE, {}).(SELECTOR_KEY_POSITION, {}).(LEFT_POSITION_VALUE, {}) ->
       -- Message: The led of respective operation position is extinguished.
       extinguishComp.TRAIN_COMMAND_APP.USER.DTEXT_SIGNALLING_ITEM.
                     (LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
       USER_P

   USER_UC_01_5E =
       -- Message: The led of respective operation position is extinguished.
       extinguishComp.TRAIN_COMMAND_APP.USER.DTEXT_SIGNALLING_ITEM.
                     (LED, {OF_RESPECTIVE_OPERATION_POSITION}) ->
       USER_P

   USER_UC_01_CANCEL_INTERRUPT =
       -- Message: Interrupt the UC_01
       cancelInterrupt.UC_02.UC_01
       -- As mensagens necessárias para interromper o caso de uso
       ....
```

Figura 5.9 Ajuste sobre o processo de USER_P em CSP

Outra mudança prevista é a alteração das referências das etapas finais dos sub-processos do caso de uso. Para cada etapa que referenciava o processo USER_P, com a adição do paralelismo essas etapas devem referenciar agora o processo que representa o inicio do caso de uso ao qual elas estão inseridas.

Estas e outras questões devem ser consideradas para que o sistema não apresente situações de não-determinismo ou livelock e serão analisadas, com mais propriedade, em trabalhos futuros.

É importante observar também que os processos do sistema não têm a representação de seus estados (parâmetros), conseqüentemente a execução de casos de uso, um após o outro, é independente; a ordem entre a execução dos casos de uso não varia o comportamento do sistema. Isto não é o que deveria acontecer na realidade, porque as variáveis do sistema podem ser afetadas depois que um caso de uso é executado. Dessa forma, os estados são usados apenas como documentação no modelo. Esse é mais um problema encontrado em nossa solução. A adição de uma nova definição capaz de manter as informações sobre o estado do sistema também é um fator pendente.

Porém, já conseguimos identificar um caminho para a solução desse problema. Analisando o template de caso de uso, verificamos que os componentes (termos de CNL) identificados como *Receiver* no caso de uso são possíveis processos candidatos a contér as variaveis de estado.

Isso pode ser observado, considerando que se algum componente envia uma mensagem para outro e após essa etapa o estado do sistema muda, então essa ação resultou numa mudança sobre o comportamento do destinátário (devido à recepção da mensagem). Assim, esse componente é visto como o responsável por manter esse comportamento.

Dessa forma, é necessário adicionar métodos e novas técnicas de transformações sobre a ferramenta de validação e conversão de casos de usos para que essas características sejam acrescentadas. Por fim uma solução no nível de especificação em CSP também deve ser inserida para termos um modelo formal que supra essas necessidades. O advento dessas características talvez traga, por exemplo, a necessidade de se incluir estruturas de controle na especificação dos processos em CSP – *if else, let with* - para manipular as variáveis de estado e definir o fluxo dos procesos. Algo que até o momento não existe.

5.4 VERIFICAÇÃO DAS PROPRIEDADES DO SISTEMA

Nesta seção serão mostrados os resultados obtidos da especificação em CSP extraída do sistema. Deve-se lembrar que tal avaliação é apenas experimental, pois o comportamento do processo gerado não correspondeu por completo a especificação dos requisitos, haja vista que necessidades de adaptações e extensões sobre a estratégia foram levantadas.

Porém, graças à aplicação da estratégia de conversão, foi possível notar que a especificação em CSP garante a propriedade de alcaçabilidade do sistema. Pois as técnicas de mapaemento que utilizam referências entre fluxo principal, fluxos alternativos e de exceção dos sitema para ligar as etapas do processo, evitam que o sistema apresente fluxos inátigíveis.

Além disso, as propriedades de ausência de deadlock, livelock e não-determinismo do modelo gerado foram vericadas através das ferramentas ProBe e FDR. Obteve-se um resultado positivo, ou seja, o modelo não apresentou nenhum desses problemas.

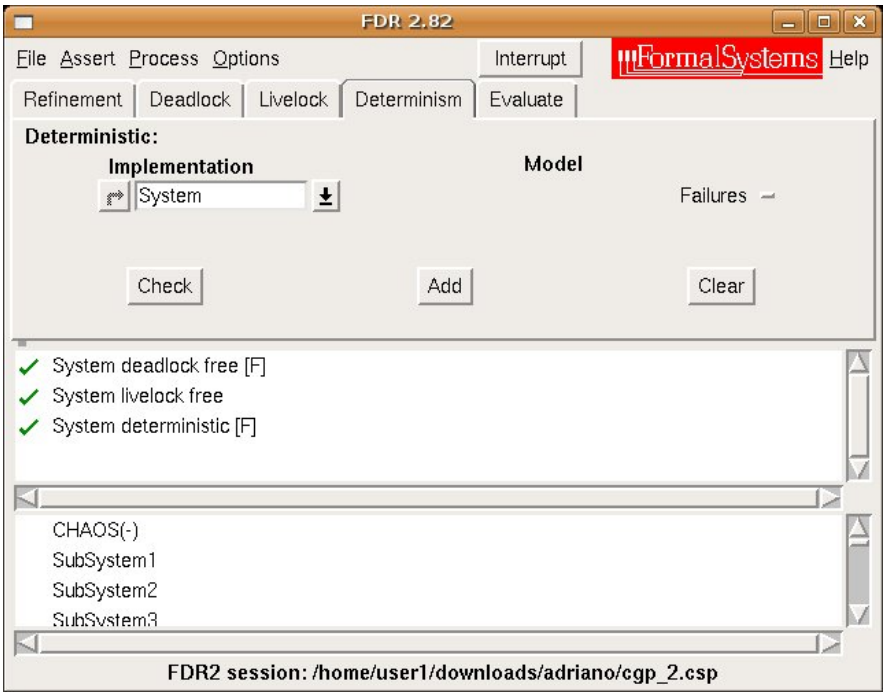

Figura 5.10 Resultados do FDR

5.4.1 Comparação de Resultados

Este tópico apresenta a verificação das propriedades do sistema realizadas em [36], com objetivo de comparar os resultados obtidos em ambos os trabalhos. Como já mencionamos, o trabalho em [36] gera a especificação em CSP a partir de códigos na linguagem C.

Alguns empecilhos impediram uma comparação de resultados a nível semântico. Um deles foi o fato das representações das comunicações entre processos (canais e eventos) serem diferentes em cada trabalho. Além disso, os processos que poderiam ser equivalentes semanticamente possuiam comportamentos diferentes, pois os artefatos disponibilizados estavam carentes de detalhes que ocasionassem uma maior paridade entre as especificações (implementação e documentos de requisitos).

Assim, a comparação dos resultados, se limitou à verificação isolada das propriedades de ambos os trabalhos, visto que encontrar uma relação de refinamento não foi possível, a primeira vista, devido aos alfabetos de ambas as extrações possuírem representações distintas. Trabalhos futuros tentarão compatibilizar as versões das especificações para que refinamentos possam ser avaliados.

A próxima figura mostra os resultados das propriedades: deadlock, livelock e determinismo do trabalho [36].

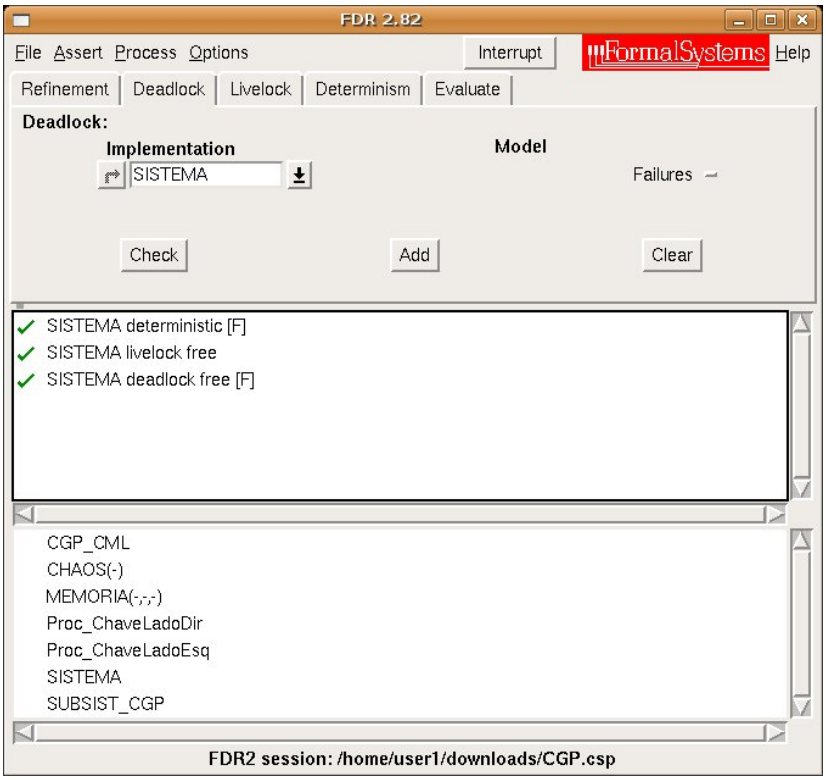

Figura 5.11 Resultados do trabalho relacionado

5.5 CONSIDERAÇÕES

Nem todas as ferramentas apresentadas nos capítulos anteriores foram utilizadas para aplicação da abordagem de geração do modelo em CSP, pois algumas delas não estavam disponíveis. Assim, boa parte dos passos da implementação foram feitos de forma manual, resultando num esforço maior e às vezes impedindo-nos de fazer uma análise mais detalhada. Em especial, a ferramenta de validação de sentenças – o *Use Case Validator* [5] – é muito importante para se obter uma boa produtividade durante o processo de tradução dos casos de uso.

Como foi visto no capítulo anterior, há casos em que a execução de um fluxo alternativo ou de exceção é habilitada por uma combinação dos fatores (ação do usuário e estado do sistema), podendo gerar situações em que o operador de escolha interna de CSP possa ser utilizado. Assim, para a aplicação em estudo, estes casos devem ser tratados com cuidado, pois o refinamento da abordagem não cobre casos de não-determinismo. Nesse sistema, na prática, a garantia dessa propriedade deve ser considerada como algo essencial. Assim, faz-se necessário que nas etapas posteriores da implementação do sistema, sejam

realizados refinamentos de mais alto nível para contornar essa situação. Esse processo já foi iniciado em [36].

Quanto à consistência e completude das adaptações propostas, só temos uma intuição de que elas nos dão uma especificação CSP válida como resultado final, mas como não provamos que isso é verdade e nossa intuição advém dos experimentos práticos que fizemos, resta explorar mais a fundo este aspecto importante sobre um novo conjunto de regras. Assim, resolvemos deixar este tópico para pesquisas futuras.

Os artefatos gerados por este trabalho estão disponíveis em [38].

5.6 RESULTADOS OBTIDOS

Mapear as sentenças de CNL em eventos de CSP constitui-se em etapas importantes para criação do modelo de CSP, pois define o alfabeto de CSP. Felizmente obtivemos sucesso nessa tarefa para o sistema em análise.

Analisando o caso de uso gerado na visão de componente, é fácil verificar que ele é uma maneira textual de especificar os diagramas de seqüência de UML [29]. O remetente e o receptor das colunas definem os atores envolvidos na comunicação e a mensagem é o próprio pedido do serviço. A ordem da mensagem determina o arranjo do diagrama de seqüência. Além de um componente que emite um pedido a um outro componente, o componente do receptor pode responder este pedido através de uma outra expedição da mensagem. Neste caso, o receptor age como o remetente, e vice-versa. Diagramas de seqüência de UML são usados extensamente por equipes de desenvolvimento de software e sua geração automática representa um benefício importante.

A partir da estratégia de extração, conseguimos gerar uma especificação formal em CSP do escopo abordado no sistema do Metrô.

Verificamos que uma etapa da estratégia usada para gerar os modelos na visão de componente não se mostrou, a princípio, satisfatória para o sistema do Metrô. Pois, a estratégia utilizada não modela bem sistemas caracterizados essencialmente em manter um conjunto considerável de informações sobre seu estado. Dessa forma, não foi possível extrair um modelo totalmente equivalente ao sistema em CSP. Nossa intenção é que em trabalhos futuros possamos implementar as adaptações à abordagem utilizada de forma a suprir essas carências.

Após a aplicação e análise da estratégia proposta, foi possível identificar os problemas e carências que impediram a validação completa da abordagem no sistema em estudo (adaptações foram propostas para trabalhos futuros). Dentre elas destacam-se:

- Os processos do sistema não têm uma representação de seus estados em CSP;
- Os fluxos de execução dos casos de uso gerados são essencialmente seqüenciais.

Mediante o modelo formal do sistema, conseguimos realizar validações sobre sistema. Assim foi possível analisar as seguintes propriedades:

- Deadlock
- Livelock
- Não-Determinismo
- Alcançabilidade

Essas propriedades também foram comparadas com um outro trabalho [36] que também que faz parte desse ciclo de pesquisa maior sobre o qual está inserido o sistema do Metrô.

CAPÍTULO 6

CONCLUSÃO

Há diversos benefícios associados ao uso dessa estratégia de automação utilizada em nosso trabalho. Formas de representar casos de usos, entendimento do contexto do sistema, validação de propriedades exigidas são alguns exemplos.

Aprender CNL pode ser uma tarefa complexa, uma vez que os termos e as expressões específicos do domínio de CNL podem ser constantemente alterados fazendo que um conjunto de elementos seja considerado cada vez. Assim, recomenda-se que o designer não desperdice muito tempo em tentar configurar uma maneira de escrever as sentenças de forma aderente a CNL. Ele deve focar a atenção no significado e na complexidade dos casos de uso.

O uso das linguagens semiformais, tais como UML, aumentou. A simplicidade e flexibilidade destas linguagens permitem sua larga compreensão e uso. Além disso, ferramentas como Racional Rose [31] permitem uma rápida implementação. UML 2.0, que inclui UML-RT e OCL, permitem uma especificação precisa dos sistemas. O desenvolvimento de ferramentas que suportem linguagens formais é uma boa alternativa para fazer seu uso mais praticável. Em especial, ferramentas unificadas que fossem capazes de encombrir todas as etapas da estratégia de conversão, por exemplo, seriam de grande utilidade.

O uso de métodos formais traz freqüentemente uma apreensação às equipes de desenvolvimento não familiarizadas com a notação abstrata da especificação tal como a álgebra processo de CSP. Entretanto, os benefícios da especificação formal dos sistemas, especialmente a respeito dos sistemas críticos, são inúmeros [32, 33]. Conseqüentemente, essa abordagem parace explorar meios alternativos para permitir o uso da especificação formal em projetos reais, e não apenas em projetos acadêmicos, escondendo as formalidadas tanto quanto possível.

Estas experiências envolveram a adoção dos templates de caso de uso, da CNL proposta, e sua tradução a um modelo formal de CSP. Este procedimento serviu para avaliar a estrutura do template de caso de uso e seu modelo gerado. Análises adicionais forneceram bastante informação para melhorar a estrutura do modelo, as bases de

conhecimento de CNL em trabalhos futuros. Além disso, validações sobre as propriedades do sistema foram apresentadas e comparadas com outra área de pesquisa do trabalho, sendo possível assim, detectar não-conformidades.

Apesar do modelo de CSP gerado ser incompleto a ponto de não ser totalmente equivalente ao sistema em estudo, as limitações e adaptações necessárias à abordagem foram identificadas após a análise, tornando possível uma continuidade sobre o trabalho e também fornecendo um argumento avaliativo que pode ser usado como fator de decisão por pessoas que trabalham nessa área.

6.1 TRABALHOS RELACIONADOS

Na literatura, poucos trabalhos têm abordado a integração de métodos formais com a especificação de sistemas através de documentos de requisitos. A seguir, apresentamos alguns dos trabalhos relacionados com o nosso:

6.1.1 Geração Automática de Requisitos em Inglês através de modelos em CSP

Em [35], o trabalho se propõe em modelar os documentos com especificações formais e escrevê-los com uma linguagem natural controlada, para que se evite a introdução de ambigüidade e sentenças não-uniformes. È uma estratégia efetiva para garantir que nenhuma incerteza a respeito dos seus conteúdos esteja presente. O objetivo principal do trabalho é apresentar uma ferramenta que ajude na manutenção de documentos de requisitos, atualizados através de especificações formais em CSP correspondentes aos casos de teste. Portanto, esse trabalho segue um caminho de implementação com sentido oposto ao nosso (modelo formal para documentos de requisitos).

6.1.2 Uma Abordagem para Extração de Especificação em CSP a partir da Linguagem C

O trabalho em [36] tem como objetivo gerar especificações formas a partir de código em C. Como foi dito anteriormente, esse trabalho em conjunto com o nosso, faz parte de um projeto maior que visa avaliar as caracteristicas e propriedades do sistema abordado aqui sobre dois caminhos.

O trabalho faz a combinação entre as vantagens oferecidas pela linguagem C juntamente com possibilidade de visualização da aderência à modelagem em CSP. Pois assim será possível verificar propriedades intrínsecas desejáveis aos sistemas (durante e após a implementação), levando a um ganho significativo de produtividade e conseqüente aumento de confiabilidade nos mesmos.

A evolução do trabalho é mostrada, assim como o nosso, através do emprego de técnicas sobre o sistema do metrô para uma documentação das mesmas em seguida, de forma que possam ser incorporadas durante o processo de desenvolvimento de sistemas. Isso fará uma ligação desejável e sutil entre o formalismo das notações em CSP junto à linguagem de programação C.

Ambos os trabalhos visam pesquisas futuras a fim de manter a continuidade do projeto.

6.1.3 Formal Specification Generation from Requirement Documents

Por fim, vale ressaltar [5] que fundamentou todo o processo de execução de nosso trabalho. O trabalho citado apresenta toda essa estratégia de extração de modelos formais a partir de métodos formais. Nele, a eficácia da estratégia é apresentada através de um estudo de caso sobre um sistema para celulares. Essa estratégia utiliza os resultados sobre CNL apresentados em [26], assim similarmente ao nosso caso, pesquisas de outros trabalhos vêm gerando oportunidades para a execução de novos trabalhos na área em domínios de aplicações diferentes.

6.2 TRABALHOS FUTUROS

Uma melhoria óbvia para o trabalho atual é poder capturar o comportamento dos componentes através de seus estados. A etapa de conversão de casos de usos na visão de componentes não suporta esta característica. Atualmente, os estados dos processos são apenas documentados no template de caso de uso, mas não são considerados na conversão. Esta é uma limitação do trabalho atual que será explorada em versões futuras.

Assim, pretende-se estender a ferramenta de conversão de modelo de componentes para que o modelo em CSP gerado seja capaz de considerar os estados dos processos que na solução estariam representados através de parâmetros. Para isso, pretendemos adaptar o analisador de template de casos de usos para que ele possar identificar os estados e seus

componentes descritos no template. Também devemos estudar uma solução de especificação em CSP que traduza essas caracteristicas em código para que ela possa ser inserida como uma nova atividade do parser de conversão do modelo.

Um outro ponto refere-se ao tratamento do paralelismo entre casos de usos. Como foi visto, a abordagem em estudo considera a execução dos casos de uso de forma sequencial. Assim, propomos uma solução com paralelismo para que um caso de uso possa ser iniciado como um processo do sistema mesmo quando outro caso de uso já está em execução. Assim, necessitamos validar a completude e consistência dessa solução para que ela possa ser adicionada ao processo de geração de especificação em CSP de forma segura.

Também gostaríamos de explorar uma relação de refinamento entre o modelo formal de CSP gerado nesse trabalho e suas versões futuras e o modelo gerado a partir do código fonte proposto em [36]. Não houve tempo hábil para explorar e analisar melhor as relações semânticas entre ambos trabalhos.

A semântica que se tentou avaliar era uma conseqüência do não tratamento de estados que ocorrem na abordagem aqui aplicada. Deveríamos verificar que um sub-processo da especificação extraída por esse trabalho - que contivesse um desvio de fluxo representado pelo operador *external choice* - seria refinado em modelo de traces por um sub-processo de mesmo valor semântico no trabalho [36] – representado por um if, else. A idéia era mostrar que esses sub-processos deveriam ser equivalentes semanticamente no nível de traces. Mas o fato da nossa especificação não considerar o tratamento de estado, mostraria que essa equivalência não iria ocorrer nesse contexto, pois nosso sub-processo não refinaria o sub-processo do trabalho [36]. Ou seja, só haveria refinamento em um sentido.

Nossa intenção é discutir e analisar esse contexto em trabalhos futuros, dessa forma, poderiamos confrontar os resultados e verificar o comportamento e propriedades do sistema para ambas situações, analisando os pontos relevantes neste contexto e identificando possíveis não-conformidades.

Como trabalho futuro, temos também a idéia de implementar um framework de desenvolvimento que acoplasse todas as ferramentas utilizadas na abordagem bem como as novas que surgissem. Uma carência gritante nesse meio é a falta de um editor de código para a linguaguem CSP. A ídeia seria criar um ambiente de desenvolvimento, no qual o desenvolvedor fosse capaz implementar todo processo de geração de especificação em CSP a partir de requsitos. Desde a gramática de CNL, à análise do código CSP gerado.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] Siddhartha R. Dalal, Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Bruce M. Horowitz. *Model-based testing in practice*. In International Conference on Software Engineering, pages 285–294, 1999.

[3] M. Thomas. *The industrial use of formal methods. Microprocessors and Microsystems*, 17:31{36, 1993.

[4] Bernhard Schätz, Andreas Fleischmann, Eva Geisberger, and Markus Pister. *Model-based requirements engineering with autoraid.* In GI Jahrestagung (2),pages 511–515, 2005.

[5] Cabral, Gustavo da Fonseca Limaverde. *Formal Specification Generation from Requerement Documents*, Tese de Mestrado, Universidade Federal de Pernambuco(UFPE), set. 2006.

[6] Ian Sommerville and Pete Sawyer. *Requirements Engineering: A good practice guide*. John Wiley and Sons Ltd., 1997.

[7] S. S. Ali. *A logical language for natural language processing*. Proceedings of the 10th Biennial Canadian Artificial Intelligence Conference Banff, Alberta, Canada, 1994. IEEE Computer Society.

[8] *Z Notation*. Wikipedia
Disponível em: http://en.wikipedia.org/wiki/Z_notation

[9] *CSP*. Wikipedia

Disponível em: < http://en.wikipedia.org/wiki/Communicating_sequential_processes >

[10] *Circus*.
Disponível em: http://www.cs.york.ac.uk/circus/

[11] Sidney de Carvalho Nogueira. *Geração automática de casos de teste csp dirigida por propósitos*. Tese de Mestrado, Universidade Federal de Pernambuco(UFPE), aug 2006.

[12] Emanuela Cartaxo. *Test case generation by means of UML sequence diagrams and label transition system for mobile phone applications*. Tese de Mestrado, Universidade Federal de Campina Grande (UFCG), aug 2006.

[13] Patrícia Muniz Ferreira - *Geração Automática de Diagramas UML-RT a partir de Especificações CSP*. Tese de Mestrado. Universidade Federal de Pernambuco(UFPE), set 2006.

[14] Patricia Ferreira, Augusto Sampaio, and AlexandreMota. *Viewing CSP specifications with UML-RT diagrams. In Brazilian Symposium on FormalMethods (SBMF)*, Natal, Brasil, sep 2006.

[15] Angela Freitas and Ana Cavalcanti. *Automatic translation from Circus to java*. FM, pages 115–130, 2006.

[16] Peter H. Welch, Jo R. Aldous, and Jon Foster. *Csp networking for java (jcsp.net)*. ICCS ’02: Proceedings of the International Conference on Computational Science-Part II, pages 695–708, London, UK, 2002. Springer-Verlag.

[17] *Especificação Técnica dos Requisitos de Software para a Controladora Geral de Portas*. Projeto Santiago Linha 2, Chile. Revisão E, 2005

[18] P. Gardiner. *Failures-Divergence Refinement*, FDR2 User Manual and Tutorial. Formal Systems Ltd., 1997.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River,NJ, USA, 1985.

[20] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. PrenticeHall PTR, Upper Saddle River, NJ, USA, 1997.

[22] JÚNIOR, J. O. C. L.; *Verificando Refatorações Automaticamente*. Trabalho de Graduação. Universidade Federal de Pernambuco (UFPE) 2006

[23] NASCIMENTO, C. M. P. *Verificação de modelos para programas em um subconjunto de JCSP*. 2006

[24] M. Goldsmith. *FDR: User Manual and Tutorial*, versão 2.77. Formal Systems (Europe) Ltd, August 2001.

[25] *PROBE Users Manual.*
Disponível em: http://www.fsel.com/software.html.

[26] Daniel Almeida Leitão. Nlforspec**:** *Uma ferramenta para geração de especificações formais a partir de casos de teste em linguagem natural.* Tese de Mestrado, Universidade Federal de Pernambuco (UFPE), aug 2006.

[27] C.J. Fillmore. *Frame semantics and the nature of language*. Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech, 280, 1976.

[28] Erik T. Ray and Christopher R.Maden**.** *Learning XML*. O'Reilly& Associates, Inc., Sebastopol, CA, USA, 2001.

[29] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[30] Montenegro, Pedro. *Model Checkers: Uma análise de ferramentas para a linguagem de programação C*. Trabalho de Graduação. Universidade Federal de Pernambuco, Agosto, 2007

[31] Terry Quatrani. *Visual modeling with Rational Rose and UML*. Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[32] Zarrin Langari and Anne Banks Pidduck. *Quality, cleanroom and formal methods*. In 3-WoSQ: Proceedings of the third workshop on Software quality, pages 1–5, New York, NY, USA, 2005. ACM Press.

[33] Jonathan P. Bowen and Michael G. Hinchey. *Seven more myths of formal methods*. *IEEE Softw.*, 12(4):34–41, 1995.

[34] *Context Programmers Editor*.
Disponível em: http://www.context.cx/

[35] Peres, Glaucia Boudoux. *Automatic English Requirements Generation from Csp Models.* Trabalho de Graduação. Universidade Federal de Pernambuco. Abril, 2007.

[36] Millano, Farley. *Uma Abordagem para Extração de Especificação em CSP a partir da Linguagem C*. Trabalho de Graduação. Universidade Federal de Pernambuco. Agosto, 2007.

[37] *Model Checking*. Wikipedia
Disponível em: http://en.wikipedia.org/wiki/Model_checking

[38] Gomes, Adriano. *Artefatos do Projeto CGP- Metrô*.
Disponível em: http://www.cin.ufpe.br/~ajog/tg

ASSINATURAS

Este Trabalho de Graduação é resultado dos esforços do aluno Adriano José Oliveira Gomes, sob a orientação do professor Alexandre Cabral Mota, sob o título sob o título inicial de "*Analisando uma abordagem para extração de uma modelagem em csp a partir de especificações em linguagem natural"*, posteriormente modificado para "*Verificando a aplicabilidade de uma abordagem de geração de especificação formal a partir de documentos de requisitos"*. Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

_____

Adriano José Oliveira Gomes

_____

Alexandre Cabral Mota