ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

MAPEANDO ESTRUTURAS LSC EM REDES DE PETRI COLORIDAS

Trabalho de Conclusão de Curso

Engenharia da Computação

Nome do Aluno: Cleyton Mário de Oliveira Rodrigues
Orientador: Prof. Ricardo Massa Ferreira Lima

Recife, maio de 2006

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de Sistemas Computacionais

MAPEANDO ESTRUTURAS LSC EM REDES DE PETRI COLORIDAS

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Nome do Aluno: Cleyton Mário de Oliveira Rodrigues
Orientador: Prof. Ricardo Massa Ferreira Lima

Recife, maio de 2006

UNIVERSIDADE
DE PERNAMBUCO

Cleyton Mário de Oliveira Rodrigues

MAPEANDO ESTRUTURAS LSC EM REDES DE PETRI COLORIDAS

Resumo

Os sistemas embarcados ou sistemas embutidos estão presentes cada vez mais no cotidiano das pessoas. Devido ao baixo custo tecnológico de desenvolvimento, nota-se uma clara tendência de crescimento na produção dessas aplicações embarcadas.

Contudo, o projeto desse tipo de sistema depara-se com vários problemas, uma vez que é necessária definir toda a arquitetura de software e hardware. Além disso, os projetos destes sistemas são extremamente caros.

Ao desenvolver um sistema, é importante estar apto a testar e simular o modelo. Ao modelar e testar um sistema desde as fases iniciais de sua implementação, é possível identificar certas exigências, como ausência de estados onde o sistema encontre-se travado (*deadlock*) e detectar erros de especificação de projeto, bem como erros de implementação, prevenindo problemas em fases posteriores do projeto.

Diversas abordagens foram criadas para modelar sistemas de tempo real, onde requisitos funcionais e não funcionais precisam ser garantidos. Alguns exemplos são o *Message Sequence Chart* (MSC)[1] e o *Unified Modelling Language*(UML). Contudo, eles apresentam algumas deficiências. Além de não permitir avaliação de desempenho do sistema, não permitem também a especificação de anti-cenários, ou seja, estados indesejáveis e de *liveness*.

Para a especificação desses sistemas, utilizamos, neste trabalho, uma linguagem para modelagem de cenários chamada *Live Sequence Chart* (LSC)[3]. Utilizando LSC, *liveness* e anti-cenários poderão ser testados e haverá a possibilidade de associarmos tempo aos cenários. Entretanto, LSC possui algumas deficiências, e, por isso, uma linguagem matemática conhecida como Redes de Petri Coloridas[4] será adotada. Através dessas redes o modelo poderá ser analisado e propriedades a respeito da estrutura, bem como do comportamento dinâmico do sistema, poderão ser coletadas.

Dessa forma, o objetivo desta monografia é apresentar uma ferramenta de mapeamento, onde cenários do tipo LSC são mapeados em redes de Petri equivalentes. Ao final, um estudo de caso realizado sobre um aparelho para medição de hidrômetros será descrito. Neste ponto, serão mostrados os resultados da ferramenta de mapeamento desenvolvida.

Abstract

Embedded systems or inlaid systems have been used in everyday life. Due to the low technological development cost, a clear trend of growth in the production of these embedded applications is noticed.

However, a design of this kind of system has come across some problems, since it is necessary to define all hardware/software architecture. Moreover, projects of these systems are extremely expensive.

When developing a system, it is important to be able to test and simulate the model. When modeling and testing a system from the initial phases of its implementation, it is possible to identify requirements, as absence of states where the system meets stopped (deadlock) and to detect errors of project specification, as well as errors of implementation, preventing problems in later phases of the project.

Several approaches have been created to model real-time systems, where requisite non-functional and functional requirements must be guaranteed. Some examples are the Message Sequence Chart (MSC)[1] and the Unified Modelling Language (UML)[2]. However, they present some deficiencies. Apart from not allowing performance evaluation, it also does not allow the specification of anti-scenarios and liveness.

For the specification of these systems, we use, in this work, a language for scenarios modeling called Live Sequence Chart (LSC)[3]. Using LSC, liveness and anti-scenarios could be tested and it is going to be possible to associate time to the scenarios. However, LSC possesses some deficiencies, and, therefore, a known mathematical language as Colored Petri Nets[4] will be adopted. Through these nets, the model could be analyzed and properties regarding the structure, as well as of the dynamic behavior of the system, could be collected.

Thus, this monograph aim is to present a mapping tool, where LSC scenarios are maped in equivalents Peri nets. In the end, a case study carried through on a device for measurement of hydrometers he will be described. At this point, the results of the developed tool of mapping will be shown.

Sumário

Índice de Figuras

Tabela de Símbolos e Siglas

MSC – Message Sequence Chart
UML – Unified Modelling Language
LSC – Live Sequence Chart
CPN – Coloured Petri Net
SOC – System On Chip
OOAD – Object Oriented Analysis and Design
ITU – International Telecommunication Union
INA – Integrated Net Analyzer
JDOM – Java Document Object Model
DOM – Document Object Model
GUI – Guide User Interface
GSM – Global System for Mobile Communications
GPRS – General Packet Radio Service
TCP – Transmission Control Protocol
IP – Internet Protocol

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao grande Deus pelo dom da vida que ele me proporcionou. Sem a sua constante presença ao meu lado, eu não conseguiria chegar onde finalmente estou. Ele, durante toda a minha graduação, proporcionou-me paz, conforto e segurança, servindo como ponto forte de consolo nas horas de tristeza e agonia.

Também gostaria de agradecer aos meus familiares, em especial àqueles que conviveram comigo durante todo o percurso da graduação. Agradeço a meus pais, Severina Rita de Oliveira Rodrigues e Mário José Rodrigues, pela força, apoio, dedicação e, sobretudo pelo amor prestado nas horas mais desestimulantes. Agradeço a meus irmãos, Caio Marcello de Oliveira Rodrigues e Cintya Millena de Oliveira Rodrigues, pelas brincadeiras e amizade oferecidas. Agradeço as minhas sobrinhas Sarah Beatriz Rodrigues da Cruz e Kauanne Silva, pelo carinho e pelo amor prestado. Agradeço também a primos, cunhados, tios, tias, por todo apoio e demonstramento de amizade.

Gostaria de agradecer também àqueles responsáveis pelo curso de Engenharia de Computação e quem estão cumprindo tão bem o papel de docentes dentro da UPE, em especial ao meu orientador e amigo, Ricardo Massa Ferreira Lima, com o qual foram mais de três anos de convivência, marcados por amizade, concordâncias e discordâncias, mas, sobretudo pelo respeito. Gostaria de agradecer também a Leonardo Barros Amorim, com o qual trabalhei durante um ano e meio neste projeto de pesquisa, e foi quem ofereceu as principais bases para que esse projeto pudesse ter alcançado seu objetivo.

Agradeço também aos meus colegas de trabalho da Procenge – Processamento de Dados e Engenharia de Sistemas Ltda, em especial a minha gerente de engenharia de software, Luciana Romeiro França, a qual soube demonstrar paciência e discernimento, nos momentos em que estava extremamente atarefado com minhas obrigações da faculdade, como esta monografia.

Por fim, mas não menos importante, gostaria de agradecer aos meus amigos, os quais posso chamar de meus irmãos. Durante a graduação, eles tornaram-se pessoas dignas de confiança e prestadoras de uma palavra de consolo nos momentos mais delicados. Em especial, agradeço a César Augusto Lins de Oliveira, Laura Arahata de Moraes, Marcelo Nunes Alves Costa, Milena Rodrigues Pinheiro de Souza, Nivia Cruz Quental e Petrônio de Luna Braga, como também a todos os outros que juntos comigo percorreram esta estrada da graduação em Engenharia da Computação.

Muito obrigado.

Capítulo 1

Introdução

Os sistemas embarcados, ou encapsulados, são sistemas computadorizados projetados para realizar tarefas pré-definidas. Apresentam-se encapsulados pelos aparelhos que estes controlam e em seu núcleo principal possuem um microprocessador encarregado de realizar a(s) tarefa(s) para as quais foram projetados. Desde o primeiro sistema embarcado de que se tem notícia, o computador de bordo da nave Apollo[5], o mercado deste tipo de sistemas vem crescendo de forma rápida. Isto pode ser evidenciado pela enorme quantidade de aparelhos embarcados que existem no mercado e que a população tem acesso.

Desde as primeiras aplicações na década de 60 do século passado, enquanto o preço desses sistemas tinha uma tendência de queda, o poder de processamento destas aplicações crescia de forma acentuada. Esta evolução é evidenciada pela famosa Lei de Moore[6], a qual mede a quantidade de transistores por placas de circuito. A representação gráfica desta lei pode ser visualizada na Figura 1. Segundo esta lei, a quantidade de transitores por placa de circuito é aumentada em 10 vezes a cada 5 anos, sendo que após o ano 2000, estas placas possuem bilhões de transistores por placa de circuito. A enorme quantidade de transistores em uma única placa favorece o desenvolvimento das aplicações *System on Chip*, SoC[7], tornando os sistemas mais complexos e poderosos.

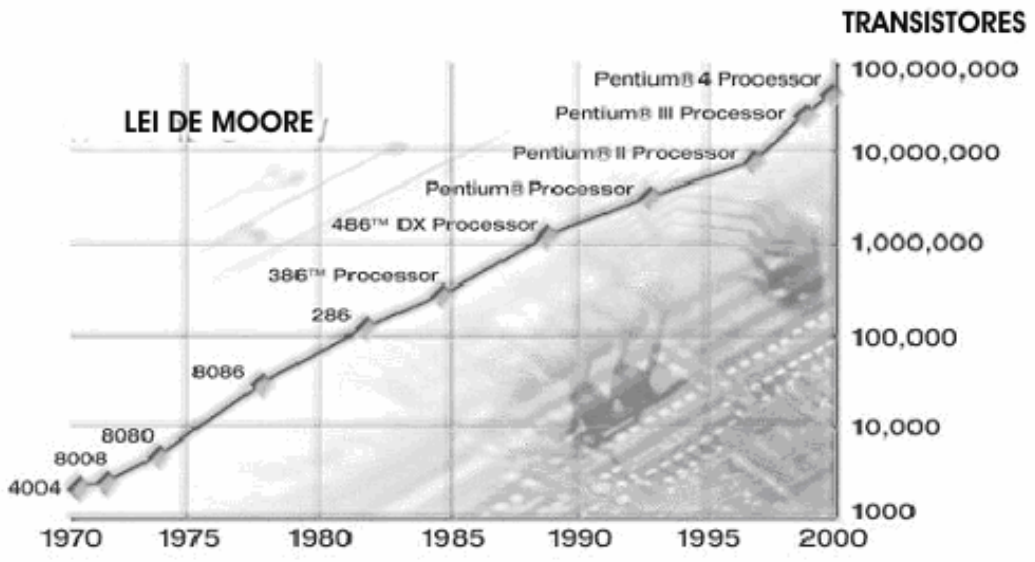


Figura 1. Lei de Moore[6]

Com a queda dos preços e com as otimizações nas placas de circuitos, mais e mais aplicações passaram a adotar essa nova abordagem de forma que, hoje em dia, existem diversos aparelhos embarcados como, por exemplo, celulares, computadores pessoais, roteadores, microondas, equipamentos médicos, entre outros.

Entretanto o projeto desses sistemas é extremamente complexo, uma vez que é necessário investigar diversas características destes, como, portabilidade, consumo de potência, segurança e confiabilidade. Um outro importante aspecto é decidir qual a melhor organização de *hardware* e *software* que o aparelho deve ter, de forma a manter estes requisitos citados acima. Questões como tamanho e quantidade de memórias, quantidade de processadores, meios de comunicação entre os mais diversos blocos internos do sistema, estilo de projeto, interface com os periféricos deverão ser vistas e julgadas, visando atender todos os requisitos desejados. Tais requisitos são essencialmente importantes em sistemas embarcados de tempo real, que possuem restrições temporais rígidas. Assim, estes sistemas não podem comportar-se fora do especificado, mesmo na presença de falhas.

Além do tempo gasto para que se possa achar a melhor configuração, isto é, a melhor combinação de *software* e *hardware*, e a mais rentável para o sistema, têm-se que empregar também uma boa quantidade do tempo de projeto para fazer validações individuais para cada estrutura do sistema, bem como realizar uma validação geral após a combinação dessas partes menores. O ideal é testar o projeto exaustivamente antes da produção do sistema embarcado em larga escala.

O projeto de um sistema embarcado inicia-se com a especificação em alto nível de abstração das funcionalidades exigidas. Esta especificação não possui informações detalhadas a respeito das implementações do sistema. Além disso, para fins de validação, faz-se necessário que tal especificação possa ser de alguma forma, executada. O sistema pode ser descrito, por exemplo, através de um grupo de objetos que se comunicam mutuamente através de mensagens, em que cada mensagem carrega informações a respeito da execução do sistema. Essa especificação em alto nível possibilita que validações sejam realizadas, e os resultados obtidos servirão como referência no processo da escolha da configuração da arquitetura do sistema

Uma das abordagens para modelagem de sistemas empregada tem sido a *Object Oriented Analysis and Design* (OOAD) [8], isto é, análise e projeto orientado a objetos. É caracterizada pelo uso de uma linguagem orientada a objetos para implementar os objetos do sistemas, responsáveis por capturar as suas funcionalidades. As análises são feitas em cima de casos de uso, os quais são modelados e, posteriormente, implementados. Um formalismo bastante conhecido que segue esta metodologia é a *Unified Modelling Language* (UML) [2].

Como dito anteriormente, é importante validar o sistema. Essa validação pode ser feita tanto através de simulações ou mesmo através de uma verificação formal. Durante essa validação, será investigado se algumas exigências estão presentes no sistema. Os sistemas embutidos, por exemplo, devem apresentar duas características essenciais para o seu perfeito funcionamento: segurança e *liveness.*

Uma nova forma baseada na metodologia da OOAD é o *Message Sequence Chart* (MSC) [1]. Assim como o UML, MSC especifica os cenários do funcionamento do sistema através de mensagens trocadas por objetos, funções e processos.

Através do cenários MSC, pode-se realizar simulações e validações dos requisitos do sistema, e documentações do próprio sistema. Para evitar problemas de ambigüidade nas interpretações de suas estruturas, a semântica do MSC foi formalizada e é atualmente mantida pela *International Telecommunication Union* (ITU) [9].

Através do MSC é especificado o funcionamento do sistema. Este funcionamento pode ser visto em forma de cenários, que representam os objetos, as funções, e o relacionamento entre eles. Todavia, MSC apresenta alguns problemas graves[10]. O primeiro deles é que a semântica

da linguagem é fraca de forma que não podemos ter certeza de que o sistema irá fazer aquilo que foi especificado nos cenários. Ou seja, nós podemos descrever os possíveis fluxos de execução, mas não podemos afirmar que eles efetivamente ocorrerão. Um outro ponto fraco é a incapacidade de especificar anticenários, isto é, estados que sob hipótese nenhuma possam existir. Isto é fundamental para garantirmos que exigências como segurança e *liveness* sejam preservadas. O UML também não permite verificar propriedades críticas do sistema, além de não permitir a avaliação de desempenho.

Para transpor as barreiras impostas por estas metodologias, foi proposto em 1998 por David Harel e W. Damm uma nova linguagem para a especificação de sistemas, na qual as propriedades de segurança e *liveness* pudessem ser preservadas nos cenários de modelagem. Esta linguagem é a LSC[3], isto é *Live Sequence Chart*. Além disso, David Harel junto com Remi Marelly desenvolveram a *Play Engine*[11], ferramenta através da qual os cenários são desenvolvidos como também simulados. A LSC resolve os problemas levantados pelo MSC, uma vez que, através dela podemos diferenciar os cenários que podem acontecer daqueles que devem acontecer. Isto é obtido através da introdução dos conceitos de cenários existencias e cenários universais[12]. O primeiro tipo, à semelhança do MSC, serve para representar simples cenários que podem convenientemente acontecer durante a execução. Os cenários universais, por sua vez, possuem uma pré-condição. Quando essa pré-condição for satisfeita, todos os eventos que estão descritos posteriormente a esta pré-condição devem obrigatoriamente ocorrer.

Contudo, LSC também possui algumas deficiências, como, por exemplo, a incapacidade de realizar avaliação de desempenho e de verificar propriedades do sistema.

Na década de 60, em sua tese de doutorado *Kommunication Mit Automaten*, Carl Adam Petri descreveu o conceito das redes de Petri [13]. Estas podem ser vistas como um modelo matemático para a modelagem de sistemas concorrentes, paralelos, não-determinísticos e para a modelagem de sincronização em sistemas distribuídos. Este modelo, por ter uma abordagem visual, auxilia o entendimento do comportamento de sistemas. Além disso, elas permitem avaliação de desempenho e utilizando-se de algumas ferramentas disponíveis no mercado, como a *Integrated Net Analyzer* (INA) [14], ou mesmo a *CPN Tools* [15], análises a respeito das propriedades do sistema podem ser realizadas. Dessa forma, essas redes resolvem os problemas apresentados pelo LSC.

Ao descrever sistemas embarcados através destas redes de Petri tradicionais, a descrição tende a ser grande e complexa, devido à própria complexidade de um sistema embutido. Entretanto, existe uma subclassificação das redes de Petri conhecida como *Coloured Petri Nets* (CPN) [4]. As redes de Petri coloridas são redes hierárquicas que possibilitam condensar a descrição do sistema. Estas redes conseguem esse feito porque utilizam tipos de dados abstratos e, além disso, utilizam o poder de uma linguagem de programação conhecida como CPN ML [16], uma variação da linguagem ML [17].

Pelo fato da linguagem LSC utilizar noções de orientações a objetos para representar fielmente o funcionamento dos módulos de um sistema qualquer, iremos trabalhar com as redes de Petri coloridas, uma vez que, estas oferecem maneiras de realizar validações do modelo, no que se refere às propriedades do sistema e à avaliação de desempenho. Além disso, as redes coloridas são hierárquicas o que possibilita a criação de redes mais otimizadas e por fim, diferente das redes tradicionais, cada *token* é indistiguível, ou seja, cada marca da rede possui um valor o que facilita a manipulação de dados na rede. Dessa forma, análises de *liveness*, de ausência de *deadlock* (isto é, ausência de estados mortos), grafos de alcançabilidade, reversibilidade, persistência, entre outros, podem ser realizadas. No Capítulo 3, estas análises são descritas de forma mais clara e objetiva.

Como foi discutido anteriormente, devido à complexidade de um sistema embarcado, existem vários problemas que podem levar à falha do projeto. Por isso, é importante realizar a

validação do modelo buscando sempre a corretude e a robustez do sistema. Este compilador será capaz de ler um modelo formal baseado no LSC e mapeá-lo em uma rede de Petri colorida equivalente. Uma pergunta que pode surgir é: porque não criar já um modelo baseado em uma rede de Petri, visto que isso descartaria uma fase intermediária? A resposta é que, nas fases iniciais do projeto de um sistema, é mais intuitivo para um ser humano descrever seu funcionamento através de objetos e da comunicação entre eles. Além disso, como será discutido no Capítulo 2, a ferramenta *Play-Engine*, responsável por desenvolver os cenários, disponibiliza um método chamado de *Play In* [11], que torna o desenvolvimento do modelo formal fácil e intuitivo.

Esta monografia está organizada da seguinte forma: no Capítulo 2, serão discutidas as linguagens para modelagem dos cenários MSC e LSC, levantando os pontos positivos e negativos de cada uma; em seguida, no Capítulo 3, iremos detalhar o conceito de redes de Petri e das Redes de Petri Coloridas, suas características e variações; no Capítulo 4 mostraremos a implementação do compilador para a fazer a tradução de um modelo baseado em LSC para um outro baseado em uma rede de Petri colorida; no Capítulo 5 será discutido um estudo de caso realizado em um leitor de hidrômetro e por fim, no Capítulo 6 serão mostradas as conclusões destes trabalhos, bem como definiremos algumas perspectivas para trabalhos futuros.

Capítulo 2

Cenários de Seqüência

Neste capítulo são descritas algumas abordagens existentes para a criação de cenários, entre elas a MSC e a LSC levantando os pontos positivos e os pontos negativos de cada uma e mostrando o porque da escolha do LSC neste projeto.

2.1 Message Sequence Chart

Message Sequence Charts (MSC) [1] é uma linguagem gráfica para especificação do funcionamento de sistemas em geral através de cenários, caracterizados pela interação entre os processos, funções e objetos presentes. Foi padronizada pela ITU (*International Telecomunication Union*) [9], que mantêm as definições de sua semântica. Esta padronização, além de definir as construções sintáticas permitidas, também está acompanhada de uma semântica formal que não permite construções ambíguas.

A utilização de MSC é mais adequada nos estágios iniciais do desenvolvimento dos sistemas. Ao se pensar em como um sistema irá funcionar, primeiramente serão definidos alguns estudos de caso do mesmo e, estes, por sua vez, serão descritos através de cenários de estado (ou *state charts*). Em seguida, os comportamentos das classes do sistema serão descritos através de diagramas de estado, os quais definem o comportamento para cada instância da classe. Finalmente, os objetos são implementados como códigos em uma linguagem de programação específica. Contudo, a geração de código pode ser feita de forma automatizada, como por exemplo, através dos próprios cenários de estado. MSC é atualmente utilizado para especificar os cenários bons, isto é, aquelas situações que o sistema deve permitir [10].

Usualmente, MSC serve para modelar simples cenários de funcionamento de sistemas. Entretanto, quando o modelo de sistema torna-se refinado, a utilização de MSC torna-se precária, uma vez que ele fornece uma interpretação fraca do sistema. Um dos primeiros problemas a emergir é a incapacidade de separar aqueles cenários que podem acontecer daqueles cenários que devem acontecer. Alguns problemas do MSC são listados abaixo:

- **Fluxo de Execução** - Só pode ser especificado um único fluxo de execução dentro de um cenário. Isto demonstra a incapacidade de definir diferentes fluxos do funcionamento de um sistema dentro de uma única especificação. O MSC não dispõe dos elementos básicos para suportar *branching* e interação;

- ***Liveness*** - De acordo com a semântica do MSC, não se pode ter a certeza de que um fluxo realmente irá acontecer, isto é, ele pode acontecer, mas não há garantias de que ele realmente acontecerá. Isso demonstra a incapacidade de definir *Liveness*, isto é, não se pode afirmar que uma coisa boa eventualmente ocorrerá;
- **Condição** - Não existe uma semântica formal para especificar condições em cenários MSC;
- **Eventos simultâneos** - Não existe noção de simultaneidade, ou seja, dois eventos não podem ocorrer exatamente ao mesmo tempo;
- **Tempo de ativação** - Não se pode definir nenhum estado de ativação, ou seja, não se tem certeza de quando o cenário MSC será ativado;
- **Tratamento do tempo** - Com relação ao tempo, MSC permite a definição de intervalos, mas a semântica não dá suporte a durações de tempo.

Da mesma forma que MSC, a semântica da UML permite criar os cenários possíveis de execução, mas não permite definir aqueles cenários obrigatórios. Isto prejudica análise de *liveness*. Devido a essas deficiências, foi definida uma nova extensão do MSC, conhecida como LSC ou *Live Sequence Chart*, que resolve estes problemas.

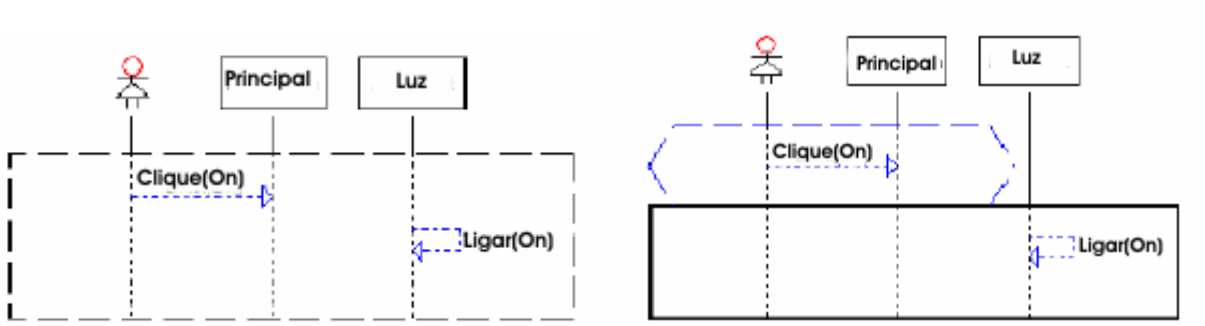
2.2 Live Sequence Chart

Nesta seção serão explicados os motivos que fizeram de LSC a linguagem utilizada para a modelagem dos cenários do modelo formal neste projeto. Dentre estas, saliente-se que a principal característica é a possibilidade de distinguir entre os fluxos que podem acontecer daqueles que devem acontecer[12]. E isto pode ser representado graficamente nos cenários, o que facilita a compreensão.

LSC também é uma linguagem de especificação baseada na metodologia OOAD[8]. Através dela torna-se possível expressar condições que não podem ocorrer em hipótese alguma e estados que devem permanecer verdadeiros durante todo e qualquer fluxo de execução. Estes dois novos pontos conseguem resolver as duas principais exigências buscadas na especificação em alto nível de um sistema: segurança e *liveness*. Isto também é conseguido pelo emprego de novos tipos de cenários para a modelagem, os cenários existenciais e os cenários universais. Graficamente, os cenários são formados por objetos, cada qual com seu tempo de vida representado por uma linha vertical do topo até a base do cenário. Esses objetos, os quais representam funções ou processos, se comunicam através de trocas de mensagens representadas graficamente por uma linha horizontal. O fluxo de execução ocorre de cima para baixo do cenário e da direita para esquerda.

Para expressar um simples fluxo de execução, indicando que este pode ser executado, utilizaremos os chamados cenários existenciais. Por outro lado, para modelarmos aqueles fluxos que obrigatoriamente precisam ser executados, utilizaremos os cenários universais. Estes são formados por duas seções: a pré-condição e o corpo do cenário. A pré-condição corresponde graficamente a um hexágono tracejado, o qual representa uma condição que, se satisfeita, força o fluxo de execução para o corpo do cenário, ou seja, caso o primeiro ocorra, o segundo deve obrigatoriamente ser executado. Se isso, porventura, não ocorrer, ocasionará uma violação do sistema. Ambos os tipos de cenários são ilustrados na Figura 2. No primeiro caso, o cenário de execução mostra um simples fluxo de execução que ocorre quando um usuário externo ao sistema modifica a instância Principal para “*On*”. Após isso, pode acontecer da instância Luz modificar a si mesma para o estado “*On*”. Por outro lado, na Figura do cenário universal, quando o usuário realizar a ação sobre a primeira instância (isto representa a pré-condição), obrigatoriamente a

segunda instância deve modificar seu estado. Este tipo de cenário introduz o conceito de ação-reação entre a pré-condição e o corpo do cenário. Neste segundo caso, nota-se na figura que, enquanto a primeira instância participa tanto da pré-condição quanto do corpo do cenário, a segunda participa apenas do corpo do cenário.



Cenário Existencial Cenário Universal

Figura 2. Tipos de Cenários

Localmente, a semântica do LSC permite a criação de condições, tempos e emprega elementos quentes e frios. Através dos chamados elementos quentes poderemos especificar *liveness* e usando os elementos frios pode-se definir *branching* e interações. Uma condição quente, por exemplo, dentro de um cenário especifica uma condição que, quando atingida pelo fluxo deve ser verdadeira. Caso contrário acarretará uma violação do sistema. Uma condição fria funciona de forma diferente. Diferentemente da primeira, quando o fluxo de execução atingí-la, ela pode estar verdadeira, como também pode ser falsa. Isso vai fazer com que o fluxo possa continuar por dois caminhos distintos, dando a característica de *branching*. Fica claro assim, que poderemos definir construções *if-then-else* e *while-do* utilizando as estruturas do LSC.

De forma geral, um cenário LSC é definido por (segundo [18])

$L:=(I_L, V_L, M_L, [Pch_L], A_L, C_L, SUB_L, ITE_L, LOOP_L, M'_L, C'_L$, *strict, event, subchart, temp*),

Onde

1. I_L é um conjunto das instâncias LSC;
2. V_L é o conjunto das variáveis;
3. M_L é o conjunto de mensagens. Uma mensagem Mi $\in$ M_L é representada graficamente por uma flecha com um ponto inicial na linha de vida de uma instância, a qual representa o evento de enviar a mensagem, e outro ponto em uma segunda linha de vida da segunda instância (pode ser a mesma), que representa o evento de recebimento.
4. Pch_L corresponde ao conjunto de pré-condições (caso tratemos de cenários universais);
5. A_L é o conjunto de atribuições;
6. C_L é o conjunto de condições;
7. $LOOP_L$ é o conjunto de *loops;*
8. M'_L é o conjunto de mensagens proibidas;
9. C'_L é o conjunto de condições proibidas;
10. SUB_L é o conjunto de *subCharts;*
11. ITE_L é o conjunto de estruturar *if-then-else;*
12. *Strict* é um valor *booleano;*
13. *temp* é uma função que atribui uma temperatura para mensagens, condições, mensagens e condições proibidas;
14. *event* é uma função que mapeia uma localização a um evento;
15. Funções *subchart* são aplicadas a estruturas internas de um *chart* LSC.

Nas próximas Seções, são listadas as principais características do LSC, seguidas de uma breve discussão de cada uma.

2.2.1 Instâncias e Mensagens

Instâncias e Mensagens correspondem aos blocos principais do LSC. A representação gráfica de uma instância segue aos padrões do próprio MSC. Como já abordado, as linhas verticais de cada instância correspondem ao tempo em que a instância permanece "viva" dentro do cenário. Além disso, utilizando-se LSC pode-se definir formalmente uma instância *User* ou *Environment* (Env) como pode ser visto na Figura 3 (referenciada em [19]). Ao usar LSC para verificações formais, uma instância do tipo Env oferece a possibilidade de expressar características reais da interação do ambiente com o cenário, empregando o mesmo conjunto de elementos oferecidos para tratar as outras instâncias. Além disso, mensagens podem representar interações síncronas ou assíncronas. Por exemplo, um objeto, ao enviar uma mensagem para si próprio, caracteriza uma mensagem síncrona. Contudo se, por exemplo, quisermos expressar o atraso de interação entre um usuário ou um ambiente e o tempo em que o objeto percebe a ação realizada, pode-se utilizar propriedades assíncronas. Graficamente, uma mensagem síncrona é representada por uma flecha com uma cabeça triangular fechada e uma mensagem assíncrona, por uma seta com uma cabeça triangular aberta.

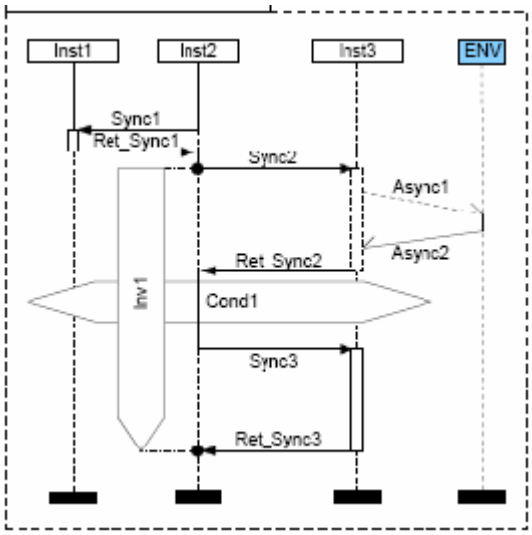


Figura 3. Exemplo de um cenário MSC com uma entidade do tipo ambiente[19]

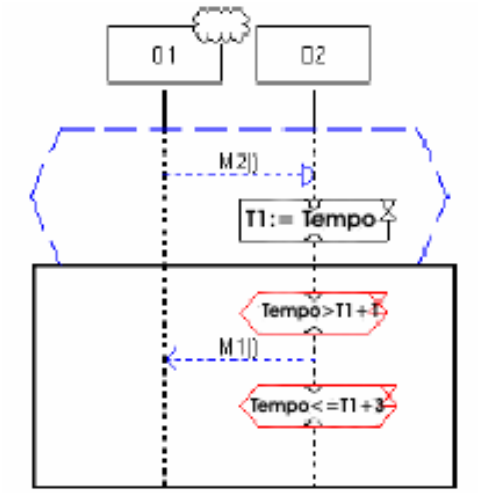


Figura 4. Mensagens síncronas e assíncronas

Na Figura 4, pode-se perceber a diferença gráfica entre uma mensagem síncrona, localizada na pré-condição, e uma mensagem assíncrona, localizada no corpo do cenário. De uma forma geral, este LSC representa o seguinte: a instância O1 envia a mensagem M2() para O2. Simultaneamente (isto é expresso pela mensagem síncrona) o tempo que O2 recebe esta ação é guardado em uma variável T1. Por conseguinte, O2 tem de uma (notar o tempo da primeira condição) a três unidades (notar o tempo da segunda condição) de tempo para enviar a mensagem M1() para O1. Este tempo de atraso é expresso pelo assincronismo da mensagem.

2.2.2 Liveness e Temperaturas

A grande vantagem de LSC sobre MSC é a capacidade de forçar um fluxo de execução dentro do cenário. Isto é conseguido através do conceito de temperatura. A cada mensagem e aos pontos dos cenários são associados uma temperatura que pode ser quente ou fria. Uma mensagem quente necessita mudar de posição e achar um local mais adequado. Uma mensagem fria, por sua vez, não precisa mudar de posição e ocorre muitas vezes da mesma ficar perdida por um bom tempo.

Uma mensagem quente, denotada graficamente por linhas sólidas vermelhas, precisa ser recebida por algum objeto depois de enviada. Diferentemente, uma mensagem fria, denotada graficamente por linhas tracejadas azuis, não necessariamente precisa ser recebida depois de enviada. Uma clara aplicação disto são em sistemas de comunicação, onde se deseja avaliar a quantidade de mensagens que são enviadas, mas não são recebidas no destinatário.

2.2.3 Condições

Para realizar investigações a respeito do estado do sistema, a semântica de LSC fornece estruturas básicas para a construção de condições. Há dois possíveis tipos de condições: as possíveis (condições frias) e as obrigatórias (condições quentes). As primeiras representam simples condições que quando não satisfeitas causam uma saída do bloco LSC na qual está inserida.

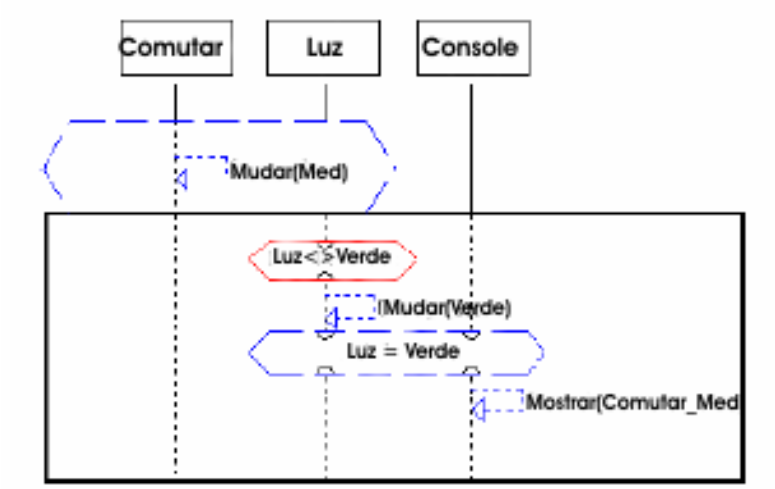


Figura 5. Condições quentes e frias

Já o segundo tipo representa condições que devem estar verdadeiras quando avaliadas. Caso isto não ocorra, uma violação do sistema é lançada. Em termos gráficos, uma condição quente é representada por um hexágono com linhas vermelhas sólidas. Por sua vez, uma condição fria é representada por uma linha azul tracejada. Ambas podem ser visualizadas na Figura 5. No primeiro caso é verificado se o valor da variável Luz é diferente do valor da variável Verde. Caso esta condição seja falsa é lançada uma exceção (contudo, mesmo com essa violação, ocorre que o sistema ficará nesta posição até que essa condição torne-se verdadeira). Já no segundo caso, se a condição não for verdadeira, apenas ocorre a saída do fluxo deste cenário. Ou seja, a ação que a instância Console exerce sobre si mesma, representada pela auto mensagem enviada Mostrar(Comutar_Med), não acontecerá.

2.2.4 Ativação, Quantificação e Pré-Condição

Como já foi dito no começo desta Seção, LSC introduz novos conceitos a respeito dos fluxos de execução de um sistema. Através da linguagem MSC, podemos apenas modelar fluxos que podem, em algum momento no sistema, ser executado. Em termos de LSC isto pode ser representado por cenários existenciais. Entretanto, LSC possui os cenários universais o qual possui um fluxo que é obrigatoriamente executado.

Diferentemente do que acontece no MSC, a ativação do cenário pode ser especificada. Ou seja, podemos determinar quando o fluxo de execução do cenário começará. Isto é conseguido pelo emprego das pré-condições. Quando uma pré-condição for satisfeita, podemos ter certeza de que o fluxo instantaneamente passará para a execução do corpo dos cenários, caso contrário, haverá uma violação das exigências do sistema.

2.2.5 Tempo

O conceito de tempo em LSC é uma evolução do que é utilizado no MSC, uma vez que há diversas abordagens nas quais ele pode ser utilizado (diferentemente do MSC, que só suporta os conceitos de intervalos de tempo). Assim, como condições e mensagens, LSC adota o conceito de tempos quentes e tempos frios.

Com relação aos tempos quentes (temporizadres utilizados em condições quentes), LSC permite a construção de três tipos de tempo: atraso vertical, atraso horizontal e temporizadores (ou *timers*). Um intervalo de tempo vertical permite definir o atraso mínimo e máximo permitido

entres dois eventos consecutivos em uma mesma linha de uma instância. Na Figura 6 (Atraso Vertical)[20], na pré-condição é armazenado o tempo depois que a instância O2 receber a mensagem da instância O1. O tempo de atraso mínimo é definido através de uma condição quente, expressa da seguinte forma:

TEMPO > TEMPO_VARIÁVEL + ATRASO_MÍNIMO. **(Inequação 1)**

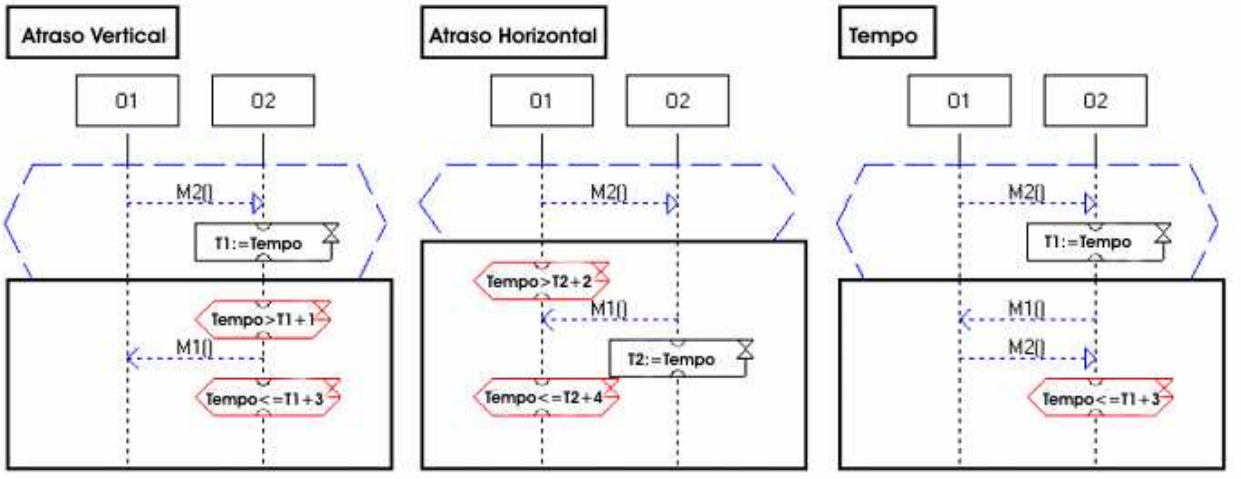
Quando o fluxo de execução atingir essa condição, ele apenas continuará quando ela for avaliada como verdadeira (a semântica do LSC garante que uma condição quente seja avaliada constantemente até que ela torne-se verdadeira) . Com relação ao atraso máximo, definido pela expressão:

TEMPO < TEMPO_VARIÁVEL + ATRASO_MÁXIMO **(Inequação 2)**

A avaliação da condição ocorrerá da mesma forma. Caso o fluxo atinja essa condição e o tempo seja menor do que o especificado, ela será avaliada como verdadeira e não haverá problemas. Por outro lado, caso o tempo seja superior, a condição nunca poderá ser avaliada como verdadeira, e por ser uma condição quente, haverá uma violação de exigência.

Na Figura 6 (Atraso Horizontal), temos o atraso das mensagens. Neste caso, estamos preocupados em saber o tempo que uma mensagem demorou em ser recebida por uma instância do cenário, após ter sido enviada. A diferença em relação ao atraso vertical está no fato de que o tempo é armazenado em uma instância e checado em outra (isto é feito para simular o objeto responsável pelo envio da mensagem e aquele responsável pelo recebimento da mensagem).

Finalmente, na Figura 6 (Tempo) temos os temporizadores. LSC permite uma maior flexibilidade no uso dos temporizadores, uma vez que eles podem ser empregados na avaliação de *timeout* na definição de atrasos mínimos e atrasos máximos, entre outros. Esta flexibilidade permite que especifiquemos um sistema mais próximo da sua realidade de execução.



Atraso Vertical Atraso horizonal Tempo

Figura 6. Variações de Tempo

Por outro lado, o conceito de tempo frio (temporizadores usados em condições frias) é definido de forma semelhante às condições frias. Ela pode ser bem explicada de acordo com a Figura 7 . A rainha do país das maravilhas solicita ao coelho estar presente em seu palácio em cinco segundos. Isto é representado por uma mensagem de solicitação. Neste momento, a rainha grava no seu relógio o tempo atual

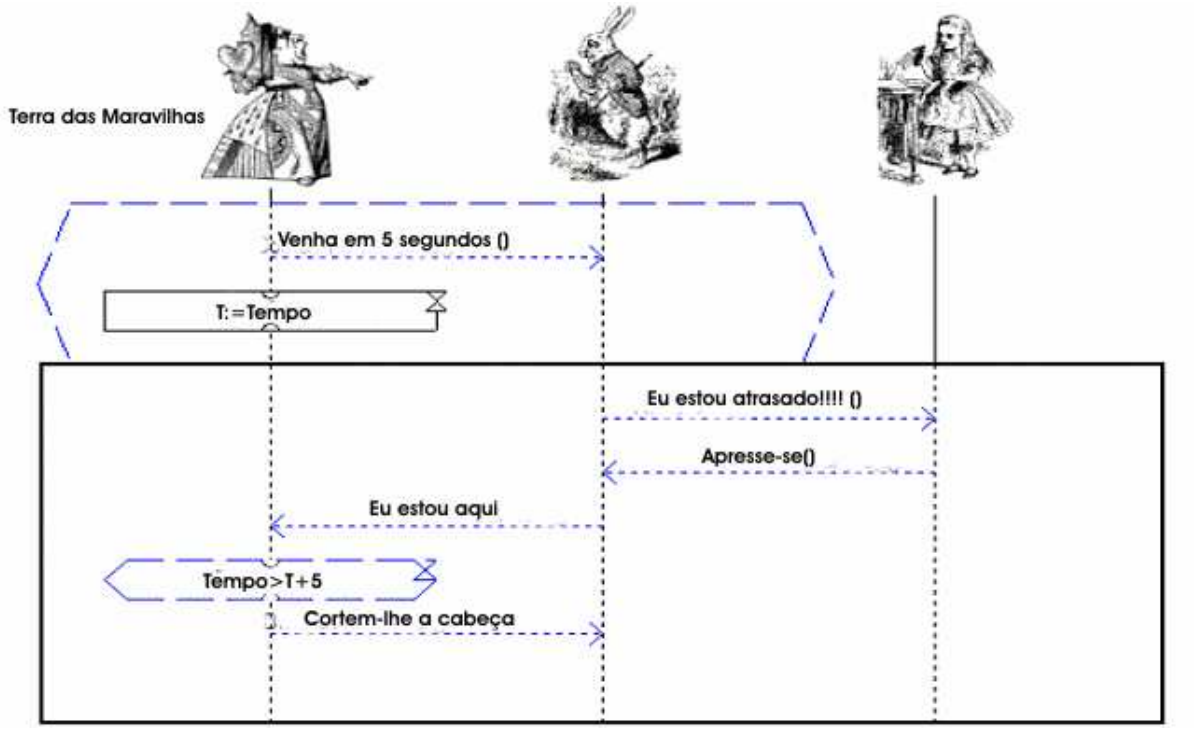


Figura 7. Tempo Frio [20]

O coelho, por sua vez, avisa pra Alice que está atrasado e esta pede para ele se apressar. Acontece que, caso o coelho chegue no tempo determinado (expresso através de uma condição fria), definido por:TEMPO < TEMPO_VARIÁVEL + 5 nada acontece. Contudo, caso ele não chegue nesses 5 segundos (TEMPO > TEMPO_VARIÁVEL + 5), a rainha pedirá que ele seja decapitado (representado pela ação logo abaixo da condição). A idéia, neste caso, é o emprego de *branching*. Se tivéssemos tratando com condições quentes isso não aconteceria. Se o fluxo atingisse a condição e ela não fosse avaliada como verdadeira, causaria uma violação de exigência.

2.3 Play Engine

A ferramenta *Play Engine*[11] foi desenvolvida para que cenários LSC pudessem ser criados e simulados. Nesta é possível desenvolver cenários utilizando elementos quentes e frios. Além disso, ela possui uma *symbol Table*, a qual mantêm todas as variáveis utilizadas nos cenários LSC.

Diferentemente do MSC, LSC não é uma linguagem adequada para a modelagem do sistema nos estágios iniciais de desenvolvimento Por esta razão, foi proposta uma abordagem em alto nível de sistema denominada *play in scenarios*[11], oferecida junto com a ferramenta *Play Engine*, que também suporta os cenários LSC. Isto proporciona aumentar o nível de abstração do sistema, de forma a facilitar o entendimento e o fácil manuseio deste. Além disso, o uso da metodologia de *play in* é bastante intuitivo.

Play In se baseia no fato do usuário poder construir uma *Guide User Interface* (GUI) em uma ferramenta gráfica e depois importá-la na *Play Engine*. Uma GUI é uma representação visual do aparelho que possui o sistema embarcado, com seus botões, telas, controles e outros objetos. Através da manipulação destes objetos (apertando os botões, chamando funções) é que serão descritas as propriedades, as variáveis, os tipos e todos os outros elementos que o sistema deve possuir. Da mesma forma, a cada interação deverão ser especificadas as reações que podem ou

devem (daí o conceito dos cenários existenciais ou universais) ser obtidas. Enquanto o usuário define estas características de uma forma intuitiva, será gerado, sem que este perceba, o código LSC referente.

O passo seguinte é testar a GUI para verificar se ela realmente faz aquilo que planejamos que ela fizesse. Isto é conseguido por uma outra abordagem adotada pela *Play Engine*, conhecida como *Play Out*[11]. *Play Out* se baseia em testar a GUI e verificar se as especificações estão realmente funcionando. Qualquer usuário pode realizar essa tarefa. Ele irá interagir com a GUI como se ela própria fosse um sistema real. Cada reação obtida para cada ação (ou interação) na GUI será estudada e analisada.

A metodologia LSC auxiliada pela ferramenta *Play Engine*, aliada ao seu fácil manuseio, proporciona as bases necessárias para podermos definir a especificação formal do sistema, que será, posteriormente, mapeada em uma outra especificação formal fiel ao sistema original e que proporcione a busca de propriedades e a investigação da sua eficiência. Isto porque, embora possamos simular os cenários criados pela ferramenta, para uma interpretação mais detalhada de seu comportamento para certas condições de entrada, é necessário ao invés de simularmos exaustivamente, propor análises e avaliações do modelo formal descrito. Além disso, existem no meio acadêmico ferramentas poderosas, com uma semântica bem definida, onde tais análises podem ser testadas.

A *Play Engine* é uma ferramenta paga e é atualmente a única ferramenta para auxiliar o desenvolvimento de cenários baseados na metodologia da LSC.

Capítulo 3

Redes de Petri

As redes de Petri [13] foram inicialmente definidas em 1962 por Carl Adam Petri em sua tese de doutorado *Kommunication Mit Automaten* na Universidade de Dramsdadt na antiga Alemanha Ocidental. É um modelo matemático capaz de modelar sistemas paralelos, concorrentes, assíncronos e não-determinísticos. As redes são consideradas uma evolução da teoria dos autômatos finitos[20], uma vez que essas redes, diferentemente destes últimos, são capazes de modelar o paralelismo dos sistemas. Por ser gráfica, facilita a compreensão e interpretação do funcionamento de sistemas. Uma rede, como visto na Figura 8, é formada por dois elementos principais:

- Os lugares, os quais representam os elementos passivos da rede. Os lugares representam os estados que um sistema pode assumir durante o seu fluxo de execução. Para tal, os lugares apresentam certos elementos denominados marcas ou *tokens*, os quais indicam a presença ou disponibilidade de algum recurso. É a disposição destas marcas nos lugares que define os estados da rede. Graficamente, os lugares são representados por círculos ou elipses e as marcas por pontos dentro dos lugares;
- As transições correspondem aos elementos ativos da rede, pois são as responsáveis por dar a característica do dinamismo desta. Quando certas pré-condições são satisfeitas, uma transição passa do estado de desabilitada para habilitada e pode disparar. O disparo das transições é responsável pela criação e pela destruição dos recursos (marcas) da rede. Graficamente, são representadas por retângulos dispostos horizontalmente, como também verticalmente.

A Figura 8 mostra dois lugares e uma transição de uma rede de Petri.

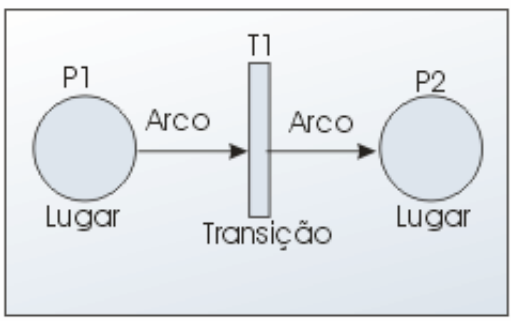
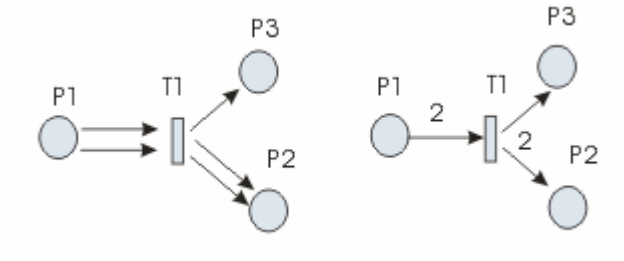


Figura 8. Elementos básicos de uma rede de Petri

Além de lugares e transições, existe um terceiro elemento responsável por estabelecer as conexões entre lugares e transições ou vice-versa, mas nunca de lugar para lugar ou de transição para transição. São os arcos. Associados a estes podem existir, por exemplo, pesos indicando o grau do arco. Ou seja, caso existam dois arcos ligando um mesmo lugar a uma mesma transição, isto corresponde a um único arco de peso '2'. Graficamente, o valor '2' é colocado próximo ao arco. Isto pode ser visto na Figura 9(a) e 9(b). Inicialmente, na Figura 9 (a), temos dois arcos com ponto de partida no lugar P1 e ponto de chegada na transição T1, cada qual com peso 1. Na Figura 9(b), ambos arcos são representados por um único arco, porém com peso 2. Esta segunda representação é a adotada na comunidade acadêmica das redes de Petri.



(a) (b)

Figura 9. Multiplicidade dos Arcos

Uma rede de Petri marcada é definida por uma quintúpla (P, T, I, O, M_0) [13] onde:

- P = {p_1, p_2, p_3, ...,p_m} é o conjunto dos lugares da rede, sendo que 'm' representa o identificador do lugar da rede;
- T = {t_1, t_2, t_3, ... ,t_n} é o conjunto das transições da rede. Da mesma forma, 'n' representa o identificador da transição;
- I:(P,T) é a função que define o número de arcos de entrada das transições. Por exemplo, para dizer que existem k arcos saindo do lugar p_m e chegando a transição t_n poderíamos representar desta forma: $I(p_m, t_n) = k$;
- O:(T,P) por outro lado, representa o conjunto de arcos que saem das transições e chegam nos lugares da rede.
- M_0 é a marcação inicial da rede, definida pela distribuição das marcas nos lugares. M = ($M(p_1)$, $M(p_2)$, $M(p_3)$, ..., $M(p_j)$), onde $M(p_j)$ define a quantidade de marcas no lugar p_j, sendo que $p_j \in P$.

Existem pré-condições que habilitam o disparo de uma transição. O disparo de uma transição caracteriza-se pela mudança de estado da rede, o qual acontece devido a mudança da posição das marcas em relação aos lugares. De forma geral, dize-se que uma transição t_i está habilitada se para cada lugar p_m que é entrada da transição ($I(p_m,t_j) > 0$) o peso dos arcos que ligam os lugares às transições é no mínimo igual a quantidade de marcas de cada lugar. Essa definição pode ser expressa pela seguinte fórmula:

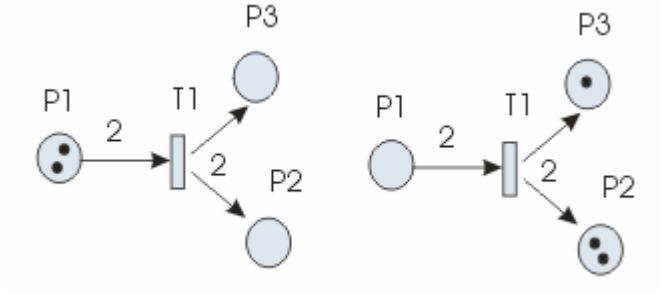
$$M(p_j) \geq I(p_j , t_i) \qquad \textbf{(Inequação 3)}$$

Dessa forma, caso a transição habilitada dispare haverá uma mudança de estado da rede, que pode ser expressa, por sua vez, da seguinte maneira:

$$M_i[t_j > M'$$

Esta representação afirma que a rede passou da marcação M_i para a marccação M', devido ao disparo da transição t_j. Caso a mudança ocorra devido aos disparo de mais de uma transição, podemos representar dessa forma: $M_i[sq > M'$, onde sq: t_0; t_1; ...; t_k.

Vale ressaltar que, ao habilitar uma transição, esta não precisa obrigatoriamente disparar. Isto é o que dá o caráter não-determinísticos das redes. Contudo, caso o disparo aconteça, será retirado dos lugares de entrada um número de marcas igual ao peso dos arcos de entrada ($I(p_m,t_i)$) e será adicionado aos lugares de saída um número de marcas igual ao pesos dos arcos de saída ($O(t_i,p_n)$). Este fluxo das marcas pode ser visualizado na Figura 10(a) e na Figura 10(b):



(a) (b)

Figura 10. Habilitação e disparo de uma transição

Note que na Figura 10(a) a transição T_1 está habilitada, pois $(I(P_1,T_1)) = 2$ e $M(P_1) = 2$. Isto é, a relação $M(p_j) \geq (I(p_j,t_i))$ é verdadeira. Ao disparar a transição T_1, retiram-se marcas do lugar P_1 e cria novos recursos nos lugares P_2 e P_3. Como o peso do arco $O(T_1, P_3)$ é igual a 1, é adicionada uma marca no lugar P3. Da mesma forma, como $O(T_1,P_2)$ é igual a 2 são adicionadas duas marcas no lugar P_2.

Existem ainda dois tipos especiais de transição. Uma, chamada de transição fonte, é caracterizada por sempre estar habilitada, isto porque não há lugares de entrada para a transição, ou seja,

$$I(p,t) = 0, \forall\ p \in P. \qquad \textbf{(Equação 1)}$$

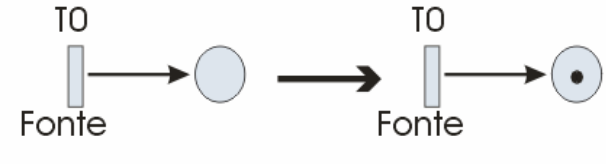


Figura 11. Transição Fonte

A outra transição é a transição de absorção, a qual consome marcas dos lugares de entrada, mas não cria outros recursos na saída. Isto acontece devido a ausência de lugares de saída para a transição. Da mesma forma:

$$O(t,p) = 0, \forall\ p \in P. \qquad \textbf{(Equação 2)}$$

Figura 12. Transição Absorção

3.1 Matriz de Incidência

A matriz de incidência serve para representar a estrutura dos sistemas. Ela representa a incidência dos arcos de entrada e saída em cada transição da rede. Sendo I a matriz de entrada da rede, O a matriz de saída, a matriz de incidência C é representada pela relação PxT → Z, definida formalmente por:

$$\forall\, p \in P, \forall\, t \in T \mid C(p,t) = O(p,t) - I(p,t) \qquad \textbf{(Equação 3)}$$

A matriz de incidência serve para retirar informações importantes da rede, as quais devem ser utilizadas nas análises comportamentais, ou seja, análises baseadas no comportamento da rede definido pelo fluxo das marcas. Entretanto, caso a rede seja impura, a matriz de incidência não mostra completamente a estrutura da rede. Uma rede impura é caracterizada por apresentar *self-loops*, isto é, um lugar $p_i \in P$, que é ao mesmo tempo pré e pós-condição de uma transição t_j. De uma forma geral, uma rede é dita pura se e somente se:

$$I(pj,ti) * O(pj,ti) = 0, \forall\, t_i \in T, \text{ e } \forall\, p_j \in P \qquad \textbf{. (Equação 4)}$$

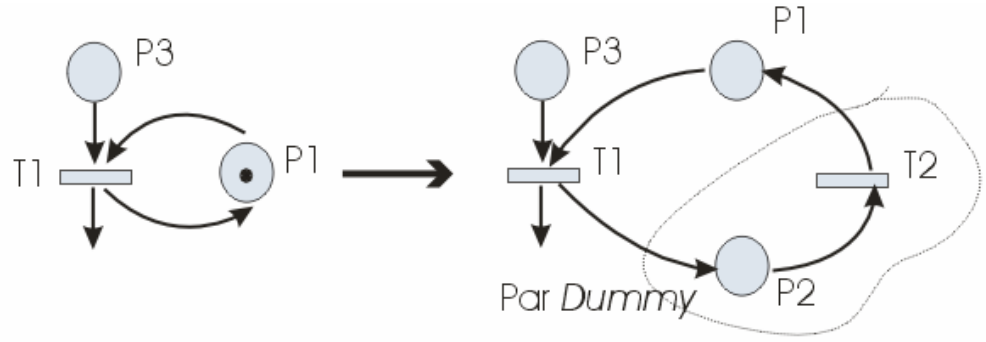


Figura 13. Refinamento de uma Rede

Para tratarmos esse problema, antes de calcularmos a matriz de incidência de uma rede impura, devemos refiná-la, isto é, eliminar os *self-loops*. Isto é feito através da inserção de um par *dummy* (Figura 13), o qual é composto por um lugar e um transição. O par *dummy* elimina o *self-loop* porque o lugar e a transição são interpostos entre a ligação do lugar e da transição que estão gerando o *loop*.

T1	
0	P1
-1	P3

	T1	T2	
P1	-1	1	P1
P2	1	-1	P2
P3	-1	0	P3
	.	0	.

Matriz da rede impura Matriz da rede pura

Figura 14. Matrizes de Incidência

Na Figura 14 podemos verificar a matriz de incidência da rede apresentada na Figura 10, antes e depois dela ser refinada pela utilização do par *dummy*. Uma Matriz de incidência é formada pela relação das transições (na parte horizontal da matriz) e pelos lugares (parte vertical). Para cada relação transiçãoxlugar, iremos realizar a subtração da quantidade dos arcos de saída pelos arcos de entrada. Tomando a relação do lugar P_1 com a transição T_1 na rede refinada, por exemplo, vemos que não há nenhum arco de saída de T_1 para P_1. Entretanto, há um arco de entrada de P_1 para T_1. Por isso, o valor dessa relação é '-1'. No caso da matriz da rede impura (Figura 14), o valor dessa relação é '0'. Isso dá a falsa idéia de que não há conexão entre P1 e T1. Por isso, é que a matriz de incidência das redes impuras não mostram por completo a estrutura da rede.

As redes adotam o conceito de vetor característico $\bar{s}$, que é um vetor do tamanho da quantidade de transições (#T), onde cada posição representa a quantidade de disparos daquela transição: $\bar{s} = [s(t_0)^T + s(t_1)^T + ... s(t_k)^T]$ **(Equação 5)**

3.2 Equação Fundamental

Também conhecida como equação dos estados ou equação das marcações, a equacação fundamental descreve o comportamento das redes possibilitando a sua análise estrutural e comportamental. Detalhes sobre estas análises podem ser encontradas na Seção 3.3.

A idéia desta equação é a partir da matriz de entrada, da matriz de saída das transições e da marcação atual da rede, poder definir as possíveis novas marcações da rede, resultantes do disparo de alguma transição. Essa equação pode ser alcançada a partir da matriz de incidência e do vetor característico, sendo expressa da seguinte forma:

$$M'(p) = M_0(p) + C.\ \bar{s}\ ,\ \forall\ p \in P \qquad \textbf{(Equação 6)}$$

3.3 Análise das Redes de Petri

Vários estudos sobre a metodologia de mapeamento das redes foram realizados durante anos. Em alguns destes, foram definidas uma série de análises[21], as quais permitem realizar verificações das propriedades dos sistema. Estas análises estão divididas em dois grandes conjuntos, que são: as análises voltadas para as propriedades estruturais e aquelas outras voltadas para as propriedades comportamentais. As primeiras destacam-se pelo fato de serem propriedades que se baseiam apenas em sua estrutura. Já as propriedades comportamentais tratam de averiguar características das redes baseadas na marcação destas, ou seja, baseadas nos estados alcançados durante a simulação do funcionamento de uma rede de Petri. Nestas análises, o emprego da matriz de incidência é de grande importância. Ao longo da próxima seção, segue uma lista das propriedades pertencentes a cada um destes dois grupos.

3.4 Propriedades Estruturais

- **Limitação Estrutural.** Uma rede é limitada estruturalmente se ela é limitada para qualquer marcação inicial. Matematicamente falando, existem um vetor W de números

inteiros positivos (#P, ou seja, do tamanho do conjunto dos lugares) onde o produto matricial deste pela matriz de incidência é menor ou igual a zero, isto é,

$$W \times C <= 0. \qquad \textbf{(Inequação 4)}$$

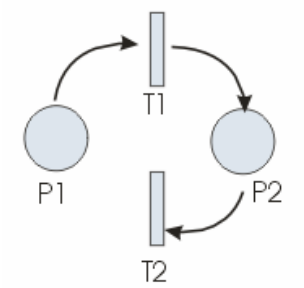


Figura 15. Limitação Estrutural

- **Conservação.** Diz-se que uma rede é estritamente conservativa quando recursos não são criados nem destruídos, isto é:

$$\Sigma M(i) = \Sigma M(k). \qquad \textbf{(Equação 7)}$$

Isto acontece sempre que #I(tj) = # O(tj), para $t_j \in T$. Isto é, sempre que a cardinalidade dos arcos de entrada de uma transição for igual a cardinalidade dos arcos de saída, esta propriedade é satisfeita. Um exemplo deste tipo de rede pode ser verificado na Figura 16. Note que o peso dos arcos ligando os lugares as transições (e vice-versa) tem peso '1'.

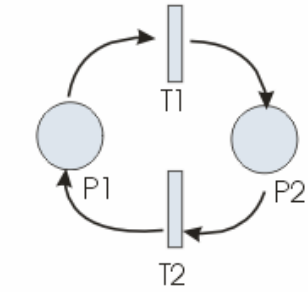


Figura 16. Conservação

Ainda existem os conceitos de redes conservativas, que são aquelas que podem ser transformadas em redes estritamente conservativas. Podemos associar a estas redes conservativas um vetor W de inteiros positivos, onde o somatório do produto dos pesos pelas marcações dos lugares deve ser constante. As redes parcialmente conservativas, por outro lado, são aquelas que apenas alguns elementos da rede permanecem conservativos.

- **Repetividade e Consistência.** Uma rede é repetitiva quando a partir de uma marcação inicial e uma seqüência de transições disparáveis para esta marcação, todas as transições serão disparadas indefinidamente. Por outro lado, uma rede é dita consistente se para uma marcação inicial, é possível retornar a essa marcação desde que todas as transições disparem pelo menos uma vez.

$$M\,[sq > M_0, t_i \in P$$

Esta representação acima mostra que para sair da marcação M e chegar na marcação M_0 (marcação inicial), é disparada uma seqüência de transições sq. A rede da Figura 17 é consistente e repetitiva.

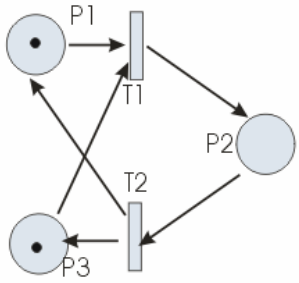


Figura 17. Repetividade e Consistência

Na Figura 17 temos que, caso a transição T_1 dispare e depois a transição T_2 também dispare, a rede volta a sua marcação inicial. Existem redes na qual para voltar a marcação, é necessário o disparo de apenas um subgrupo das transições, ou seja, nem todas transições disparam. Neses casos diz-se que a rede é parcialmente conservativa.

3.5 Propriedades Comportamentais

- **Alcançabilidade.** A análise de alcançabilidade procura saber se é possível a partir de uma marcação M_0 e de uma seqüência de transições disparáveis obter uma marcação M', ou seja, M_0 [sq > M'. Na Figura 18 temos que a marcação inicial da rede é M_0=(0,1,0,0), e a intenção é descobrir se a marcação M'=(1,0,0,0) é alcançável. Neste caso, notamos que as seqüências sq =T_2;T_3;T_4 ou sq=T_5;T_4 nos leva a marcação desejada. Por outro lado, vemos que a rede não destrói recursos de forma que nunca poderemos atingir a marcação: M''=(0,0,0,0) Entretanto, essa busca pode ser feita pela matriz de incidência da rede. A importância da alcançabilidade se concentra no fato de podermos descobrir, por exemplo, se um determinado estado indesejável é alcançável. Além disso, ao nos depararmos com redes extensivamente longas poderemos trabalhar com apenas uma subparte desta rede, ou seja, realizaremos a análise de alcançabilidade apenas para uma submarcação.

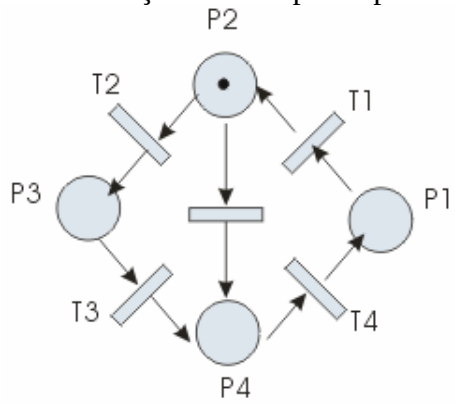


Figura 18. Análise de Alcançabilidade

- **Limitação.** A Limitação serve para verificar se existem lugares na rede que acumulam marcas infinitamente. Redes formadas pelas redes elementares do tipo atribuição são freqüentemente redes não limitadas. Ao modelar um protocolo de comunicação (com

produtor, receptor e *buffer*) poderíamos, por exemplo, saber se há em algum momento sobrecarga do *buffer*. Isto é possível se considerarmos que o *buffer* representa um lugar da rede, e as marcas representam os pacotes transmitidos. Caso o *buffer* tenha uma capacidade limitada e passe a acumular marcas infinitamente, em termos reais, isto representaria a perda de pacotes durante a transmissão.

Dizemos que um lugar p_i é k-limitado, se para todas as possíveis marcações da rede o número de marcas no lugar é inferior a k, ou seja, $M(p_i) < k$.

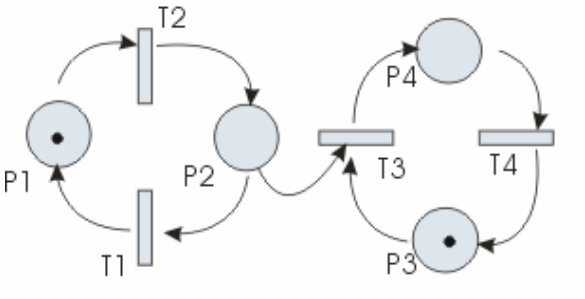


Figura 19. Limitação

Na rede da Figura 19, podemos garantir que o lugar P_1 é 1-limitado, isto é, para todos os estados os da rede, sempre teremos $M(p_1)<=1$. Da mesma forma, P_3 e P_4 também são 1-limitado.

- **Segurança.** Segurança é uma particularidade da análise de limitação, caracterizada por ter todos os lugares 1-limitado, em todos os possíveis estados da rede. De uma maneira geral, para toda $M' \in A(R, M0)$, teremos uma rede segura se $M(p_i) <=1$, $\forall\ p_i \in P$.

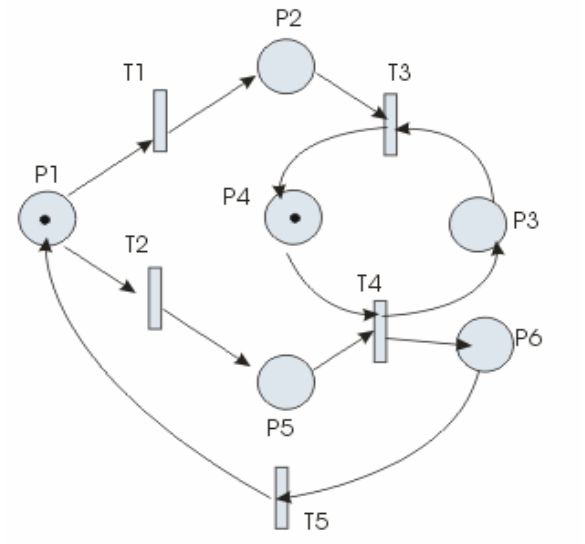
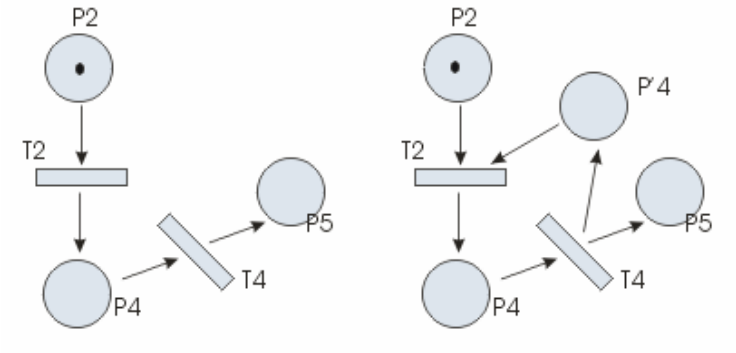


Figura 20. Segurança

Na Figura 20, temos que todos os lugares são 1-limitado, o que caracteriza a rede ser segura.

Geralmente, desde que a rede só possua arcos de peso unário, podemos transformar uma rede insegura em um rede segura através do emprego de lugares duais. De uma forma geral, se $p_k \in O(t_i)$ e $p_k \notin I(t_i)$, então se cria o lugar p'_k, onde $p'_k \in I(t_i)$ e $M(p'k)= 1$.

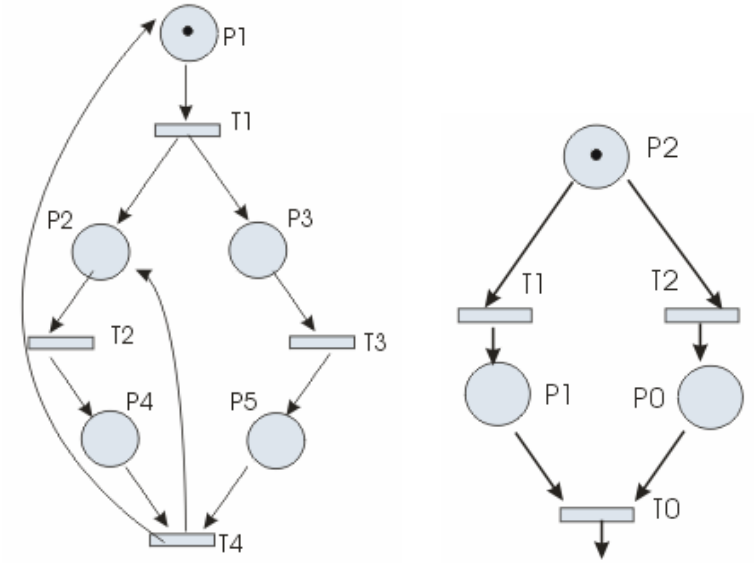


(a) (b)

Figura 21. Lugares Duais

Na rede da Figura 21(a), o lugar P_4 poderia acumular marcas através de sucessivos disparos de T_2. Entretanto, na Figura 21(b), com a introdução do lugar dual P'_4, o lugar P_4 já não poderá acumular marcas infinitamente. Isto porque T_2 só estará habilitada caso haja uma marca em P'_4, e esta marca só pode ser conseguida ao disparar T_4, a qual retirará a marca do lugar de entrada P_4.

- ***Liveness.*** A propriedade de *liveness* resume-se a checar o nível de *liveness* das transições presentes na rede. Assim, sendo $t_i \in T$, a transição t_i pode apresentar algum dos seguintes níveis: N1 – *Live* caso t_i possa ser disparada pelo menos uma vez em alguma seqüência de disparos, N2- *Live*, onde ela pode ser disparada k vezes ou N3 –*Live*, onde t_i aparece infinitamente em uma seqüência de disparos. Caso t_i seja N1 – *Live* para todas as marcações da rede, dize-se que t_i é N4-*Live*. Por fim, uma transição pode estar morta, ou seja, para toda marcação M', t_i permanece desabilitada.



(a) (b)

Figura 22. (a) Rede Live; (b) Rede não Live

Na Figura 22 (a) podemos garantir que todas as transições são N1-*Live*, ou seja, podem ser disparadas em alguma seqüência de disparos, como, por exemplo, sq=T_1; T_2; T_3; T_4. Entretanto, na Figura 22 (b), temos que a transição T_0 é uma transição morta, isto é, ela não pode disparar em hipótese alguma.

- **Cobertura.** A propriedade de cobertura está relacionada com a marcação da rede. Dizemos que uma marcação M'(p_i) ∀ pi ∈ P está passível de cobertura, quando existe alguma outra marcação M''(p_i) obtida através de uma seqüência de disparos a partir de M' e, aonde M''(p_i) >= M'(p_i), ou seja, para qualquer lugar da rede, o número de marcas na marcação M''(p_i) é sempre maior ou igual ao número de marcas presente nos lugares de M'(p_i). O conceito de cobertura está ligado aos conceitos de outras duas propriedades: alcançabilidade e *liveness*. Da mesma forma que a alcançabilidade, podemos verificar se uma submarcação (formada apenas por um conjunto dos lugares da rede) é passível de cobertura.

- **Persistência.** A propriedade de persistência é de grande importância principalmente em sistemas paralelos ou até mesmo para quaisquer circuitos com processos assíncronos. Isto porque, a persistência procura por conflitos dinâmicos na rede, ou seja, procura por pares de transição onde o disparo de uma delas desabilita a outra. Isto caracteriza sua importância nestes sistemas, os quais precisam manter atividades paralelas em execução.

- **Reversibilidade.** A propriedade de reversibilidade averigua se é possível, a partir de uma seqüência de disparos, retornar a marcação inicial da rede, ou até mesmo a uma outra marcação acessível. O fato de a rede poder voltar a uma marcação (sem ser a marcação inicial) diferencia esta propriedade da Repetividade.

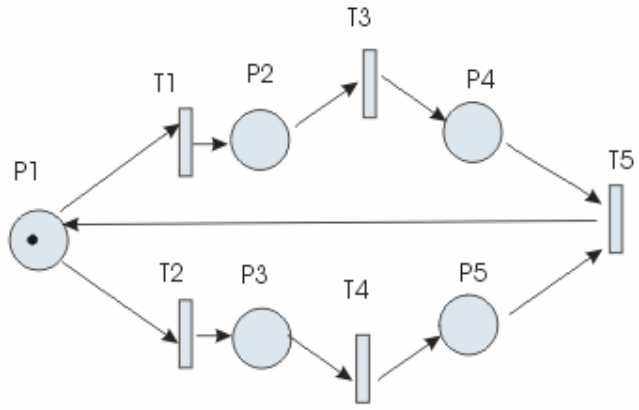


Figura 23. Reversibilidade

A rede de Petri da Figura 23 é reversível. Através das seguintes seqüências de disparos sq=T_2;T_4;T_5 sq= T_1;T_3;T_5, entre outras é possível retornar a marcação inicial. Considere também a marcação M'(0,0,0,1,0). Através da seguinte seqüência de disparos sq= T_5;T_1;T_3; é possível retornar a marcação M'. Quando uma rede não apresenta a propriedade de limitação, ela, provavelmente terá lugares que acumulam marcas. Isso faz com que a rede deixe de ser reversível.

- **Justiça.** O conceito de justiça está relacionado a quantidade de disparos entre duas transições. Dizemos que duas transições (t_n, t_m) são justas quando a quantidade de disparos de uma transição está limitada pela quantidade de disparos da outra. Ou seja, uma transição não pode ficar disparando infinitamente, enquanto a outra não dispara.

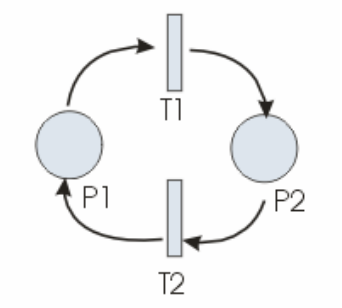


Figura 24. Justiça

Nesta rede, as transições T_1 e T_2 são justas. Note que existe sempre uma relação de um para um no disparo das transições, ou seja, a cada disparo de T_1 haverá um disparo de T_2.

As ferramentas mais utilizadas no meio acadêmico para estas análises são a *INA* e a *CPNTools*. Entretanto, quando estamos analisando redes coloridas, o leque de análises fica reduzido, uma vez que estas redes, por possuírem tipos, variáveis e funções, tornam-se mais complexas.

3.6 Extensões das Redes de Petri

Atualmente, existem diversos tipos, ou extensões, das redes de Petri. Estes tipos foram criados devido a necessidades surgidas pela utilização das redes mais simples. Essas necessidades eram, dentre outras, a busca de uma rede com um poder computacional maior, através das quais, sistemas complexos que envolvem grande quantidade e manipulação dos dados pudessem ser facilmente representados numa especificação de alto nível.

Desta forma, surgiram as redes hierárquicas [22], a qual provê uma melhor estruturação do sistema devido a utilização de páginas, as redes de Petri Temporizadas[22], a qual foi uma solução para a modelagem de sistemas que trabalhavam em tempo real, para a avaliação de desempenho e em problemas de escalonamento. Existem ainda subdivisões dentro das redes temporizadas, uma vez que o tempo pode ser associado tanto às transições quanto aos lugares. Ainda existem extensões das redes, as quais apresentam diferenças visuais. Um exemplo típico são as redes com arco inibidor[22].

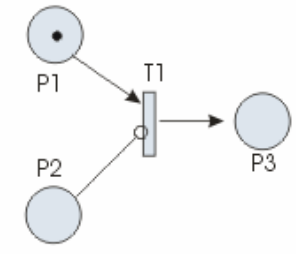


Figura 25. Rede com arco inibidor

Um arco inibidor é representado graficamente por um seta, sendo que há uma circunferência em sua extremidade(Figura 25). Estes tipos de arcos servem para realizar o teste a zero, isto é, verificar a presença de marcas em um determinado lugar. Isto porque, diferente do conceito padrão de habilitação, uma transição com um arco inibidor de entrada estará habilitada se o lugar de entrada desta transição não detiver nenhuma marca. Na Figura 25 a habilitação da transição t acontece quando: $M(p1) \geq (I(p1,t))$ e quando $M(p2) = 0$. Entretanto, nosso projeto

está voltado para a modelagem de sistemas embarcados que são sistemas com atividades concorrentes e com alto nível de manipulação de dados. Assim, para a criação de um modelo formal em alto nível, utilizamos uma extensão das redes de Petri conhecidas como redes de Petri coloridas. Estas são redes hierárquicas, e possuem tokens “tipados” que podem ser acessados por uma linguagem de programação funcional (CPN ML), o que facilita a manipulação de dados na rede. Estas e outras características podem ser vistas nas seções seguintes, nas quais os conceitos e a utilização destas redes são definidos.

3.7 Redes de Petri Coloridas

Desde o surgimento das redes de Petri, no início da década de 60, elas rapidamente tornaram-se populares nos campos acadêmico e industrial. Este sucesso deve-se, sobretudo à sua extensa aplicabilidade na modelagem e análise de sistemas com atividades concorrentes e não-determinísticas, como os sistemas de comunicação. Durante a década de 70 começaram a ser apontadas algumas limitações das redes de Petri[23]. Primeiro, verificou-se que, por não suportarem o conceito de dados, os modelos gerados eram extremamente grandes e complexos. Isso pode ser explicado por que as manipulações dos dados eram expressas diretamente na estrutura da rede através de lugares e transições. Por outro lado, também não havia suporte a características hierárquicas, ou seja, era impossível modelar um sistema através de vários módulos, conectados entre si através de interfaces. Na década seguinte, as pesquisas realizadas sobre as redes de Petri apresentaram ao mundo novas extensões conhecidas como redes de alto nível, que resolviam os problemas citados anteriormente. Dentre os novos membros da família das redes de alto nível, destacavam-se as redes de Petri coloridas [4], que são uma rede hierárquica e suporta a estruturação e manipulação de dados do sistema. O conceito destas redes será explicado ao longo destas seções através de um exemplo. Em seguida serão mostrados os motivos que nos levaram a adotar as redes coloridas, também conhecidas como CP*Nets*, para representar os cenários da LSC, seguindo os passos descritos na metodologia referenciada em [18].

Um modelo de uma CP*Net* apresenta, além das transições, arcos e lugares, páginas. Cada página possui uma interface através da qual pode-se estabelecer a comunicação da rede com as outras páginas. Graficamente, o conceito de página auxilia no entendimento de como um processo dentro de um sistema maior executa durante o tempo. As redes coloridas apresentam uma semântica e uma sintaxe bem definida, bem como uma representação matemática formal, o que fornece as bases para construir ferramentas e linguagens poderosas.

A simulação de uma rede colorida pode ocorrer de duas formas distintas: através da simulação interativa e da simulação automática. O primeiro tipo é caracterizado pela constante interação do usuário durante a simulação. O usuário garante parâmetro de entrada à rede e analisa como se dá o seu fluxo de execução, podendo verificar a cada passo o que está acontecendo internamente ao sistema. Uma simulação automática, por outro lado, caracteriza-se pelo fato do usuário não interagir com a rede. Ele apresenta as entradas, realiza a simulação de uma forma rápida, e checa a saída.

3.8 Modelando o protocolo Produtor/Consumidor com redes Coloridas

O exemplo a ser descrito nesta seção trata de um sistema do tipo Produtor/Consumidor, através do qual mostraremos a importância da manipulação de dados na rede e, assim, descreveremos as estruturas constituintes de uma CP*Net* e suas diferenças com as redes de Petri básicas. Este sistema é formado basicamente por um produtor, um consumidor e um canal de transmissão bidirecional. A funcionalidade do nosso produtor é agrupar uma certa quantidade de dados e enviá-los através do canal para um receptor. Entretanto, nosso canal é passível de falhas, de forma que não há garantia prévia de que a mensagem enviada seja recebida. Com isso, nosso produtor foi programado de forma a repetir sempre o envio de um pacote de dados até que uma mensagem de reconhecimento proveniente do receptor seja recebida, indicando que o pacote enviado foi recebido.

Como em sistemas de envio e recebimento de dados, a ligação entre um pacote de dados gerado pelo produtor, e uma mensagem de reconhecimento criada por este último, é estabelecida através de um número seqüencial. O seu funcionamento dá-se da seguinte forma: além dos próprios dados, o pacote gerado por um produtor também leva consigo um número identificador (seqüencial). Quando o pacote é recebido do outro lado do canal, os dados são enviados para o consumidor e o número do pacote é utilizado para criar o *Ack* de reconhecimento. O número desse *Akc* será definido por:

SeqAck = Seqüencial + 1;

indicando que o receptor está esperando pelo próximo pacote, ou seja, aquele com número seqüencial de valor SeqAck.

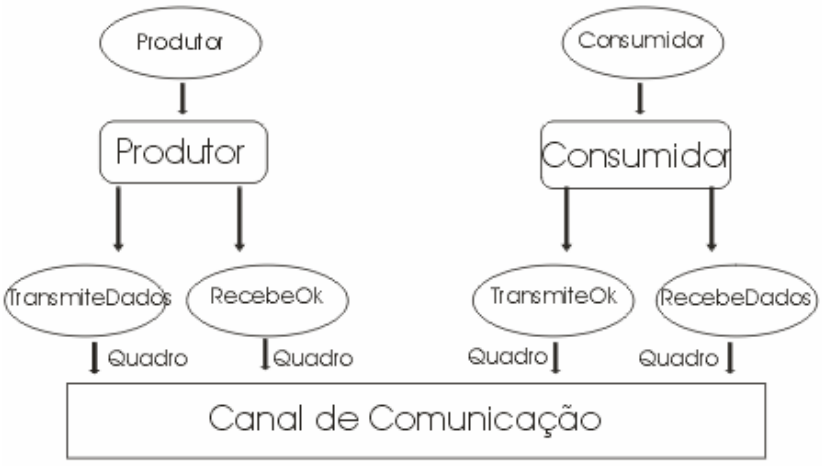


Figura 26. Protocolo Produtor/Consumidor

Na Figura 26, pode-se visualizar a representação do exemplo que estamos trabalhando. Os três componentes acima estão ilustrados em conjunto com os caminhos dos pacotes, que podem ocorrer durante a sua transmissão. O estado de uma rede é designado através da marcação dos lugares presentes. Todavia, como será visto adiante, a marcação de uma rede colorida apresenta novas características não presenciadas nas redes mais simples.

Na Figura 27, apresenta-se a rede necessária para modelar o elemento produtor do exemplo. Esta figura tem a função de representar toda a estrutura de um produtor, neste caso, representando seus processos de envio de pacote de dados e de identificação de mensagens de reconhecimento. Como nas redes originais, o estado de uma rede colorida é fornecido através da identificação da marcação dos seus lugares. Entretanto essa marcação não está apenas baseada em

simples marcas associadas aos lugares. As redes coloridas adotam o conceito de tipos ou *colour sets*. Estes são declarados em uma linguagem chamada de CPN ML[16], que é uma extensão da linguagem ML[17].

A cada lugar da rede está associado um tipo, indicando os tipos de dados que o lugar pode possuir. Esses tipos são similares aos tipos de uma linguagem de programação e de forma semelhante, as redes coloridas adotam também o conceito de tipos compostos, ou seja, um tipo formado pela junção de estruturas de dados simples como, por exemplo: *record, list* e *union.*

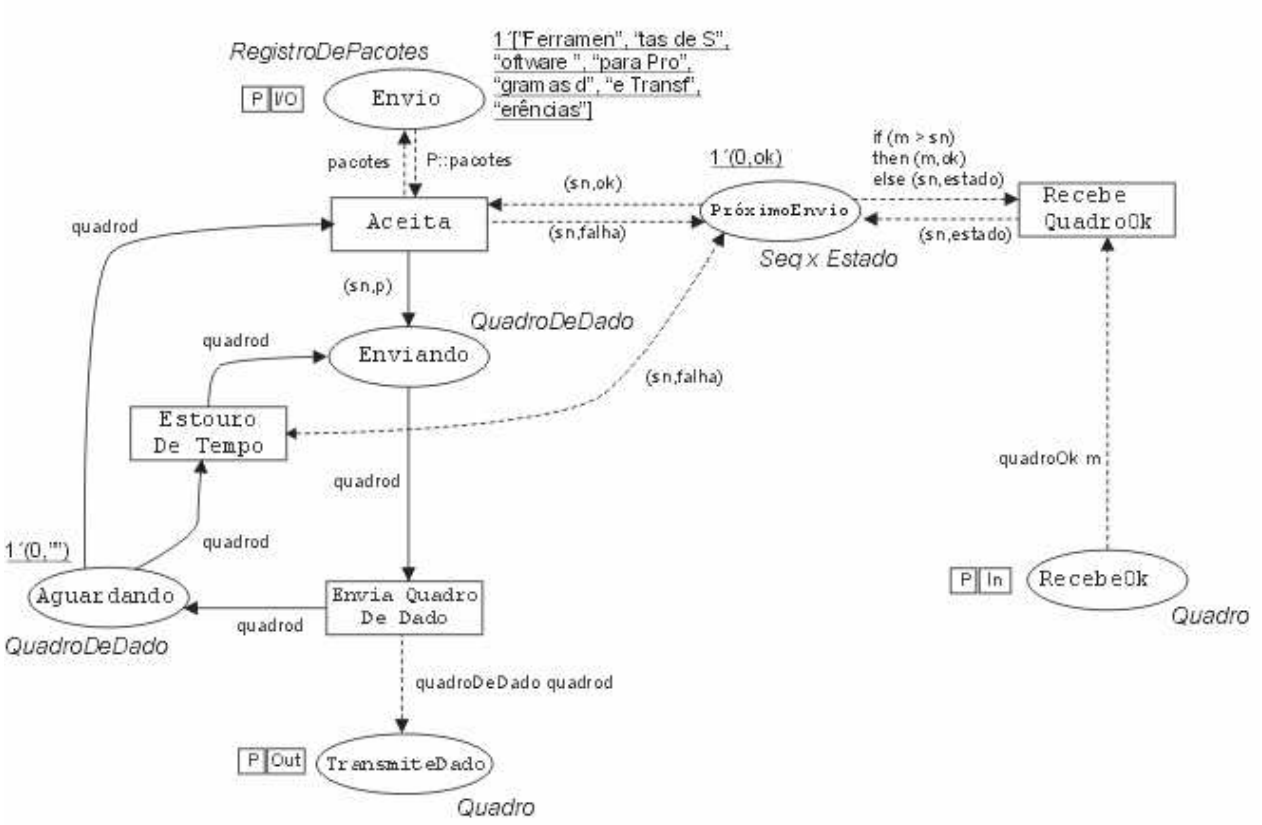


Figura 27. Representação do Produtor

As redes coloridas suportam basicamente três tipos básicos de dados: *string, int* e *enumerated.* O tipo *boolean* geralmente é descrito através de um tipo *enumerated* com dois valores: *True* ou *False.*

Na rede da Figura 27 há seis lugares (representado pelas elipses) para modelar a estrutura de uma entidade produtora de recursos:

O lugar `Envio` na parte superior da Figura representa um registro onde estão armazenados os dados a serem enviados para o consumidor. Na outra extremidade da Figura, notam-se dois outros lugares: `TransmiteDado` e `RecebeOk`. Ambos atuam na extremidade da conexão do produtor ao canal de comunicação. Estes representam o estado de transmissão dos dados e de recepção da mensagem de reconhecimento, respectivamente. O lugar `PróximoEnvio`, por outro lado, serve para mostrar a fase em que o produtor analisa a mensagem de *Ok*, que acaba de chegar (ou seja, seu número de seqüência) e decide se continua a enviar o mesmo pacote ou se já pode enviar o próximo pacote de dados. Por fim, os dois últimos lugares `Enviando` e `Aguardando` representam o estado do produtor. De uma forma geral, ou o produtor está enviando um pacote de dados ou está apenas aguardando uma reação do sistema depois de ter enviado o pacote.

Por se tratar de uma rede colorida, existem tipos (cores) associadas a cada lugar da rede. Estes tipos podem ser visto na Figura 28. De uma forma geral, podemos identificar alguns tipos

de dados: `Pacotes` representa o próprio pacote de dados a ser enviado, representado neste caso por uma string, `Registro de Pacotes`, o qual é um tipo composto, representando a lista de todos os pacotes a serem enviados, `Seq`, um inteiro indicando o número seqüencial de um pacote, `Estado` para o estado do pacote, isto é, se ele foi recebido ou não (por apresentar dois valores possíveis, este tipo é identificado por um tipo enumerado). Graficamente, os tipos são escritos em itálico e colocados próximos aos lugares a que estão associados.

```
1 color Pacotes = string;
2 color RegistroDePacotes = list Pacotes;
3 color Seq = int;
4 color Estado = with Ok| NaoOk
5 color SeqXEstado = product Seq * Estado;
6 color QuadroDeDados = product Seq * Pacotes;
7 color QuadroOk = Seq;
8 color Quadro = union quadroDeDado: QuadrodeDados + quadroOk: QuadroOk;
```

Figura 28. Colour Set do Produtor

Já foi descrito anteriormente o conceito da marcação de uma rede. Enquanto que em uma rede básica, as marcas são iguais pra cada lugar, o mesmo não ocorre nas redes coloridas. Um lugar pode apresentar um número de marcas de um determinado valor (*color*) identificado pelo tipo a qual está associado. Esta convenção é que define a marcação deste lugar. Voltando à representação do produtor do protocolo, pode-se tomar o lugar `PróximoEnvio` como um pequeno estudo de caso. Este lugar é do tipo SeqxEstado, o qual é um produto de um inteiro por um tipo enumerado, sendo que o inteiro representa um número seqüencial e o tipo enumerado representa o estado do lugar. Este lugar inicialmente apresenta uma única marca, representada por:

1’(0,ok)

onde:

1 : representa a quantidade de marcas;
0 : representa o primeiro elemento do par, ou seja, o seqüencial;
ok : representa o segundo elemento do par, ou seja, o estado do lugar.

Caso quiséssemos representar uma marcação formada por mais de um tipo de dado, poderíamos utilizar o símbolo ‘+’, como, por exemplo:

1’(0,ok) + 2’(1,falha)

Além de uma marca com seqüencial zero e estado ok teríamos também outras duas marcas com seqüencial um e estado falha.

Para esse mesmo lugar, o tipo 1’(0,ok) representa a marcação inicial da rede para o `PróximoEnvio`. De forma padronizada, a marcação inicial é representada graficamente próximo ao lugar na parte superior, do lado esquerdo ou direito do lugar. Os lugares que não apresentarem essa informação, certamente não apresentam marcação inicial. Assim, temos também para o lugar `Envio` a seguinte marcação inicial:

1’,[“Ferramen”, “tas de S”, “oftware ”, “para Pro”, “gramas d”, “e Transf”, “erências”]

O lugar `Envio` apresenta uma única lista de pacotes (tipo) sendo os pacotes: “Ferramen”, “tas de S”, “oftware ”, “para Pro”, “gramas d”, “e Transf”, “erências” que provavelmente estão aguardando para serem transmitidos pelo canal de dados.

As ações e o fluxo de execução de uma rede são determinadas pelas transições. Como nos lugares, para fins didáticos, os rótulos das transições estão dentro dos objetos gráficos representantes. Na rede existem quatro transições: - `Aceita` representando a ação de um próximo pacote ser aceito para o envio, - `Envia Quadro de Dados` que é a ação de enviar um pacote de dados pelo canal de transmissão, - `Recebe Ok` , que é ação de receber a mensagem de reconhecimento e por fim, - `Estouro de Tempo`, o qual representa a ação de mudança do estado do produtor, isto é, ele sai do estado de aguardando e passa para o estado de enviando. Isso deve ocorrer devido ao atraso de uma mensagem de reconhecimento.

Os arcos, como nas redes de Petri básicas, servem para conectar os lugares às transições. São de fundamental importância no estabelecimento do comportamento dinâmico da rede, pois são quem dita quais marcas serão retiradas dos lugares de entrada e quais serão postas nos lugares de saída. Próximos aos arcos estão as chamadas expressões dos arcos. Estas definem quais e a quantidade de marcas que estão sendo criadas e/ou destruídas. As redes coloridas representam sistemas concorrentes de modo fiel, por isso é comum encontrar em uma rede colorida o disparo em paralelo de várias transições.

Devido ao emprego de um *colour set*, o comportamento dinâmico de uma rede comporta-se de uma forma um pouco diferente com relação às redes básicas. Primeiramente é necessário saber que uma rede colorida também utiliza variáveis para a sua execução. Estas variáveis estão presentes nas expressões dos arcos, como também nas expressões booleanas que são associadas às transições, as quais são também conhecidas como expressões guardas. Para a rede colorida em questão, temos na Figura 29 a declaração das variáveis presentes na rede:

```
1 var p: Pacotes;
2 var pacotes: RegistroDePacotes;
3 var sn, rn: Seq;
4 var estado: Estado;
5 var quadrod: QuadroDeDados;
```

Figura 29. Variáveis da Rede

A estas variáveis são atribuídos valores de acordo com os tipos definidos para cada lugar. Essa atribuição é feita através do operador ←. Tomemos como exemplo a transição `Aceita`, a qual possui quatro variáveis: `p` do tipo Pacote, `pacotes` do tipo RegistroDePacotes, `sn` do tipo Seq, e `quadrod` do tipo QuadroDeDados. A partir dessas variáveis pode-se criar uma lista de atribuição, a qual atribui-se um valor a cada uma destas.

```
1 p<-- "Ferramen"
2 pacotes<-- ["tas de S","oftware ","para Pro","gramas d","e Transf", "erências"]
3 sn<-- 0
4 quadrod<-- (0,"")
```

Figura 30. Atribuição de valores às variáveis das expressões

De forma geral, a atribuição de uma transição é descrita na forma $<v_1=d_1; v_2=d_2; v_3=d_3; ... v_4=d_4>$, onde v_i para i =1,2,3..., n é uma variável e d_i, é o valor atribuído a vi.

Para que uma transição esteja habilitada, é necessário que cada lugar de entrada possua as marcações dos tipos estabelecidas nas expressões dos arcos de entrada.

No caso da transição `Aceita`, temos que para que ela esteja habilitada é necessário uma marca de valor (0,"") no lugar `Aguardando`, uma marca com valor [“Ferramen”, “tas de S”,

“oftware ”, “para Pro”, “gramas d”, “e Transf”, “erências”] no lugar `Envio`, e uma marca com valor (0,ok) no lugar `PróximoEnvio`.

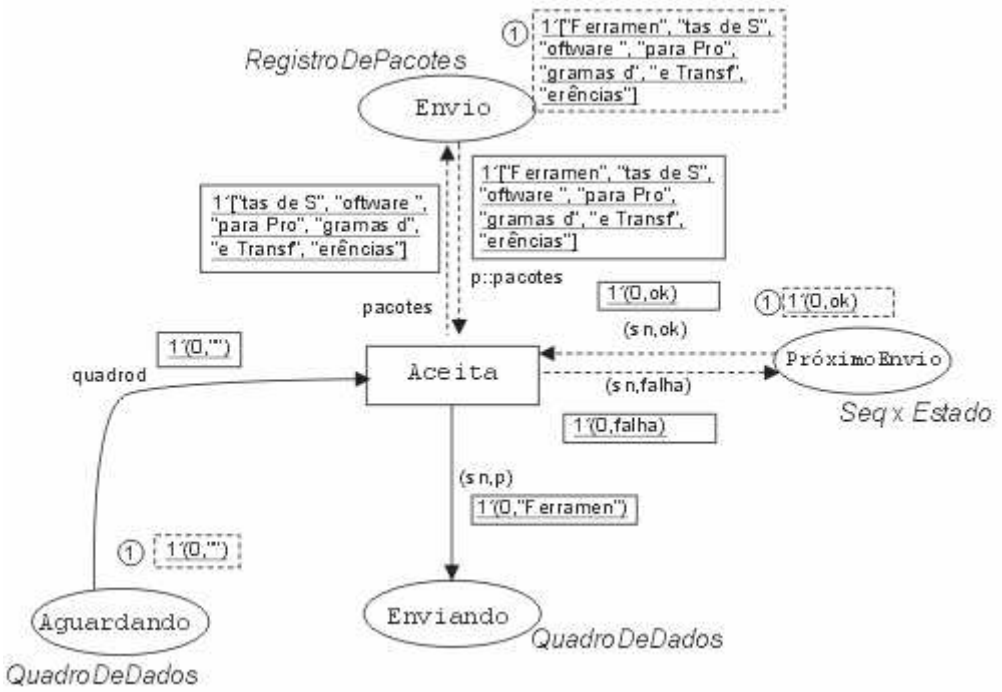


Figura 31. Rede com as informações dos lugares, arcos e transições

A habilitação de uma transição não a obriga a disparar. Através do disparo da transição `Aceita`, as expressões dos arcos de entrada são avaliadas e as marcas são removidas dos lugares de entrada. Da mesma forma, as expressões dos arcos de saída também são avaliadas, e conseqüentemente marcas são adicionadas aos lugares de saída. As expressões dos arcos são garantidas devido às atribuições realizadas como mostradas na Figura 30. Na Figura 32, por outro lado, está ilustrado o resultado do disparo da transição `Aceita`.

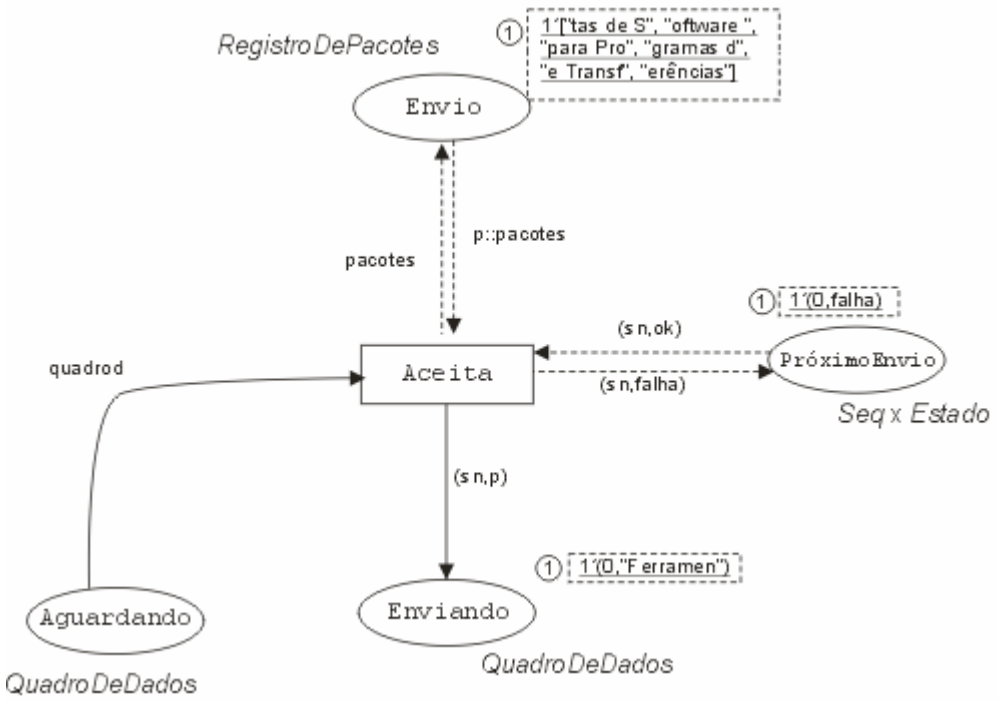


Figura 32. Resultado do disparo da transição Aceita

De uma forma geral, o pacote ‘Ferramen’, que fazia parte da lista de pacotes do lugar `Envio`, está sendo enviado pelo canal. Ele não faz mais parte desta lista. O próximo pacote a ser enviado torna-se então o pacote ‘tas de S’. O estado do lugar `PróximoEnvio` foi alterado para falha. Isto ocorreu porque para o pacote de número seqüencial 0, que foi enviado, foi recebida uma mensagem de reconhecimento com valor ‘0’, indicando que o consumidor ainda não recebeu o pacote ‘0’. Por outro lado, o estado do produtor muda de Aguardando para Enviando. Isto pode ser evidenciado pelo fluxo das marcas com relação a estes dois lugares. Observando ambos os lugares, nota-se que a marca do lugar `Aguardando` foi removida e foi adicionada uma nova marca ao lugar `Enviando`.

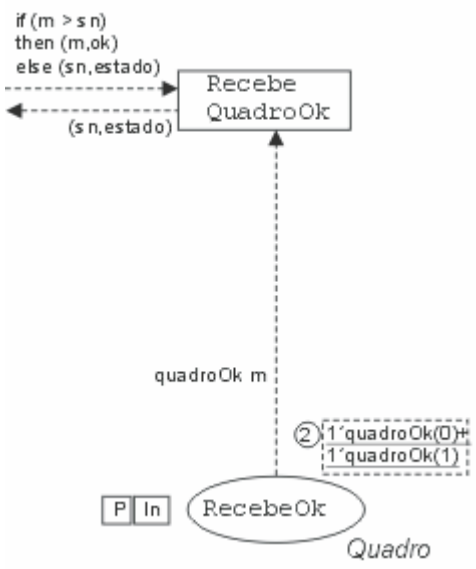


Figura 33. Modelo referente ao recebimento da mensagem de reconhecimento

Com relação ao `RecebeOk` (Figura 33), o disparo da transição `Recebe Quadro de Dados` retira um marca do `RecebeOk`, a qual é um quadroOk, e em seguida, compara o valor seqüencial deste quadroOk rn com o valor seqüencial sn do quadroDeDados que está sendo enviado. Caso o valor de rn seja superior, isso indica que o pacote que está sendo enviado, de seqüencial sn, foi recebido, e o estado será modificado para (rn, ok), isto é, o próximo pacote a ser enviado será o com número seqüencial rn.

Através de uma seqüência de disparos a rede colorida vai passando por vários estados, isto é, por várias marcações diferentes. Na Figura 34, por exemplo, o lugar `RecebeOk` possui duas marcas, representado por: 1’quadroOk(0) + 1’quadroOk(1). Essa marcação indica que o consumidor recebeu o pacote de dados de seqüencial ‘0’ e espera o pacote de seqüencial ‘1’. Uma rede pode apresentar uma seqüência de disparos finita, ou seja, há um número limitado de marcações, ou uma seqüência de disparos infinita, isto é, há uma indeterminação no fluxo do sistema já que o mesmo pode atingir vários estados.

A representação da transição `Estouro de Tempo`, em destaque na Figura 34, é apenas de cunho didático, para facilitar o entendimento dos atrasos que podem ocorrer num canal de comunicação. Entretanto, as redes coloridas suportam o conceito de temporização, de forma que se pode estabelecer um intervalo de tempo para o disparo da transição `Estouro de Tempo`, e devido a um canal lento, essa transição será a responsável por mudar o estado da rede

de Aguardando para Enviando, informando que nenhuma mensagem de reconhecimento do pacote foi retornada, e que tal pacote deve ser reenviado.

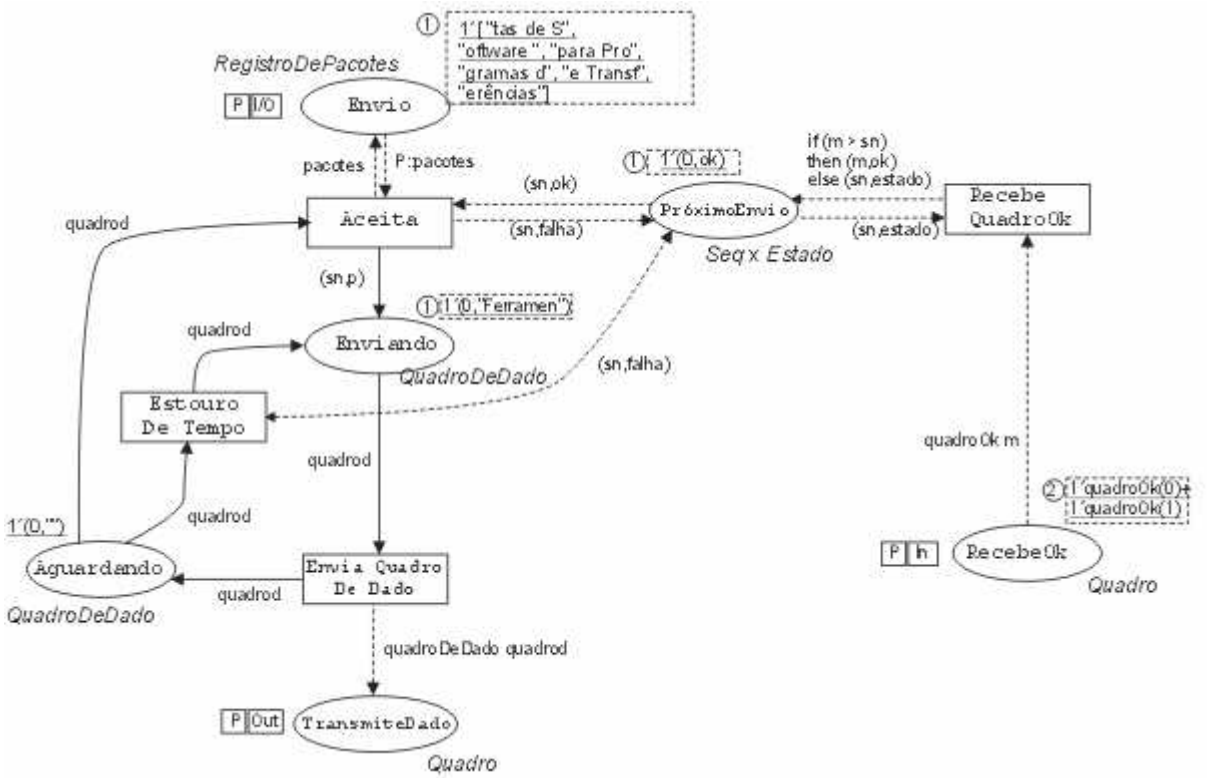


Figura 34. Resultado do disparo de um seqüência de transições

Ao término deste Capítulo, queremos enfatizar as características das redes coloridas, mostrando porque a escolhemos como base para a criação de um modelo formal em alto nível dos sistemas e como parte do nosso compilador de modelos formais. Vem sendo evidenciado durante este projeto que as redes de Petri, além de serem matematicamente fundamentadas, oferecem uma representação gráfica de fácil assimilação o que auxilia qualquer gerente de projetos na análise do sistema e na tomada de decisões a respeito da arquitetura do mesmo. Além disso, as redes oferecem uma alternativa à simulação para realizar análise de sistemas, onde propriedades podem ser testadas e o desempenho do sistema pode ser estudado. As redes coloridas aumentaram este leque de qualidades das redes. Possuem uma semântica e uma sintaxe bem definida, e também suportam manipulação de dados. Além disso, são redes hierárquicas, o que facilita ainda mais a construção de modelos mais legíveis. Não se pode deixar de mencionar também o fato das redes coloridas conseguirem mesclar o conceito de manipulação de controle e sincronismo com o conceito de manipulação de dados, o que ajuda na construção de modelos mais fiéis e menores.

Capítulo 4

Metodologia de Mapeamento

O objetivo deste capítulo é descrever como ocorreu o desenvolvimento do compilador, através do qual, os cenários LSC representados por arquivos do tipo XML serão mapeados em um outro arquivo XML, porém representando uma rede de Petri Colorida. Também será enfatizado as ferramentas e o material de apoio utilizado, bem como os problemas que surgiram durante a pesquisa. Também será mostrada a metodologia adotada para o desenvolvimento deste trabalho.

4.1 CPN Tools

CPN Tools [15] é uma ferramenta computacional para análise, simulação e edição das redes coloridas[4]. Essa ferramenta é mantida pelo *CPN Group* da Universidade de Aarhus, Dinamarca. A ferramenta *CPN tools* é considerada uma evolução da *Design/CPN*[24]. Dentre seus principais pontos, destacam-se o novo estilo de menu mais amigável e a utilização de um simulador *CPN* mais rápido do que o *Desing/CPN*. A primeira versão foi lançada em abril de 2000, sendo que, atualmente, a ferramenta está na versão 2.0.0.

Durante o desenvolvimento do projeto da *CPN tools*, seus idealistas estavam preocupados em definir uma interface poderosa e, ao mesmo tempo, simples de se trabalhar. Dessa forma, os projetistas não estavam mais interessados em utilizar simples técnicas de barra de menus, *pop ups* e mecanismo de *drag and drog*. Ou seja, os mecanismos de *WIMP interfaces* (correspondem a simples interação com menus, caixas de diálogos, janelas)[25]. Estes sabiam que programas mais complexos necessitavam de ferramentas com interfaces mais poderosas. Isto os levou a adotar novas práticas que estavam começando a entrar na área de ferramentas acadêmicas, como mecanismos de *toolsglasses*, *marking menus*, e, até mesmo, o conceito de páginas[25].

A ferramenta *CPN tools* foi construída em cima de três princípios de projeto:

- *reification*, que significa que todos os objetos (mesmo os abstratos) serão tratados como entidades físicas reais;
- polimorfismo, o que garante que comandos sejam aplicados a tipos pertencentes a uma mesma classe;
- reuso, oferece a possibilidade de reutilizar instâncias previamente criadas em outros módulos.

Na comunidade acadêmica, a ferramenta *CPN tools* vem se destacando por ser poderosa na análise, simulação e edição das redes de Petri coloridas. Por isso, resolvemos adotá-la como parte integrante do nosso projeto. Isto porque, o compilador será capaz de, a partir de arquivos do tipo XML criados pela *Play Engine*, gerar o modelo de redes coloridas em XML, que pode ser interpretado pela ferramenta *CPN tools*. Para que esse feito pudesse ser alcançado, foi necessário entender como esta ferramenta define as estruturas da rede, ou seja, lugares, transições, arcos, tipos, em seu arquivo XML. As declarações, bem como as inscrições presentes na rede (expressões dos arcos e expressões guardas) são definidas baseadas na linguagem CPN ML.
Um arquivo criado pela *CPN tools* apresenta duas *tags* principais:

- a *tag globbox*, utilizada para definir os tipos (simples e compostos), as variáveis e as funções presentes na rede;
- a *tag page*, onde são definidas as páginas e, para cada uma, seus lugares, arcos e transições. Como será descrito mais detalhadamente à frente, dentre essas estruturas, existem algumas especiais que servem para criar a interface da página, isto é, servem para implantar o mecanismo de como uma página irá se comunicar com as demais.

A Figura 35 exibe uma representação inicial de como se apresenta este arquivo da *CPN tools*.

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <!DOCTYPE workspaceElements (View Source for full doctype...)>
3 <workspaceElements>
4   <generator tool="CPN Tools" version="1.5.33" format="5" />
5 - <cpnet>
6   + <globbox>
7   + <page id="id189">
8   </cpnet>
9 </workspaceElements>
```

Figura 35. Estrutura básica de um XML da CPN Tools

4.2 JDOM

Durante o início deste projeto, nosso objetivo era realizar o mapeamento entre os arquivos do tipo .XML da Play Engine e da CPN Tools através de um *framework* de mapeamento desenvolvido em[26]. Para realizar um mapeamento, a metodologia aplicada pelo *framework* era transformar arquivos .XML em estruturas de objeto Java. O inverso também era realizável, isto é, podia-se mapear uma estrutura de objetos Java em um arquivo do tipo .XML.

Ambos eram feitos a partir de um único arquivo de mapeamento. Esse arquivo, escrito em XML, descreve o mapeamento das tags e dos atributos das mesmas em objetos Java. Utilizando este *framework*, nossa meta era mapear os arquivos XML da Play Engine em uma estruturas de objetos Java. Em seguida, realizaríamos o mapeamento dessa estrutura Java em um novo arquivo do tipo XML, este porém, representando uma rede colorida e podendo ser interpretado pela *CPN Tools*. Como a tradução realizada por este *framework* era de via dupla, isto é, ao realizar uma forma de mapeamento, a outra, por conseqüência, também era conseguida, ao terminarmos a tradução de tipos do projeto, por exemplo, o fluxo inverso de tradução estaria pronto da mesma forma.

Entretanto, durante o mapeamento foram detectados alguns problemas que estavam prejudicando o desenvolvimento do projeto. O arquivo de mapeamento apresentava constantemente problemas (o arquivo, mesmo criado de acordo com a metodologia do *framework*, não estava conseguindo mapear todos os atributos do arquivo XML de entrada), além

de não haver suporte para a utilização do *framework*. Por outro lado, verificou-se que a tradução dupla oferecida por este, não era tão simples. Tais problemas nos levaram a adotar uma nova forma de mapeamento, a qual apresentou-se mais fácil e mais rápida. Contudo, vale ressaltar que muito do que já havia sido desenvolvido com relação às estruturas de classes Java puderam ser reutilizadas.

A solução encontrada para resolver os impasses e dar continuidade ao nosso projeto foi a utilização do *Java Document Object Model* (JDOM) [27]. JDOM, que requer a presença de um *parser*, define uma *XML DOM API* construída a partir de objetos e tipos de dados em Java. A diferença básica com relação ao DOM é que este último foi definido totalmente à parte de qualquer linguagem de programação. JDOM, por outro lado, foi desenvolvido para ser utilizado especificamente em Java. Dessa forma, ele utiliza características da linguagem Java, tais como sobrecarga de métodos, *collections*, *reflection*, tornando o seu uso mais fácil para programadores em Java.

JDOM será utilizado tanto no *frontend* da ferramenta de mapeamento, para ler as *tags* e os atributos do arquivo XML de entrada, como também no *backend*, para gerar o arquivo XML final.

A ferramenta *Play Engine* representa os cenários através de dois arquivos do tipo XML. Em um destes arquivos estão definidos todos os tipos e variáveis utilizados na modelagem dos cenários, enquanto que o outro arquivo define os cenários propriamente ditos, com as mensagens, atribuições, condições e tempos utilizados. A nova proposta então é ler esses dois arquivos como entrada da ferramenta e gerar um terceiro arquivo também do tipo XML, mas que possa ser entendido pela ferramenta *CPN Tools*. Esta nova metodologia é descrita na Figura 36.

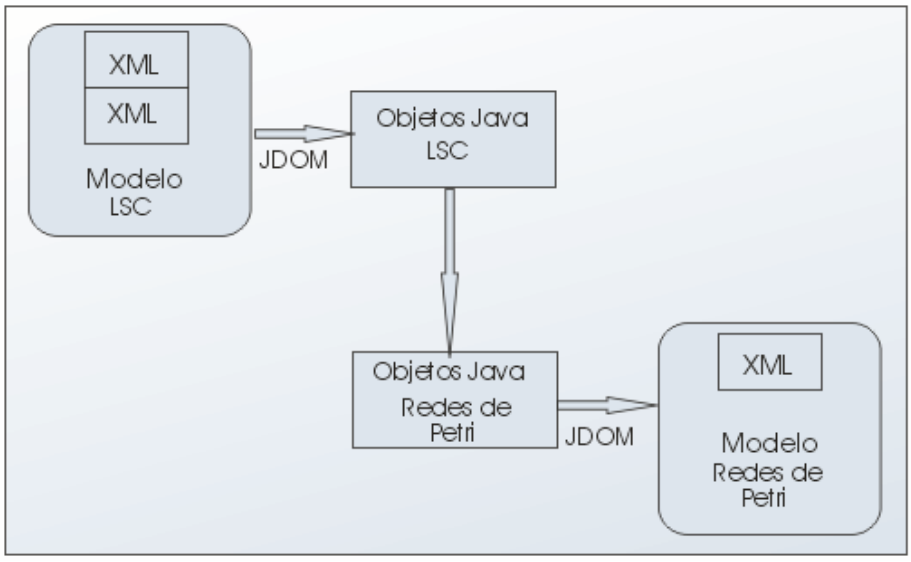


Figura 36. Metodologia de Mapeamento da Ferramenta

As classes Java estão estruturadas em dois pacotes: *engine* e *workspace*. No pacote *engine*, estão as classes relativas ao processo de leitura dos arquivos XML e à escrita do arquivo XML final. Neste pacote, também estão definidas as classes necessárias para se identificar as *tags* (e seus atributos) presentes nos arquivos para realizar o mapeamento. Por outro lado, o pacote *workspace* define as classes que representam as estruturas presentes na rede colorida, tais como as classes para instanciar os lugares, as transições, as variáveis, os tipos, entre outros.

4.3 Utilizando o JDOM

Nesta Seção, serão mostrados trechos de código ressaltando como é que se desenvolveu os processos de leitura e escrita dos arquivos do tipo XML.
A Figura 37 mostra uma dentre as dezenas de classes criadas para realizar a tradução. Como nas outras, esta classe recebe o arquivo XML para ser lido. Perceba que isto é feito através do *SAXBuilder*. É instanciado um objeto desta classe, o qual através do método '*build()*', consegue ler o arquivo de entrada e instancia, como resultado, um objeto do tipo *Document.* A partir do objeto *doc* é que serão capturadas e armazenadas as informações dos arquivos do tipo XML. Para tal, utilizamos três métodos oferecidos no pacote do JDOM: *getRootElement*(), *getChild*(<*parameter*>) e *getChildren*(). O método *getRootElement*() serve para capturamos a *tag* raiz, ou a *tag* mãe do arquivo, isto é, aquela a partir da qual são incorporados , como *tags* filhas ou atributos, as outras informações do arquivo. Em seguida, o método *getChild(<parameter>)* serve para capturar a *tag* filha com o mesmo nome que está sendo passada como parâmetro para o método. Retornado a Figura 37, o objeto "*usecases*" do tipo *Element* armazena as informações da *tag UseCases*. Esta *tag* descreve todos os cenários presentes. Para cada cenário (*tag LSC*), além de seus atributos (nome, modo...), são descritos as mensagens (*tag* Message), as condições (*tag Conditions*), as atribuições (*tag Assignmet*), e todas as outras estruturas presentes no cenário. Por fim, o método *getChildren()*, diferentemente do *getChild()*, serve para capturar não uma, mas todas as *tags* filhas associadas a um *Element.*

```
protected PEngineCondition GetCondition(File file) {

    try {
        SAXBuilder parser = new SAXBuilder();

        // passa como parâmetro o nome do arquivo XML
        Document doc = parser.build(file);

        //Captura a raiz, nesse caso a tag Specification
        Element root = doc.getRootElement();

        //Captura o filho Use Cases
        Element useCases = root.getChild("UseCases");

        //captura as instâncias UseCase
        List allUseCases = useCases.getChildren();

        //pra cada instância serão capturas suas características
        sizeOfUseCases = allUseCases.size();
```

Figura 37. Métodos para leitura de arquivo com JDOM

Assim, o retorno do método *getChildren()* é um objeto do tipo *List*. Através destes três métodos (*getRootElement*(), *getChild*(<*parameter*>) e *getChildren*()), conseguimos mapear todas as *tags* existentes no arquivo. Entretanto, as informações essenciais às quais devemos ter acesso, não são acessadas apenas desta forma. Estas informações estão, em sua maioria, presente nos atributos das *tags* do documento. Para recuperarmos estas informações, utilizamos um novo método: *getText().*

```
for (int f = 0; f < sizeOfCondSection; f++) {

    id = ((Element) allCondSection.get(f)).getChild(
            "ID").getText();
    if(((Element) allCondSection.get(f)).getChild("SectionKind") != null){
    sectionKind = ((Element) allCondSection.get(f)).getChild("SectionKind").
    getText();
    }
```

Figura 38. Utilização do método GetText()

Na Figura 38, temos que para a *tag "SectionsKind"* é recuperada a informação através do método *getText()*.

Com relação à escrita, para criarmos o documento final, utilizamos métodos para determinar o atributo de uma *tag*, *setAttribute(<parameter>)*, e também, métodos para definir uma *tag* filha associada a uma *tag* mãe, *addContent(<parameter>)*.

```
//criando a tag root
Element workspaceElements = new Element("workspaceElements");
Document myDocument = new Document(workspaceElements);

//criando os filhos

//definindo a tag generator
Element generator = new Element("generator");
generator.setAttribute(new Attribute("tool", "CPN Tools"));
generator.setAttribute(new Attribute("version", "-5.6.1"));
generator.setAttribute(new Attribute("format", "2"));
workspaceElements.addContent(generator);

//definindo a tag cpnet

Element cpnet = new Element("cpnet");
workspaceElements.addContent(cpnet);

//definindo a tag globbox
Element globbox = new Element("globbox");
Element page = new Element("page");
cpnet.addContent(globbox);
cpnet.addContent(page);
```

Figura 39. Métodos utilizados na criação do arquivo final

A primeira etapa do processo de escrita é definir a *tag root workspace*, a partir da qual desenvolveremos o resto do documento. Utilizando o método *addContent()*, estabelecemos a conexão da *tag* principal com as *tags* filhas. Para isso, basta apenas criar a *tag* filha através da classe *Element*, e associá-la à *tag* principal. Caso esta associação não seja concretizada, a tag filha não aparecerá no documento final. Alguns exemplos podem ser vistos na Figura 39.

Para definirmos os atributos utilizamos *setAttribute()*. Através deste, definimos o nome do atributo e seu valor. Observe, na Figura 39, a criação da *tag* <*generator*>. Esta tag apresenta três atributos: *tool, version e format.* Para cada atributo também é fornecido seu valor. É assim a criação destes três atributos. Em seguida, esta tag é associada a tag workspace:

workspaceElements.addContent(generator);

Na próxima seção, será descrito o processo utilizado para desenvolvermos o mapeamento entre os arquivos XML. Será detalhada a metodologia por trás do mapeamento, extraída a partir de um trabalho relacionado[18].

4.4 Mapeamento LSC2CPN

Esta seção corresponde ao ponto principal deste trabalho. Durante toda sua extensão, será detalhado, passo-a-passo, como ocorreu o desenvolvimento da ferramenta de mapeamento entre os arquivos do tipo XML da *Play Engine* e da *CPN Tools*. Saliente-se que esse trabalho foi possível graças a tese de mestrado referenciada em [18]. Nesta tese está definida todos os detalhes para realizar o mapeamento entre estas duas estruturas de arquivo, de forma que informações essenciais não fossem perdidas durante o processo de tradução. Isto é, para as estruturas presentes nos cenários (condições, atribuições, *subCharts*...) é descrito o que deve ser instanciado em termos de lugares, arcos, transições para representar essas estruturas em uma rede de Petri. Assim, essa metodologia foi seguida para implementar, nesta monografia, as redes correspondentes. Outro aspecto a ser ressaltado é que estamos trabalhando com um escopo reduzido do projeto. Dentre todas as estruturas apresentadas na *Play Engine*, serão apresentadas aquelas que precisaram ser mapeadas nesta primeira versão do projeto. As estruturas mapeadas constam na lista abaixo:

- tipos;
  - primitivos;
  - compostos;
- variáveis;
- mensagens ;
  - síncronas;
  - assíncronas;
- condições;
- atribuições;
- charts;
  - existenciais;
  - universais;
- *subcharts*;
- tempo vertical.

A descrição da tradução ocorrerá da seguinte forma: inicialmente será mostrada a definição da metodologia de mapeamento. Em seguida, explicaremos com detalhes como a tradução foi implementada em Java 1.4, utilizando a plataforma Eclipse 3.0[28].

4.4.1 Tipos e Variáveis

Como já descrito anteriormente, os tipos primitivos presentes nos cenários de modelos podem ser:

- tipo Enumerado, responsáveis por definir um grupo de valores, como por exemplo, o tipo Estações = {Verão, Inverno, Outono, Primavera};
- tipo Discreto, o qual é definido através de três parâmetros, que são, o valor mínimo, o valor máximo e o valor da diferença entre dois itens consecutivos. Por exemplo, um tipo

bit, possui valor mínimo ‘0’, valor máximo ‘7’, com uma diferença de valores consecutivos ‘1’;

- tipo *String*, definido por um tamanho máximo;

Abaixo, segue os passos para realizar um mapeamento de um especificação LSC em um tipo na CPN Tools:

- para mapear um tipo enumerado, deve-se criar um tipo CPN declarado da seguinte forma: *color enum-name = with id1|...|idn*, onde *color* e *with* são palavras reservadas da linguagem CPN ML, *enum-name* é o nome do tipo e *id1|id2|...|idn* são os itens do conjunto enumerado;
- para mapear um tipo discreto, por sua vez, deve-se criar um tipo CPN como *color type-name = int with min..max* ou *color type-name = real with min..max*, onde *type-name* é o nome do tipo, *int* e *real* são tipos primitivos da CPN ML.
- por fim, para mapear um tipo *String* definiremos *color type-name = string*, que, da mesma forma que no tipo discreto, *type-name* é o nome do tipo e *string* também é um tipo primitivo da CPN ML.

Após esta fase de identificação, passa-se realmente à implementação. Sabendo das informações essenciais para a construção das estruturas, passamos à fase onde o arquivo do LSC é lido. Por questões de simplificação, a idéia é que esse projeto inicial tenha apenas uma única página, sendo que, em projetos futuros, isto pode ser reformulado. Para os tipos é necessário identificar as *tags* que representam o *id* do tipo e o próprio tipo realmente utilizado. Depois que é feita essa busca e armazenagem das informações, instanciamos os tipos, como mostrado na Figura 40. Dessa forma, essa figura representa a instanciação das cores da rede, após ter seus respectivos atributos armazenados, como na Figura 38.

```
//definindo o conjunto das tags filhas color
Hashtable typesSet = new Hashtable();
typesSet = (newTypes.getTypesToTranslate(file)).getType();

Enumeration enumType = typesSet.elements();
Hashtable newColorSet = new Hashtable();

//instancia objetos das classes da CPNtools
while (enumType.hasMoreElements()) {
    PEngineTypeInstance tempType = (PEngineTypeInstance) enumType
            .nextElement();
    //instancia as cores
    CPNColor tempColor = new CPNColor(tempType.getId(), tempType
            .getKind());

    newColorSet.put(tempType.getId(), tempColor);
}
tagGlobbox.setColor(newColorSet);
```

Figura 40. Instanciando os tipos

```
//definindo o conjunto das tags filhas var (Symbol Table)
Hashtable varSet = new Hashtable();
varSet = (newVariables.getVariables(fileSpec).getVariable());

Enumeration enumVariable = varSet.elements();
Hashtable newVarSet = new Hashtable();

//instancia objetos das classes da CPNtools
while (enumVariable.hasMoreElements()) {
    PEngineVariableInstance tempVariable = (PEngineVariableInstance)
    enumVariable.nextElement();
    //instancia as variáveis

    CPNVar tempVar = new CPNVar(tempVariable.getStrName(), tempVariable
            .getTypeId(), tempVariable.getStrValue());
    newVarSet.put(tempVariable.getId(), tempVar);

}
tagGlobbox.setVar(newVarSet);
```

Figura 41. Instanciando as variáveis da Symbol Table

Para os tipos compostos é necessário identificar dentro do XML os tipos primitivos que formarão o tipo maior. Assim, o tipo composto será formado da seguinte forma : *type-Name* x *type-Name* x *type-Name*..., para quantos tipos primitivos existirem.

Com relação às variáveis, o LSC define em um de seus arquivos XML uma *symbol table.* Nesta, é que são definidas todas as variáveis usadas nos cenários. A partir das *tags* são recuperadas essencialmente três informações para instanciar as variáveis do modelo CPN: o tipo da variável, o nome, e seu valor. A Figura 41 apresenta este processo de instanciação.

Depois deste processo, é necessário criar o XML final. Para isto, baseados no documento *CPN DTD* definido no Apêndice A, definimos como cada estrutura deveria ser criada.

4.4.2 Condições

As condições (quentes e frias) estabelecem pontos de sincronização dentro de um cenário. Desta forma, para cada ponto de sincronização, os seguintes passos devem ser seguidos para criar o modelo CPN que represente as redes:

- deve-se criar uma transição com o rótulo formado pela junção do nome do cenário com a seguinte constante "*CD_ID_TRUE*", onde *ID* é um número inteiro maior que zero. A transição criada deverá ter como expressão guarda a seguinte expressão "*#id VarName oper Value*", onde *id* é a propriedade utilizada na comparação, *VarName* é o nome da variável criada pra representar a instância, *oper* é o operador da comparação e *Value* é o valor que está sendo comparado;
- deve-se criar uma transição, com o rótulo formado pela junção do nome do cenário com a seguinte constante "*CD_ID_FALSE*", onde *ID* é um número inteiro maior que zero. A transição criada deverá ter como expressão guarda a seguinte expressão "*#id VarName oper Value*", onde *id* é a propriedade utilizada na comparação, *VarName* é o nome da variável criada pra representar a instância, *oper* é o operador da comparação oposto e *Value* é o valor que está sendo comparado;

- em seguida, deve-se criar um lugar de entrada comum para ambas as transições. O tipo deste lugar é o tipo da instância mapeada;
- logo após, cria-se um lugar de saída para a transição definida no passo 1, com o mesmo tipo da instância mapeada;
- da mesma forma, cria-se um lugar de saída para a transição definida no passo 3, com o mesmo tipo da instância. Para garantir que uma condição quente seja sempre testada até que seu valor torne-se verdadeiro, este lugar não pode, em hipótese alguma, ser o mesmo lugar de entrada da transição. Caso a condição seja fria, este lugar deve ser o mesmo do lugar de entrada.
- Por fim, atribui-se o valor da variável desta instância aos arcos de entrada e saída de ambas transições criadas nos passos acima;

O XML da *Play Engine* possui uma *tag* chamada '*Condition*', a qual define o operador da condição, o elemento à esquerda do operador, o elemento à direita do operador, a propriedade que está sendo alterada, entre outros. Além disso, o arquivo define a *tag Temperature* indicando se a condição é quente (valor '1') ou fria (valor '0'). Após esta etapa de identificação das estruturas presentes no XML, segue-se a definição do modelo CPN listado nos passos acima. Da mesma forma, primeiro define-se a leitura do arquivo, com o armazenamento das informações relevantes. Em seguida, com estas informações, instanciam-se as transições, arcos, lugares, tipo e variáveis da classes da CPN tools. As Figuras 42 e 43 definem etapas da instanciação das transições e dos lugares.

```
instancia a transição
CPNTransition tempTransCnd = new CPNTransition(Integer
        .toString(idGeneration), "#" + tempExpression.propId
        + tempExpression.lhsVarId + tempExpression.operator
        + tempExpression.strValue, tempCondition.lscName
        + "CD_" + Integer.toString(idGeneration) + "_TRUE");
newTransSet.put(tempTransCnd.getId(), tempTransCnd);

idGeneration++;

CPNTransition tempTransCndFalse = new CPNTransition(Integer
        .toString(idGeneration), "#" + tempExpression.propId
        + tempExpression.lhsVarId + tempExpression.operator_inv
        + tempExpression.strValue, tempCondition.lscName
        + "CD_" + Integer.toString(idGeneration) + "_FALSE");
newTransSet.put(tempTransCnd.getId(), tempTransCnd);

idGeneration++;
```

Figura 42. Instanciando as transições das condições

```
//instancia os lugares

CPNPlace tempPlaceInCnd = new CPNPlace(Integer
        .toString(idGeneration), "In", tempCondition.symbolic);
newPlaceSet.put(tempPlaceInCnd.getId(), tempPlaceInCnd);

idGeneration++;

CPNPlace tempPlaceOutCnd = new CPNPlace(Integer
        .toString(idGeneration), "Out", tempCondition.symbolic);
newPlaceSet.put(tempPlaceOutCnd.getId(), tempPlaceOutCnd);
```

Figura 43. Instanciados os lugares das Condições

4.4.3 Atribuições

Assim como as condições, as atribuições definem pontos de sincronização dentro dos cenários nos quais estão inseridas. Para cada ponto de sincronização, com a finalidade de mapear todas as atribuições das propriedades do sistema, os seguintes passos devem ser tomados:

- deve-se criar uma nova variável do tipo da propriedade que está sendo armazenada, com o seguinte rótulo: *chartName_LSCVariable*, onde *chartName* é o nome do cenário e *LSCVariable* é uma constante.
- em seguida, cria-se uma transição com o seguinte rótulo: *chartName_AS_ID*, onde *chartName* é o nome do cenário, *_AS_* é uma constante e *ID* é um seqüencial inteiro maior que zero;
- criam-se os lugares de entrada e saída para a transição, com o mesmo tipo da instância mapeada;
- a variável criada deverá ser atribuída aos arcos de entrada e saída da transição;
- deve-se criar um novo lugar de saída para a transição do tipo da propriedade que está sendo salva e com rótulo: *chartName_VariableName*, onde *VariableName* é o nome da variável que está armazenando a propriedade. Este passo deve ser feito para a propriedade que está sendo salva;
- para o arco que liga a transição ao lugar criado neste último passo, deve-se atribuir a expressão *#id VarName*, onde id é a propriedade que está sendo salva e *VarName* é a variável desta instância.

O XML da ferramenta apresenta a *tag Assignment* responsável por manter as informações das atribuições presentes nos cenários. Esta *tag* define a propriedade que está sendo modificada, o valor, o tipo, entre outros. Da mesma forma que nas condições, após a identificação das estruturas, lê-se estas *tags* e armazena as informações mais necessárias. Em seguida, instancia-se as estruturas definidas no passo acima. Nas figuras 44, 45 e 46, seguem-se trechos das instâncias das variáveis, transições e lugares, respectivamente.

```
//instancia as variáveis
CPNVar tempVarAss = new CPNVar(Integer.toString(idGeneration),
        newMapping.returnPropertie((tempAssignment.property)),
        tempAssignment.nameLsc + "_" + tempAssignment.strValue);
newVarSet.put(tempVarAss.getId(), tempVarAss);

idGeneration++;
```

Figura 44. Instanciando as variáveis das Atribuições

```
//instancia as transições
CPNTransition tempTransAss = new CPNTransition(Integer
        .toString(idGeneration), tempAssignment.nameLsc
        + "_AS_" + Integer.toString(idGeneration));
newTransSet.put(tempTransAss.getId(), tempTransAss);

idGeneration++;
```

Figura 45. Instanciando as transições das Atribuições

```
//instancia os lugares
CPNPlace tempPlaceInAss = new CPNPlace(Integer
        .toString(idGeneration), "In", tempAssignment.kind);
newPlaceSet.put(tempPlaceInAss.getId(), tempPlaceInAss);

idGeneration++;
```

Figura 46. Instanciando os lugares das Atribuições

4.4.4 Mensagens

No Capítulo 2 foi mostrado que o LSC possibilita o uso de dois tipos de mensagens, as mensagens quentes e as mensagens frias. As primeiras são caracterizadas pelo rigor imposto de que uma mensagem enviada, precisa ser recebida. Já as segundas são caracterizadas pelo fato da mensagem enviada não necessariamente chegar ao seu destino final. Entretanto, o autor da metodologia de mapeamento [18] considerou que as mensagens frias uma vez enviadas, chegariam ao seu destino final. Isso fez com que não houvesse diferenciação entre estas mensagens. Além do mais, a metodologia apresentada para a modelagem das mensagens recebidas (ou enviadas) pela instância *user*, ou pela instância *environment*, também não foi utilizada, uma vez que essas instâncias só podem ser criadas dinamicamente através do uso da técnica de *Play –in*. Contudo, já foram mostrados no Capítulo 2, os problemas ocorridos durante o uso da ferramenta *Visual Basic*, o que nos impossibilitou o uso desta técnica.

As mensagens ainda fazem parte de dois pequenos grupos, as mensagens síncronas e as mensagens assíncronas. Para as primeiras, não existe uma ordem de estabelecimento da ação de enviar ou receber a mensagem, ou seja, ela é enviada e recebida ao mesmo tempo. Para o mapeamento destas mensagens(e das assíncronas também), o arquivo XML dispõe de uma lista de *tags Messages*, exibindo todas as mensagens trocadas no cenário, bem como todos os atributos necessários para a modelagem e para diferenciar os tipos de mensagens.

Uma mensagem estabelece um ponto de início e um ponto de chegada, de forma que durante a modelagem, precisamos definir o modelo para a instância de envio e de recebimento da mensagem. Assim, para obtermos o modelo CPN do objeto de envio, para representar uma simples mensagem que modifica o valor da propriedade de uma instância do sistema, deve-se seguir estes passos:

- primeiro, cria-se uma transição com o rótulo formado por: "*receiver-instance_propertyName_valueName*",onde *receiver-instance* é o nome da instância que recebe a mensagem, *propertyName* é o nome da propriedade modificada e por fim, *valueName* é o novo valor da propriedade;

- em seguida, criam-se os lugares de entrada e saída para a transição, do mesmo tipo da instância mapeada;
- por fim, criam-se os arcos e atribui as variáveis criadas para esta instância às expressões dos arcos.

Por outro lado, para a criação do modelo CPN da instância de recebimento da mensagem, a modelagem é similar ao primeiro caso. Entretanto, a expressão do arco de entrada será a variável criada pra identificar a instância. Contudo, para o arco de saída, deve-se atribuir o seguinte rótulo: "*cs.set_idi c v*", onde *cs* é o tipo, *idi* é o nome da propriedade, *c* é a variável e *v* é o valor da variável.

Caso a mensagem seja assíncrona, os passos deverão ser os mesmos acima. Entretanto, o rótulo da transição será acrescido na posição inicial de *SND_* e para a instância de recebimento, será acrescentado *RCV_*. Para destacar que uma mensagem é assíncrona os lugares de saída e de entrada das transições não podem ser os mesmos lugares, o que identificaria um mensagem sendo enviada de uma instância para si mesma, ou seja, uma mensagem síncrona.

4.4.5 Cenários Existenciais e Universais

A obtenção dos modelos CPN referentes aos cenários dá-se conforme as instâncias participantes, tanto dos cenários existenciais, quanto dos cenários universais. A participação das instâncias dentro destes cenários definem pontos de sincronização, sendo que para cada ponto deste deveremos realizar os seguintes passos:

- caso a instância participe da pré-condição, deve-se criar duas transições com os seguintes rótulos: "*chart-name_Pch_Start*" e "*chart-name_Pch_End*" para o começo e o término da pré-condição, respectivamente;
- caso a instância participe do corpo do cenário, deve-se criar duas transições com os seguintes rótulos: "*chart-name_CB_Start*" e "*chart-name_CB_End*" para o começo e o término do corpo do cenário, respectivamente;
- deve-se criar os lugares de entrada e saída para estas transições, sendo que o tipo destas deve ser o tipo criado para representar a instância;
- deve-se criar uma variável para representar a instância, e associar esta variável aos arcos de entrada e saída das transições criadas previamente.

Por exemplo, tomando o cenário visualizado na Figura 47, podemos definir a rede resultante para a instância Principal após seguir estes passos acima. Nota-se que, como esta instância participa tanto da pré-condição quanto do corpo do cenário deve-se criar quatro transições. Assim, chegamos a rede mostrada na Figura 48.

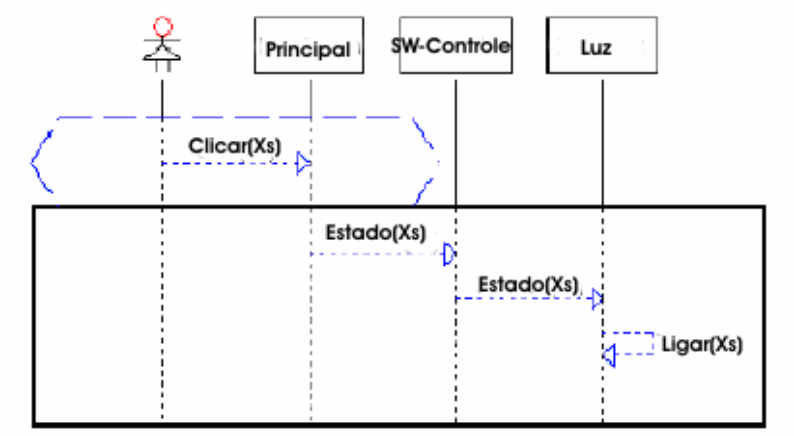


Figura 47. Exemplo de um LSC para criação do modelo CPN correspondente

O arquivo XML da ferramenta apresenta as *tags* LSCs, bem como as *tags Instances* para identificar as instâncias dos cenários. Pode-se distinguir se uma instância participa apenas do corpo do cenário, como também da pré-condição devido à presença da *tag Mode*. Esta *tag* pode possuir dois valores: '*Universal*' ou '*Existencial*'. Assim, com posse desta informação, no mapeamento, decide-se para aquela instância daquele cenário, se deverão ser criadas 4 ou 2 transições. Lendo as informações do arquivo, passa-se a criação do modelo CPN, baseado nos passos acima.

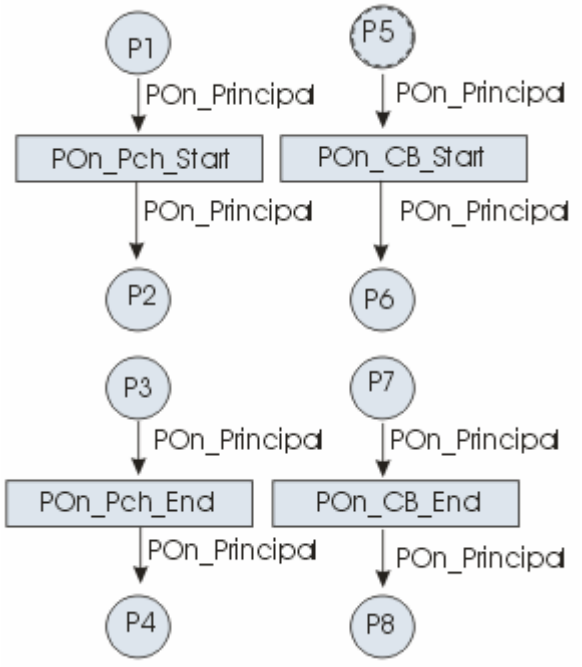


Figura 48. Modelo CPN para a instância Principal

4.4.6 SubCharts

SubCharts são pequenos cenários LSC internos, ou seja, localizados dentro de um cenário LSC maior. A grande vantagem da presença de *subCharts* na LSC quando comparado com os cenários

descritos pela MSC, é a questão da escalabilidade, uma vez que é possível descrever em um único cenário LSC uma situação, a qual só poderia ser descrita em MSC com a utilização de dois cenários. Graficamente, um *subChart* é descrito através de um retângulo com linhas grossas dentro do corpo do cenário. Na Figura 49, segue uma ilustração de um *subChart.*
De forma semelhante aos cenários, uma instância pode estabelecer pontos de sincronização dentro do *subChart* e para cada ponto de sincronização deve-se seguir os passos listados abaixo:

- deve-se criar duas transições com os seguintes rótulos: "*chart-name_Sub_ID_Start*" e "*chart-name_Sub_ID_End*", para o começo e o final do *subChart*, respectivamente. Tem-se que *chart-name* é o nome do *chart*, e ID é um inteiro seqüencial maior que zero;
- em seguida criam-se os lugares de entrada e saída pra estas transições. Os tipos destes lugares devem ser do mesmo tipo criado para representar a instância;
- finalmente, cria-se uma variável para instância e atribui esta aos arcos de entrada e saída das transições.

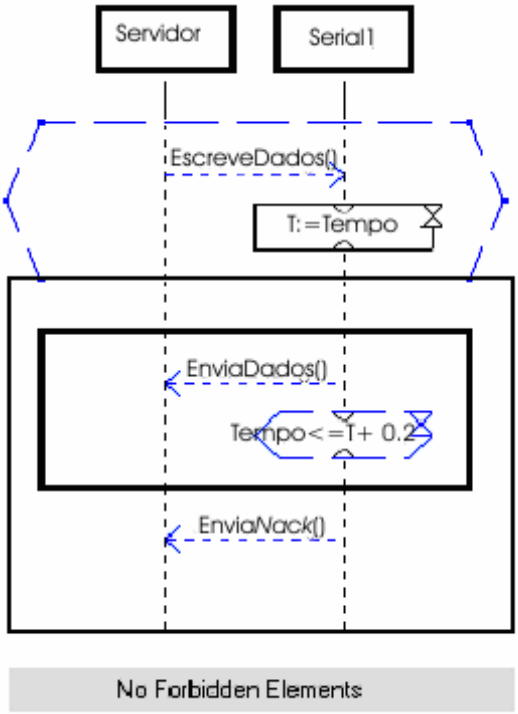


Figura 49. Exemplo da utilização do SubChart

O arquivo XML da *Play Engine* apresenta, não obrigatoriamente, a *tag subChart*, indicando a sua presença no cenário. É através desta que conseguimos recuperar as informações internas e desenvolver o código de mapeamento para o modelo CPN. As Figuras 50 e 51 mostram o processo de instanciação das transições e dos lugares, respectivamente.

```
//instancia as transições
CPNTransition tempTransCHStart = new CPNTransition(Integer
        .toString(idGeneration), tempChartInstance.name + "_Sub_" +

        Integer.toString(idGeneration)
        + "_Start");
idGeneration++;
CPNTransition tempTransCHEnd = new CPNTransition(Integer
        .toString(idGeneration), tempChartInstance.name + "_Sub_" +
        Integer.toString(idGeneration)
        + "_End");
idGeneration++;

newTransSet.put(tempTransCHStart.getId(), tempTransCHStart);
newTransSet.put(tempTransCHEnd.getId(), tempTransCHEnd);
```

Figura 50. Instanciando as transições dos SubCharts

```
//instancia os lugares

CPNPlace tempPlaceCHStarIn = new CPNPlace(Integer
        .toString(idGeneration), "In", tempChartInstance.kind);
idGeneration++;
CPNPlace tempPlaceCHStarOut = new CPNPlace(Integer
        .toString(idGeneration), "Out", tempChartInstance.kind);
idGeneration++;
CPNPlace tempPlaceCHEndIn = new CPNPlace(Integer
        .toString(idGeneration), "In", tempChartInstance.kind);
idGeneration++;
CPNPlace tempPlaceCHEndOut = new CPNPlace(Integer
        .toString(idGeneration), "Out", tempChartInstance.kind);
idGeneration++;


newPlaceSet.put(tempPlaceCHStarIn.getId(), tempPlaceCHStarIn);
newPlaceSet.put(tempPlaceCHStarOut.getId(), tempPlaceCHStarOut);
newPlaceSet.put(tempPlaceCHEndIn.getId(), tempPlaceCHEndIn);
newPlaceSet.put(tempPlaceCHEndOut.getId(), tempPlaceCHEndOut);
```

Figura 51. Instanciando os lugares dos SubCharts

4.4.7 Tempo

Com relação ao tempo, o mapeamento foi realizado sobre o *Vertical Delay*. Há a pretensão de, em projetos futuros, estendermos o mapeamento para os outros tipos de tempo. Como dito anteriormente, este temporizador caracteriza-se por definir um tempo mínimo e um tempo máximo permitido para o envio de uma mensagem. A definição deste tempo pertence a uma única instância do cenário. Nos casos de usos ilustrados no Capítulo 5, poderá ser verificado o uso de uma variação do *Vertical Delay*. Neste caso, a primeira condição representando o tempo mínimo exigido para a resposta da mensagem, não existe. Isto é, a partir do momento que a mensagem chega na instância, esta poderá enviar a mensagem de resposta. Entretanto, há o tempo máximo definido para que esta resposta seja enviada.

Seguindo o mesmo padrão das estruturas anteriores, após mapear as outras estruturas que participam do temporizador, para criar o modelo CPN, os seguintes passos devem ser tomados:

- inicialmente, deve ser criado um tipo “Tempo” para associar a esta instância. Assim, poderá ser aplicado às marcas da rede um rótulo indicando quando aquela marca com a propriedade de tempo poderá ser utilizada;
- adicione a seguinte expressão: *IntInf.toInt(time())* ao arco que liga a transição da atribuição ao lugar que representa a variável especial da atribuição, onde *IntInf.toInt* representará um valor inteiro do tempo capturado;
- para as transições *true* e *false* adicione as seguintes expressões guardas: *[IntInf.toInt(time()) > VarAsg+Min-Delay*] e *[IntInf.toInt(time()) <= VarAsg+Min-Delay]*, onde *VarAsg* é a variável que armazenou o tempo inicial e *Min-Delay*, é o tempo mínimo. Por fim adicione *InstVar @+1*ao arco de saída da transição que mantém o valor falso da condição;
- para as transições *true* e *false* adicione as seguintes expressões guardas: *[IntInf.toInt(time()) <= VarAsg+Max-Delay*] e *[IntInf.toInt(time()) > VarAsg+Max-Delay]*, onde *VarAsg* é a variável que armazenou o tempo inicial e *Max-Delay*, é o tempo máximo.

Após mapear as estruturas presentes no cenário, tem-se que juntar todos estes elementos mapeados. Esta junção é estabelecida pela unificação de lugares com a mesma identificação. O mesmo procedimento deve ser aplicado às transições. Além disso, durante o mapeamento, deve-se ligar os lugares de saída de um elemento aos lugares de entrada representantes do próximo elemento mapeado.

A etapa seguinte é gerar, baseados no arquivo CPN DTD (vide Apêndice A), os lugares, as transições e os arcos, ou seja, definir as informações presentes na *tag page*. Estes são definidos com base nas informações extraídas durante o processo de mapeamento. Nosso intuito é manter as principais informações da manipulação de dados. Por outro lado, verifica-se que as informações gráficas não são definidas de forma que a rede não se apresenta inicialmente legível na *CPNTools*.

Por fim, para que o arquivo XML pudesse ser entendido pela ferramenta *CPN Tools*, foi necessário definir algumas outras *tags*, as quais estão associadas à *tag cpnet*, e que servem para definir: a instanciação da página, o tamanho e o estilo da página, suas dimensões, entre outros elementos. Este código consta no Apêndice B.

O arquivo XM é então escrito no mesmo diretório da ferramenta com o nome *Out.XML*. Além disso, fizemos com que a ferramenta gerasse o mesmo arquivo com a extensão *.cpn, isto é *Out.cpn*, que é o formato padrão utilizado para salvar as redes coloridas na *CPN Tools.*

A metodologia de mapeamento é extensa. Ela ainda inclui diversos outros passos para modelar funções, cenários com instâncias do tipo *User* e *Environment*, tempo horizontal, dentre tantos outros. Entretanto, utilizamos apenas um escopo reduzido deste projeto de modelagem. Contudo, este escopo compreende as principais estruturas presentes nos estudos de caso descritos no Capítulo 5, o qual mostra parte dos cenários desenvolvidos para modelar a comunicação do aparelho ConnectOk com um servidor de aplicações.

Nossa expectativa é que possamos, em projetos futuros, aprimorar a modelagem existente e estender a modelagem para outras estruturas.

Capítulo 5

Estudo de Caso - ConnectOk e Resultados

Durante o período do estudo e desenvolvimento da ferramenta de mapeamento, buscou-se alguns estudos de caso sobre os quais poderíamos trabalhar com esta ferramenta. A partir destes estudos de caso, realizaríamos a avaliação da ferramenta e identificaríamos se o mapeamento está sendo feito de forma correta. Assim, a ferramenta também ajudaria a melhorar a metodologia de mapeamento definida.

5.1 Projeto CONNECTOk!

Para conseguir tal feito, fizemos uma parceira com os desenvolvedores de um novo mecanismo que será em breve utilizado por companhias de água para realizar a medição de água em casas, apartamentos e similares. Este mecanismo é conhecido como ConnectOK! CN5001 Este aparelho, desenvolvido pela Procenge – Processamento de Dados e Engenharia de Sistemas Ltda é dotado de microprocessador e GSM/GPRS, *Global System for Mobile Communications / General Packet Radio Service*[29], utilizando o protocolo TCP-IP,*Transmission Control Protocol/Internet Protocol*[30], destinado à leitura de hidrômetros e sistemas de gás e energia. Este terminal por trabalhar em tempo real, permite, por exemplo, a emissão imediata de contas de água.

Questões como localização, comunicação e distribuição das informações são realizadas a partir da comunicação do aparelho com um servidor central de terminais, que deve ter um IP fixo e uma porta determinada pela comunicação. O fluxo de comunicação é determinado por um *socket* aberto pelo terminal. Assim, o servidor é capaz de identificar o modelo do equipamento, a versão do *firmare*, o *ID* do *Modem* GSM e o *ID* do *SimCard* do terminal. As informações passadas pelos terminais são interpretadas por este servidor, que as processa e envia um pacote de informações como resposta. Além disso, durante a comunicação, é comum o servidor enviar mensagens *ACK* (*acknowlegdment*), caso o terminal pare de fazer qualquer comunicação.

Figura 52. Aparelho ConnectOk

O terminal, ilustrado na Figura 52, é o responsável por criar e manter a comunicação com o servidor de terminais. Este inicialmente conecta-se na rede GPRS de uma companhia de celular, e em seguida, abre uma *socket* no servidor. Ao perceber uma falha na conexão, cabe ao terminal abrir um outro *socket* e restabelecer a comunicação. Por outro lado, cabe ao servidor identificar que um terminal abriu um novo canal de comunicação e, em conseqüência, fechar o *socket* antigo. Além de tudo, os terminais possuem uma porta serial, através da qual os dados podem ser lidos ou enviados para o servidor. Seu visor possui quatro linhas, cada qual com 20 caracteres, sendo que as três primeiras servem para receber as informações provenientes do servidor, e a última é uma linha de edição própria do terminal. Além disso, o aparelho possui um teclado alfanumérico e um botão responsável por realizar a troca entre letras e números , além de botões de liga/desliga, de correção e de envio. Os botões F1, F2, F3 e F4 não possuem uma função específica, sendo utilizados para enviar qualquer informação ao terminal.

De uma forma geral, o servidor poderá fazer as seguintes comunicações com os terminais:

- realizar a identificação do operador (matrícula e senha) e do aparelho (*ID* do *modem*, *ID* do *SimCard* e modelo do equipamento);
- escrever mensagens no visor do aparelho(até180 *bytes*);
- salvar e exibir uma tela padrão na memória;
- emitir sinal de *beep;*
- escrever/ ler da porta serial, como por exemplo, a porta serial da impressora para impressão das contas de água (máximo de 1199 bytes por vez);
- utilizar o GPS para localizar o terminal;
- criar uma ligação telefônica;
- enviar mensagens de *Ack* ou *Nack*.

Já o terminal pode se comunicar com o servidor para:

- responder a um pedido de identificação;

- enviar mensagens de *Ack* ou *Nack*;
- enviar informação (através de números e letras);
- enviar os comandos F1, F2, F3 ou F4;
- responder a um pedido de localização;
- enviar dados requisitados da porta da serial (180 bytes por vez)

Nossa meta foi desenvolver os cenários de funcionamento desta ferramenta, e representá-los através das redes coloridas, para que, posteriormente, pudéssemos verificar as propriedades do mesmo, além de distinguir e corrigir possíveis falhas.

5.2 Desenvolvendo os Estudos de Caso

A idéia inicial era desenvolver os cenários utilizando o mecanismo *Play In* suportado pela ferramenta *Play Engine*. A GUI seria construída a partir da ferramenta *Visual Basic* 6.0[31], a qual provê formas de se comunicar com a *Play Engine.* Nosso intuito então passou a ser a descrição desta interface e, posteriormente, a definição das reações que deveriam ser providas pela GUI devido à interação do usuário.

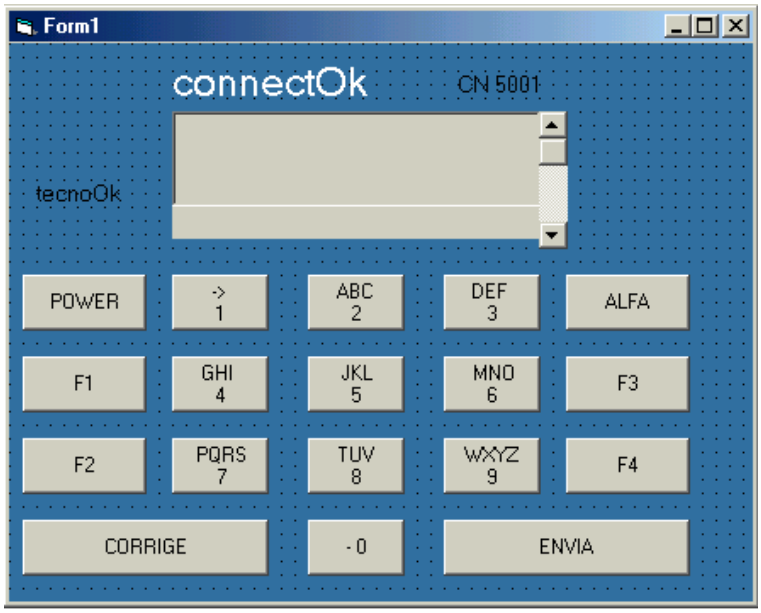
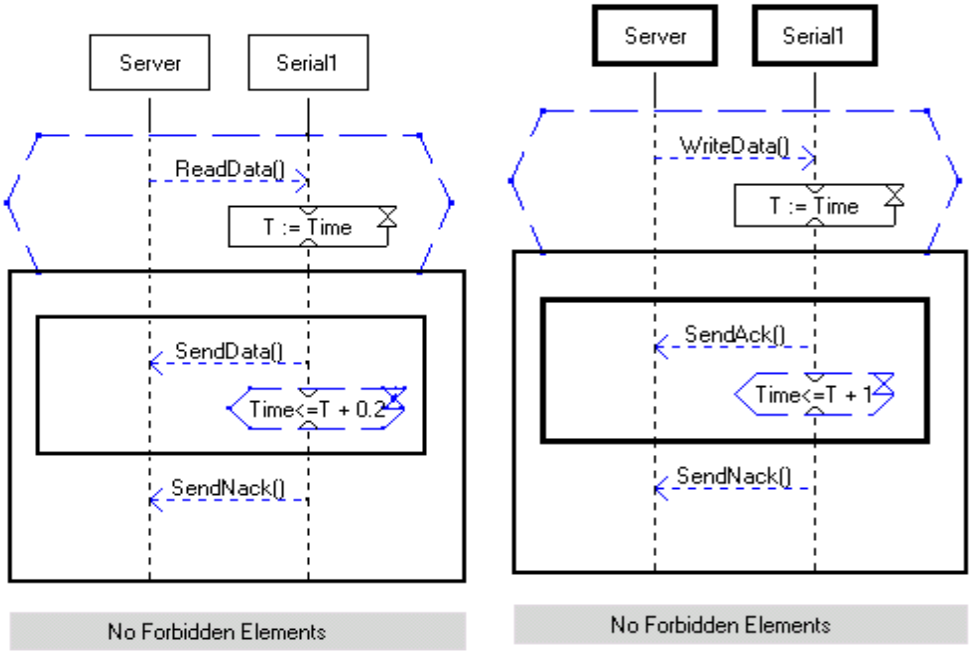


Figura 53. GUI construída no Visual Basic

Na Figura 53, encontra-se a GUI desenvolvida na *Visual Basic*. A partir da GUI, pode-se criar um arquivo do tipo .dll, o qual pode, por sua vez, ser importado pela *Play Engine*. Dessa forma, é possível criar os cenários a partir da GUI, utilizando as técnicas de *Play_in* e *Play-Out* oferecidas. Como dito no Ccapítulo 2, utilizando estas técnicas é possível gerar o código XML dos cenários pela interação com a GUI. Entretanto, a ferramenta *Visual basic* não estava gerando o arquivo .dll para ser importado pela ferramenta *Play Engine*. O problema foi relatado aos idealizadores da ferramenta, mas não foi obtida nenhuma resposta. Por isso, foi necessária a idealização dos cenários através de objetos criados diretamente na *Play-Engine*, ao invés de utilizar os objetos da GUI. Dessa forma, o arquivo XML é gerado à medida que criamos

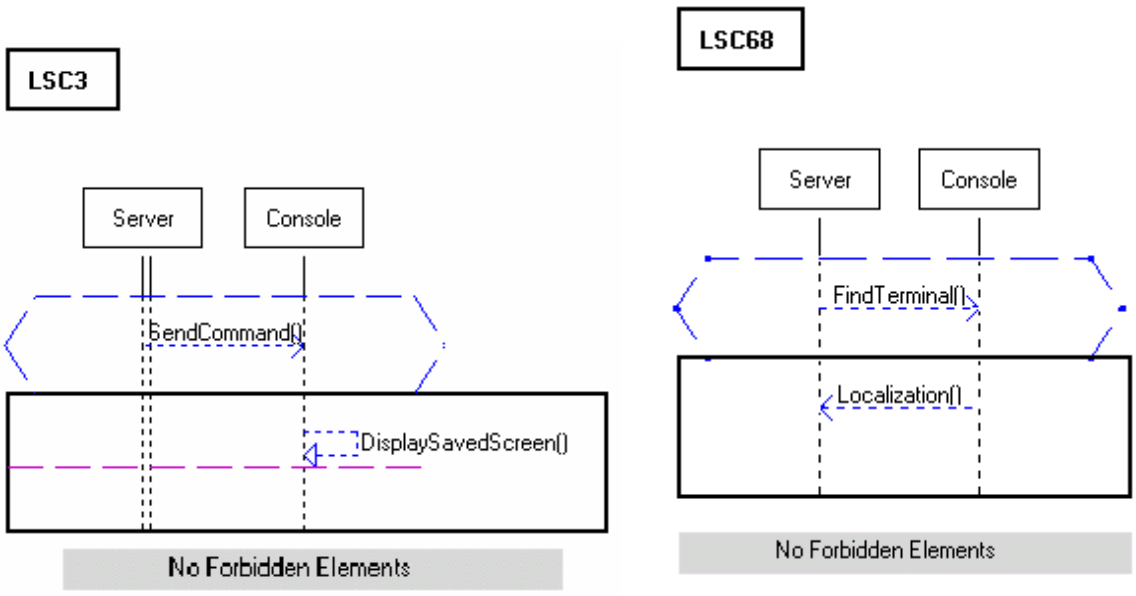
manualmente os cenários. Mesmo assim, ela serviu para tomarmos como referência na criação dos cenários. Os cenários desenvolvidos são baseados no fluxo de comunicação estabelecido entre o terminal e o servidor. Segue os cenários criados na *Play Engine*, cujo arquivos XML foram utilizados na ferramenta de mapeamento criada nesta monografia, para validar a rede de Petri gerada. Na Figura 54(a) e Figura 54(b), segue os primeiros cenários e uma breve explicação de cada um.



(a) (b)

Figura 54. (a)Leitura; (b) Escrita de dados na serial

A Figura 54(a), por exemplo, representa o processo da leitura de informações através de alguma porta serial do ConnectOk. Neste caso, foi criado um objeto Server, que representa o servidor da aplicação que interage com o ConnectOk.



(a) (b)

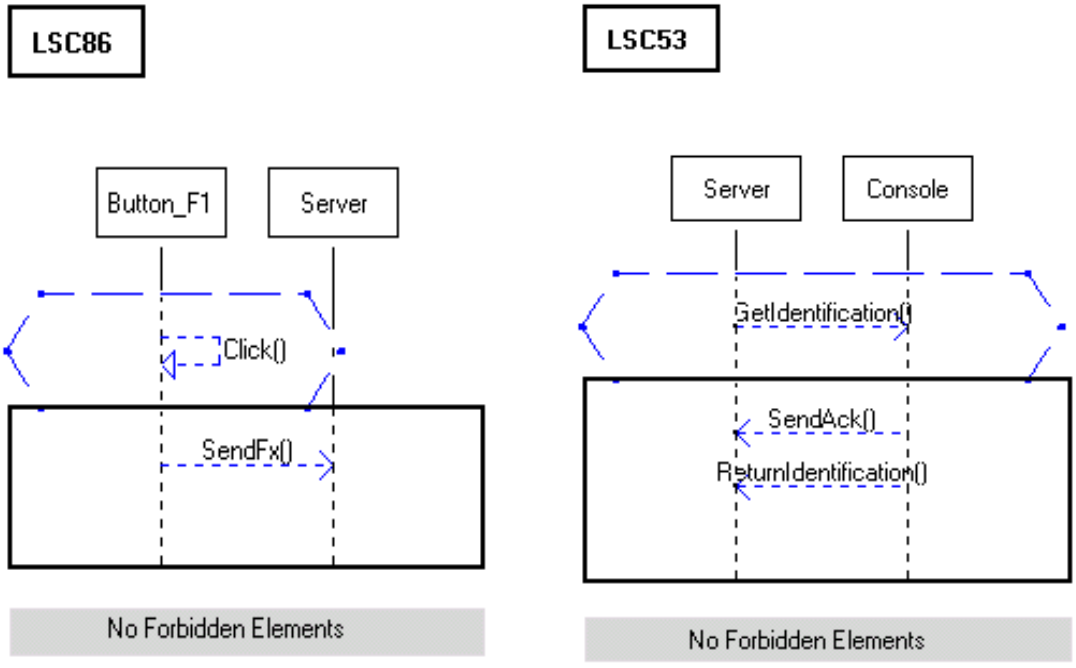
Figura 55. (a) Cenário de exibição de Tela; (b) Localização de Terminal

Assim, inicialmente ele envia uma mensagem “ReadData()” para o objeto serial da aplicação, chamado de “Serial1”. O tempo atual é então armazenado, pois cabe ao “Seria1l” enviar os dados dentro de um determinado intervalo de tempo (neste caso T+0.2). Caso isso não aconteça, o “Serial1” enviará uma mensagem “SendNack()” afirmando que todos os dados não puderam ser enviados neste tempo delimitado.

A Figura 54 (b) é semelhante e mostra o processo de escrita na porta serial. Após receber a mensagem de que dados serão escritos em sua porta, o terminal possui apenas 1 unidade de tempo para pode enviar uma mensagem de Ack, informando ao servidor que ele poderá fazer a escrita dos dados.

O cenário LSC3 na Figura 55 (a) apenas ilustra a requisição do servidor para o terminal mostrar a tela padrão que está salva em sua memória. Note que o objeto “Console”, envia uma mensagem para si mesmo, informando que a tela padrão deverá ser exibida no próprio “Console”, isto é, nas linhas de edição.

O cenário LSC68 na Figura 55 (b), por sua vez, descreve a ação do servidor requisitar a localização do aparelho. Isto será de grande importância quando estes aparelhos, já no mercado, forem utilizados para medir o consumo de água das residências. Assim, o operador sempre estará “visível” para o servidor dos terminais.



(a) (b)

Figura 56. (a)Cenário Botão Fx(); (b) Requisitando identificação

No cenário LSC86 da Figura 56 (a), segue a representação do que ocorre quando o usuário pressiona o botão F1 (ou qualquer um dos outros botões , F2, F3 ou F4). Quando o botão recebe a ação (devido aos problemas da *Visual Basic*, a instância “*User*” não pôde ser utilizada), ele envia para o servidor alguma informação. Diferentemente dos outros botões, após pressionarmos estes, não é necessário pressionar o botão de envio.

Sempre que o servidor solicitar a identificação de um terminal, como ilustrado no cenário LSC53 da Figura 56(b), este, por sua vez, além de enviar uma mensagem *Ack* de reconhecimento,

ele envia aquelas informações descritas anteriormente para identificar tanto o operador do aparelho, quanto o próprio terminal que está sendo operado.

Por fim, o estudo caso do cenário LSC82, encontra-se visualizado na Figura 57. Neste, quando o servidor enviar um comando específico para o terminal, representado através da mensagem "*SendCommand()*", cabe a este último interpretá-lo, exibindo a informação desta requisição no visor da aplicação, e enviar um sinal sonoro como resposta da chegada da mensagem.

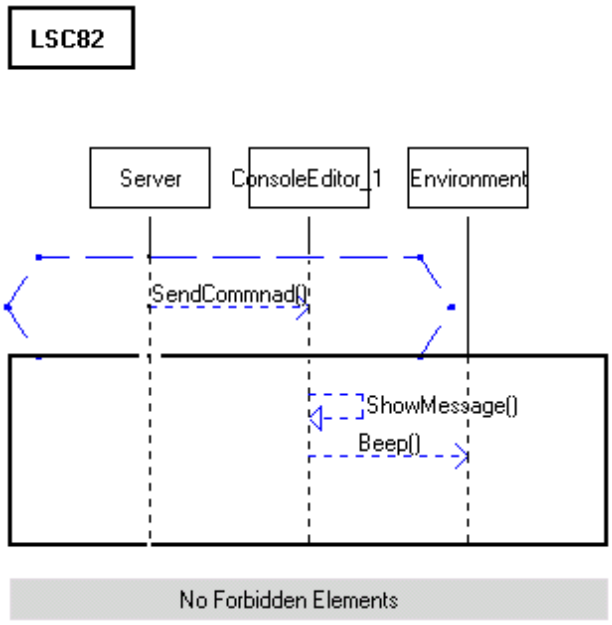


Figura 57. Emissão de Sinal Sonoro

5.3 Rede de Petri colorida do estudo de caso

Durante a tradução constatamos que o mapeamento LSC em redes coloridas define uma enorme quantidade de lugares e transições. Assim, para um maior entendimento, nestas redes serão apenas expressas partes dos estudos do cenário que trabalha com a porta serial do aparelho CONNECTOk!. A partir dos LSCs definidos, a *Play Engine* gera os arquivos do tipo XML para representá-las. A partir destes, é possível gerar os arquivos *Out.XML* e *Out.cpn*. A Figura 58 representa trechos do arquivo XML criado.

Dentro da *tag page*, primeiramente aparecem todas as transições, em seguida todos os lugares e finalmente, os arcos. Uma utilidade desse arquivo XML gerado é gerar posteriormente, um compilador para a linguagem de marcação conhecida como *Petri Net Markup Language* (PNML)[32]. Esta linguagem foi criada em 2000, durante a *International Conference on Application and Theory of Petri Nets*. Dentro os vários padrões para redes de Petri apresentados este se mostrou o mais eficiente para representar as redes de Petri. Hoje, existem no meio acadêmico diversas ferramentas baseadas no PNML.

O arquivo CPN gerado pela ferramenta de mapeamento desenvolvida, já consegue ser interpretado pela *CPN Tools*. Ao carregar o arquivo, além do nome da rede, são exibidas a listas das cores e variáveis presentes na rede (canto esquerdo da ferramenta) e ao centro, a página com a rede gerada a partir do cenário.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE workspaceElements (View Source for full doctype...)>
- <workspaceElements>
    <generator tool="CPN Tools" version="1.5.41" format="5" />
  - <cpnet>
    + <globbox>
    + <page id="id386">
    + <instances>
    + <options>
    + <binders>
      <monitorblock name="Monitors" />
    + <IndexNode expanded="true">
    </cpnet>
  </workspaceElements>
```

Figura 58. Arquivo do tipo XML gerado pela ferramenta

A Figura 59 representa a declaração destes tipos na ferramenta CPN Tools e um trecho da rede que representa os pontos de sincronização de uma instância dentro do corpo do cenário.

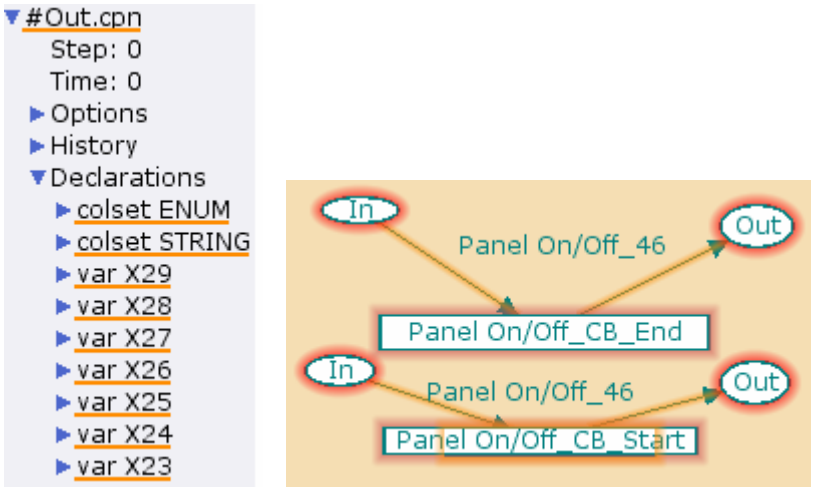


Figura 59. (a)Lista de declaração da CPN Tools; (b) Exemplo do resultado da ferramenta

Essas figuras foram criadas com base no resultado que estamos obtendo com a nossa ferramenta de mapeamento desenvolvida. Como já mencionado, um problema observado com o resultado da compilação é que, por não possuírem informações gráficas, isto é, informações de tamanho de rótulos, das elipses e dos retângulos, os elementos da rede aparecem na *CPN Tools* um sobre o outro, de forma que é necessário arrastar as figuras superiores para se poder visualizar realmente a rede.

5.4 Análise da CPN Tools

A análise das redes coloridas é mais difícil de ser implementada. Embora os conceitos das análises apresentados no capítulo das redes de Petri também sirvam para as redes coloridas, as ferramentas não disponibilizam todas aquelas análises mencionadas.

Ao carregar uma rede, a *CPN Tools* oferece a possibilidade de checar a sintaxe da rede, e verificar se há alguma problema com relação a declaração desta. Ao ser carregada, inicialmente é checado se a declaração das variáveis e das cores está correta. Durante a primeira fase, estas aparecem com um traço em laranja afirmando que o estudo da sintaxe está sendo realizado. Caso haja alguma problema, as declarações passam a ter uma cor vermelha. Caso não, azul. Após isto, será checado os lugares, transições e arcos.

A *CPN Tools* também oferece uma análise chamada *State Space*, que é a análise de alcançabilidade. Através desta, é possível gerar as possíveis marcações que a rede pode ter (desde que sejam informadas as marcações iniciais dos lugares). Entretanto, esta análise só pode ser realizada sobre redes pequenas, isto porque, para redes grandes é impossível gerar o seu *state space.* Após gerar o *state space*, a *CPN Tools* oferece a possibilidade de salvar um arquivo com informações a respeito de *deadlock*, *liveness*, limitação, justiça, como também informações a respeito da estatística da rede. Para a rede da Figura 59, é gerado, por exemplo, um relatório informando o número de elementos e o número de arcos presentes na rede. Além disso, ele informa que a rede é *live*, e que não há nenhuma marcação morta.

```
Liveness Properties
------------------------------------------
  Dead Markings:  None
  Dead Transitions Instances: None
  Live Transitions Instances: All
```

Figura 60. Análise de Liveness

Da mesma forma, ele informa a quantidade mínima e máxima de marcas que cada lugar na rede pode ter. Além destes, a *CPN Tools* também é capaz de informar outras informações a respeito da rede e de sua simulação. Entretanto, esse relatório não cobre questões relativas a simulação das redes coloridas.

Com relação a avaliação de desempenho, a *CPN Tools* ainda não oferece em sua versão atual mecanismos para traçar um avaliação eficiente e poderosa. Seus idealizadores estão implementando novas funcionalidades para auxiliar esta avaliação. A nova versão propõe a utilização de *Monitors* para coletar informações dos dados de saída da simulação, analisá-los e por fim exibi-los em forma de gráfico. Também será possível executar simulações da mesma rede em paralelo, de forma que se possa extrair informações a respeito de qual simulação é mais rápida e precisa.

Ao final deste trabalho, constatamos que nosso objetivo principal foi alcançado. Conseguimos definir uma ferramenta para mapear um escopo limitado das estruturas presentes nos cenários em estruturas de redes de Petri Coloridas e analisar esta rede gerada, buscando sempre características essências para a rede como liveness, ausência de deadlock, segurança e melhorar a rede de forma a obter um melhor desempenho do sistema real. Embora este escopo seja reduzido, notamos que conseguimos mapear os principais pontos dos cenários, isto é, aquelas estruturas mais usadas nos cenários. Os cenários do ConnectOk, por exemplo, foram satisfatoriamente criados com as estruturas desse escopo.

Capítulo 6

Conclusão

Neste capítulo serão mostradas as conclusões desta monografia, bem como os problemas encontrados e algumas perpectivas para trabalhos futuros.

6.1 Conclusões

A contribuição deste trabalho foi desenvolver uma ferramenta de mapeamento entre dois modelos formais, de modo que o arquivo resultante possar ser passado como entrada para uma ferramenta de análise, onde propriedades do sistema possam ser estudadas.

Características importantes como especificar a presença de *liveness* e *deadlock*, isto é, um estado onde o sistema está travado, podem ser aplicadas ao modelo de alto nível do sistema ainda nas fases iniciais de projeto. No Capítulo 4 foi mostrado um método de mapeamento [18] através do qual foi possível construir o compilador capaz de mapear uma especificação LSC em uma especificação baseada em redes de Petri coloridas e que são interpretadas pela ferramenta *CPN Tools*

Além disso, a utilização de um estudo de caso, como demonstrado no Capítulo 5, para testar a ferramenta de mapeamento, tem mostrado-se eficaz. Através deste estudo de caso, estamos conseguindo averiguar se o mapeamento está sendo realizado de forma coerente, de maneira que, ao final, a rede gerada represente fielmente os cenários descritos na *Play Engine*. É importante afirmar que este projeto é extenso e estamos conseguindo os primeiros resultados, os quais uma vez satisfatórios, poderemos partir para outras áreas. Além disso, estudando e validando o comportamento das redes geradas podemos levantar medidas a fim de melhorar a metodologia já definida. Esse melhoramento exige, sobretudo, o desenvolvimento de redes mais compactas.

A ferramenta gera a princípio, dois arquivos. Um do tipo XML e outro do tipo cpn, sendo que este último, é um arquivo da *CPN Tools*. Durante a tradução foi verificado que as principais informações dos cenários, como condições, atribuições, tipos, variáveis, *subCharts*, foram mapeadas em lugares, transições, arcos, cores e variáveis. Entretanto, a metodologia gera redes extensas. Contudo, em geral, o resultado apresentado mostrou-se compatível com nossas pretensões, uma vez que, além de ser aceito pela *CPN Tools*, a rede pôde ser testada, e analisada, como mostrado no estudo de caso do Capítulo 5.

6.2 Dificuldades Encontradas

Durante a fase inicial do projeto encontramos algumas dificuldades que prejudicaram o fluxo normal de desenvolvimento. Com relação ao estudo de caso, nosso planejamento era descrever o funcionamento do ConnectOk utilizando a ferramenta *Visual Basic*. Assim, esperávamos evitar a descrição de todo o funcionamento deste equipamento direto na *Play Engine*, o que é uma tarefa bastante trabalhosa Entretanto, isto não foi possível. A ferramenta *Visual basic* não estava gerando o arquivo apropriado. Estes erros foram reportados aos idealizadores da ferramenta, mas nenhuma resposta foi obtida. Com isso, especificamos o desenvolvimento do ConnectOk diretamente na *Play Engine*. Assim, algumas limitações foram impostas pela ferramenta como, por exemplo, a incapacidade de definir instâncias *User* e *Environment*.

Além disso, essa ferramenta, por ser nova, ainda apresenta alguns problemas graves. Durante a sua manipulação, a *Play Engine* gerava exceções e fechava inesperadamente, fazendo com que partes do estudo de caso fossem perdidas.

Como descrito no Capítulo 4, inicialmente o compilador começou a ser desenvolvido baseado no *framework* desenvolvido em [26]. A metodologia do *framework* era fácil e de simples aplicação para nossos objetivos. Entretanto, depois de um certo tempo de utilização, foi verificado que o arquivo de mapeamento definido por esta metodologia não era tão simples de se trabalhar. Por outro lado, não havia suporte para a utilização do *framework*. Isto fez com que optássemos por usar o JDOM para realizar o mapeamento entre as especificações formais.

6.3 Projetos Futuros

Estamos trabalhando apenas com um escopo reduzido dos objetos da *Play Engine*. Por exemplo, não estamos realizando o mapeamento das funções, e o arquivo XML gerado possui uma única página, isto é, todos os elementos da rede estão estruturados dentro de uma única *tag page*. Por não existir o conceito de várias páginas, também não há objetos de interface, ou seja, objetos que sincronizam estas diferentes páginas. Porém, em trabalhos posteriores, pode-se além de estender a utilização para mais de uma página, aumentar o escopo das estruturas mapeadas para suportar o mapeamento dos outros tipos de tempo, das condições e atribuições específicas para as instâncias *User* e *Environment*, dentre outros.

Além disso, pode-se definir algum mecanismo para gerar as informações gráficas de uma rede resultante de um mapeamento. A ferramenta desenvolvida não gera tais informações gráficas, de forma que, ao abrir o arquivo cpn, os lugares, as transições e os arcos aparecem visualmente sobrepostos. Como as redes geradas são grandes, pode-se definir uma técnica para otimizar essas redes, fundindo, por exemplo, lugares e transições equivalentes. Outra proposta seria a ampliação do compilador para que ele pudesse também, a partir do arquivo XML da *CPN Tools*, gerar os arquivos XML da *Play Engine*.

Bibliografia

[1] RUDOLPH, E., RENIERS, M. **MSC-2000 Interaction for the new Millenium**, Disponível em: < http://www.sdl-forum.org/MSC2000present/index.htm>. Acesso em: 08 de junho de 2005.

[2] BOOCH, G; RUMBAUGH, J., JACOBSON, I. **The Unified Modeling Language User Guide**. Massachusets, EUA:Addison-Wesley, 1999. 475 p.

[3] DAMM, W., HAREL, D. Lscs: Breathing life into message sequence charts. In: **Formal Methods in System Design**. Rehevot, Israel: The Weizmann Institute of Science, 2001, v. 19, p. 45–80.

[4] JENSEN, Kurt. An Introduction to the Practical Use of Coloured Petri Nets. In: Advances in Petri Nets, Lecture Notes in Computer Science. **Proceedings…** Heidelberg: Springer, 1998, v. 1492, p. 237-292.

[5] MAGALHÃES, G.; QUEIROZ, L.; PROTÁSIO, R. **Sistemas operacionais para Sistemas Embarcados**. Salvador: Universidade Federal da Bahia, 2005. Disponível em: <http://twiki.im.ufba.br/pub/MAT154/WebHome/SOE.pdf>. Acesso em: 04 fev. 2006.

[6] TURNER, M. **Permission to believe in a Moore's Law for space launch?**, Disponível em: <http://www.thespacereview.com/article/180/1>. Acesso em: 10 fev. 2006.

[7] WIKIMEDIA FOUNDATION. **System-on-a-Chip**. Disponível em: <http://en.wikipedia.org/wiki/System_on_chip>. Acesso em: 05 março de 2006.

[8] SCHMIDT, J. **Object-Oriented Analysis and Design.**, Disponível em: <http://www.sts.tu-harburg.de/teaching/ws-98.99/OOA+D/entry.html>. Acesso em: 07 março de 2006.

[9] MAUW, S. **The Formalization Of Message Sequence Charts**. Eidhoven, The Netherlands: Dept. of Mathematics and Computing Science, Eindhoven University of Technoloy, 2000. Disponível em: <http://www.cs.umd.edu/~mvz/cmsc435-s05/pdf/formalization.pdf>. Acesso em: 20 maio 2006.

[10] BUNKER, A.; GOPALAKRISHNAN,G. **Using Living Sequence Charts for Hardware Protocol Specification and Compliance Verification**. Utha, USA: School of Computing, University of Utah, 2001. Disponível em: <http://www.engineering.usu.edu/ece/faculty/bunker/pubs/hldvt01.pdf>. Acesso em: 30 fev. 2005.

[11] HAREL, D.; MARELLY, R. **Come, Lets Play: Scenario-Based Programming Using LSCs and Play-Engine**. Rehovot, Israel: Springer-Verlag, 2003, 369 p.

[12] HAREL, D.; MARELLY, R. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In: IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 10., **Proceedings…** Forth Worth, Texas, 2002, p.8-11.

[13] MACIEL, P. R.; LINS, R. D.; CUNHA, P. **Introdução às Redes de Petri e Aplicações**, Campinas: X Escola de Computação, 1996, 183 p.

[14] ROCH, S.; STARKE, Peter H. **INA: Integrated Net Analyzer**, Berlin: Institut Für Informatik, 2003. 230 p.

[15] JENSEN, Kurt. **CPN TOOLS - Computer Tool for Coloured Petri Nets**. Disponível em: <http://wiki.daimi.au.dk/cpntools/cpntools.wiki> Acesso em: 15 de julho de 2005.

[16] CRISTENSEN, S., BISGAARD, T. **Design/CPN Overview of CPN ML Syntax,** Aarhus, Denmark: University of Aarhus, 1996. Disponível em <http://heim.ifi.uio.no/~andersmo/petrinet/manualDesignCPN/CpnML.All.pdf>. Acesso em: 10 jan. 2006.

[17] MILNER, R.; TOFTE, M.; HARPER, R. **The Definition of Standard ML (revised).** Massachussetts, USA: MIT Press, 1997. 128 p.

[18] BARROS, Leonardo A**. An Approach for Analysis and Verification of Embedded Systems properties Based on Coloured Petri Nets and live Sequence Charts**. 2006. 150f. Dissertação (Mestrado) - Curso de Ciência da Computação, Centro de Informática, Universidade Federal de Pernambuco, Recife, 2006.

[19] BRILL. M, DAMM, W et.al. **Live Sequence Charts – an introduction to lines, arrows, and strange boxes in the context of formal verification.** 2003. p.1-26. Disponível em: < http://ses.informatik.uni-oldenburg.de/publications/spp-lncs-use-charts.pdf>. Acesso em: 12 fev. 2006.

[20] PAES, Rodrigo B. **Regulando a Interação de Agentes em Sistemas Abertos – uma abordagem de Leis**. Dissertação (Mestrado) – Departamento de Informática, Pontifica Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2005. p.15-20.

[21] MURATA, T. **Petri Nets: Properties, Analysis and Applications**. In: Proceedings of the IEEE, v. 77, n. 4, 1989.

[22] MARRANGHELLO, N. **Redes de Petri – Conceitos e Aplicações**. São Paulo: DCCE/IBILCE/UNESP, 2005. 33 p.

[23] CPN GROUP. **History of Petri Nets**. Disponivel em: <http://www.daimi.au.dk/CPnets/intro/history.html>. Acesso em: 20 de Abril de 2006.

[24] META SOFTWARE. **Design/CPN Reference Manual for X-Windows Version 2.0**. Cambridge: Meta Software, 1993.

[25] BEAUDOUIN-LAFON, M. et al. CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets. In: International Conference on Application and Theory of Petri Nets, 22., 2001. **Proceedings…** Newcastle upon Tyne, UK: Lecture Notes in Computer Science, Springer-Verlag, 2001, v. 2075, p. 71-80.

[26] ARCOVERDE JR., Adilson. **EZPetri - Um Ambiente para integração de linguagens de descrição de redes de Petri**. 2004. Monografia (Graduação) – Curso de Engenharia da Computação, Departamento de Sistemas Computacionais, Universidade de Pernambuco, Recife, 2004.

[27] HUNTER, J., MCLAUGHLIN, B. *JDOM*, Disponível em: <http://www.jdom.org> Acesso em: 20 de julho de 2005.

[28] INTERNATIONAL BUSINESS MACHINES CORP. Eclipse Platform Technical Overview. Disponível em: <http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html>. Acesso em: 24 set. 2005.

[29] GSM ASSOCIATION. **GSM World - GRPS Platform**. Disponível em: <http://www.gsmworld.com/technology/gprs/index.shtml>. Acesso em: 15 maio 2006.

[30] GILBERT, H. **Introduction to TCP/IP**. Disponível em: <http://www.yale.edu/pclt/COMM/TCPIP.HTM>. Acesso em: 15 maio 2006.

[31] MICROSOFT CORPORATION. **Visual Basic Language Reference**. Disponível em: < http://msdn2.microsoft.com/en-us/library/sh9ywfdk(vs.80).aspx>. Acesso em: 20 maio 2006.

[32] BILLINGTON, Jonathan et al. The Petri Net Markup Language: Concepts, Technology and Tools. In: International Conference on Application and Theory of Petri Nets, 24., 2003. Eindhoven. **Proceedings…** The Netherlands: Lecture Notes in Computer Science, Springer-Verlag, 2003. v. 2679, p. 483 - 505.

Apêndice A

CPN DTD

Apresentaremos aqui o arquivo CPN DTD da *CPN Tools*.

```
<!--
   COPYRIGHT (C) 2005 by the CPN group, University of Aarhus, Denmark.

   Contact: cpntools-support@daimi.au.dk
   WWW URL: http://www.daimi.au.dk/CPNtools/

   File: cpn.dtd
   DTD for XML format for CPN Tools ver. 2.0
     20051201:
-->

<!-- The possible orientations of an arc:
       bothdir = Bidirectional arc: O<->[]
       nodir = Arc without arrows: O-[]
       ptot = Arc from Place to Transition: 0->[]
       ttop = Arc from Transition to Place: []->O -->
<!ENTITY % arcors "CDATA">

<!-- Boolean values -->
<!ENTITY % boolean "(true | false)">

<!-- Colours. These colours corresponds to the standard colours of
     HTML:
     Name and RGB value:
         black  = #000000    green  = #008000
         silver = #c0c0c0    lime   = #00ff00
         gray   = #808080    olive  = #808000
         white  = #ffffff    yellow = #ffff00
         maroon = #800000    navy   = #000080
         red    = #ff0000    blue   = #0000ff
         purple = #800080    teal   = #008080
         fuchsia= #ff00ff    aqua   = #00ffff -->
<!ENTITY % cols "CDATA">

<!-- Types of declarations -->
<!ENTITY % decls "block | color | var | ml | globref">

<!-- Line types: -->
<!ENTITY % lintyps "CDATA">

<!-- Numbers -->
<!ENTITY % number "CDATA">

<!-- Possible attributes of objects:
       posattr = Position attributes
       fillattr = Fill Attributes
       lineattr = Line Attributes
       textattr = Text Attributes -->
```

```
<!ENTITY % objatts "posattr, fillattr, lineattr, textattr">

<!-- Possible fill patterns for objects -->
<!ENTITY % pats "CDATA">

<!-- Possible porttypes:
       in = Input Port.
       out = Output Port.
       inout = Input/Output Port
       general = General Port -->
<!ENTITY % prttyps "CDATA">

<!ELEMENT alias     (id)?>

<!ELEMENT and       (ml)+>

<!ELEMENT annot        (%objatts;, text)>
<!ATTLIST annot     id          ID          #IMPLIED>

<!ELEMENT arc       (%objatts;, arrowattr, transend, placeend, bendpoint*, annot?)>
<!ATTLIST arc       id          ID          #IMPLIED
                    orientation %arcors;    #IMPLIED
                    order       %number;    #IMPLIED>

<!ELEMENT arrowattr EMPTY>
<!ATTLIST arrowattr headsize    %number;    #IMPLIED
                    currentcyckle %number;  #IMPLIED>

<!ELEMENT Aux       (%objatts;, label, text)>
<!ATTLIST Aux       id          ID          #IMPLIED>

<!ELEMENT bendpoint (%objatts;, text?)>
<!ATTLIST bendpoint id          ID          #IMPLIED
                    serial      %number;    #IMPLIED>

<!ELEMENT binders   (cpnbinder)*>

<!ELEMENT binding   EMPTY>
<!ATTLIST binding   x           %number;    #IMPLIED
                    y           %number;    #IMPLIED>

<!ELEMENT block     (id, (%decls;)*)>
<!ATTLIST block     id          ID          #IMPLIED>

<!ELEMENT bool      (with)?>

<!ELEMENT box       EMPTY>
<!ATTLIST box       w           %number;    #IMPLIED
                    h           %number;    #IMPLIED>

<!ELEMENT by        (ml)>

<!ELEMENT color     (id, declare?, timed?, (unit | bool | int | real | string |
                                            enum | index | product | record | list |
                                            union | alias | subset)*, layout?)>
<!ATTLIST color     id          ID          #IMPLIED>

<!ELEMENT channel      (%objatts;, text)>
<!ATTLIST channel      id          ID          #IMPLIED>

<!ELEMENT channel-key  (%objatts;, text)>
<!ATTLIST channel-key  id          ID          #IMPLIED>

<!ELEMENT code      (%objatts;, text)>
<!ATTLIST code      id          ID          #IMPLIED>

<!ELEMENT code-key  (%objatts;, text)>
<!ATTLIST code-key  id          ID          #IMPLIED>

<!ELEMENT cond      (%objatts;, text)>
<!ATTLIST cond      id          ID          #IMPLIED>

<!ELEMENT cpnbinder (sheets,zorder?)>
<!ATTLIST cpnbinder id          ID          #IMPLIED
                    x           %number;    #IMPLIED
```

```
                    y           %number;    #IMPLIED
                    width       %number;    #IMPLIED
                    height      %number;    #IMPLIED>

<!ELEMENT cpnet     (globbox,page*,fusion*,instances,options,binders,monitorblock,IndexNode)>

<!ELEMENT cpnsheet  (zorder?)>
<!ATTLIST cpnsheet  id          ID          #IMPLIED
                    panx        %number;    #IMPLIED
                    pany        %number;    #IMPLIED
                    zoom        %number;    #IMPLIED
                    instance    IDREF       #REQUIRED>

<!ELEMENT declaration (ml)>
<!ATTLIST declaration name      CDATA       #REQUIRED>

<!ELEMENT declare   (id)+>

<!ELEMENT ellipse   EMPTY>
<!ATTLIST ellipse   w           %number;    #IMPLIED
                    h           %number;    #IMPLIED>

<!ELEMENT enum      (id)+>

<!ELEMENT fillattr  EMPTY>
<!ATTLIST fillattr  colour      %cols;      #IMPLIED
                    pattern     %pats;      #IMPLIED
                    filled      %boolean;   #IMPLIED>

<!ELEMENT fusion    (fusion_elm*)>
<!ATTLIST fusion    id          ID          #IMPLIED
                    name        CDATA       #IMPLIED>

<!ELEMENT fusion_elm EMPTY>
<!ATTLIST fusion_elm idref      IDREF       #IMPLIED>

<!ELEMENT fusioninfo (%objatts;)>
<!ATTLIST fusioninfo id         ID          #IMPLIED
                     name       CDATA       #IMPLIED>

<!ELEMENT generator EMPTY>
<!ATTLIST generator tool        CDATA       #IMPLIED
                    version     CDATA       #IMPLIED
                    format      CDATA       #IMPLIED>

<!ELEMENT globbox   (%decls;)*>

<!ELEMENT globref   (id?, ml?, layout?)>
<!ATTLIST globref   id     ID          #IMPLIED>

<!ELEMENT group_elm EMPTY>
<!ATTLIST group_elm idref  IDREF       #IMPLIED>

<!ELEMENT group (group_elm)*>
<!ATTLIST group id          ID          #IMPLIED
                name        CDATA       #IMPLIED>

<!ELEMENT guideline_elm EMPTY>
<!ATTLIST guideline_elm idref   IDREF       #IMPLIED>

<!ELEMENT hguideline (guideline_elm)*>
<!ATTLIST hguideline id         ID          #IMPLIED
                     y          %number;    #IMPLIED>

<!ELEMENT id        (#PCDATA)>

<!ELEMENT index     (ml, ml, id)>

<!ELEMENT IndexNode (IndexNode)*>
<!ATTLIST IndexNode expanded    %boolean;   #REQUIRED>

<!ELEMENT initmark  (%objatts;, text)>
<!ATTLIST initmark  id          ID          #IMPLIED>

<!ELEMENT int       (with)?>
```

```
<!ELEMENT instance  (instance)*>
<!ATTLIST instance  id          ID          #IMPLIED
                    page        IDREF       #IMPLIED
                       trans       IDREF           #IMPLIED>
<!ELEMENT instances (instance)*>

<!ELEMENT label     EMPTY>
<!ATTLIST label     w           %number;    #IMPLIED
                    h           %number;    #IMPLIED>

<!ELEMENT layout    (#PCDATA)>

<!ELEMENT lineattr  EMPTY>
<!ATTLIST lineattr  colour      %cols;      #IMPLIED
                    thick       %number;    #IMPLIED
                    type        %lintyps;   #IMPLIED>

<!ELEMENT list      ((with)?, id)>

<!ELEMENT marking   EMPTY>
<!ATTLIST marking   x           %number;    #IMPLIED
                    y           %number;    #IMPLIED
                    hidden   %boolean;   #IMPLIED>

<!ELEMENT ml        (#PCDATA | layout)*>
<!ATTLIST ml        id          ID          #IMPLIED>

<!ELEMENT monitor   ((node*), (declaration*), (option*))>
<!ATTLIST monitor   id          ID          #REQUIRED
                    name        CDATA       #REQUIRED
                       type        %number;    #REQUIRED
                       typedescription CDATA   #REQUIRED
                       disabled    %boolean;   #REQUIRED>

<!ELEMENT monitorblock (monitor)*>
<!ATTLIST monitorblock name     CDATA       #REQUIRED>

<!-- idref and pageinstanceidref is references to existing nodes.
     Some restriction on the type of node may apply depending on
     the type of monitor. -->
<!ELEMENT node      EMPTY>
<!ATTLIST node      idref       CDATA       #REQUIRED
                    pageinstanceidref CDATA #REQUIRED>

<!-- Options have their value as an element unless they are options
     on a monitor. In that case is the value an attribute. -->
<!ELEMENT option    (value?)>
<!ATTLIST option    name        CDATA       #REQUIRED
                    value       %boolean;   #IMPLIED>

<!ELEMENT options   (option)*>

<!ELEMENT page      (pageattr, (trans | place | arc | Aux | vguideline | hguideline | group)*)>
<!ATTLIST page      id          ID          #IMPLIED>

<!ELEMENT pageattr  EMPTY>
<!ATTLIST pageattr  name        CDATA       #IMPLIED>

<!ELEMENT place     (%objatts;, text, ellipse, (token | marking | fusioninfo | port | type |
initmark)*)>
<!ATTLIST place     id          ID          #IMPLIED>

<!ELEMENT placeend  EMPTY>
<!ATTLIST placeend  idref       IDREF       #IMPLIED>

<!ELEMENT port      (%objatts;)>
<!ATTLIST port      id          ID          #IMPLIED
                    type        %prttyps;   #IMPLIED>

<!ELEMENT posattr   EMPTY>
<!ATTLIST posattr   x           %number;    #IMPLIED
                    y           %number;    #IMPLIED>

<!ELEMENT position  EMPTY>
<!ATTLIST position  value       %number;    #REQUIRED>
```

```
<!ELEMENT product   (id)+>

<!ELEMENT real      (with)?>

<!ELEMENT record    (recordfield)+>

<!ELEMENT recordfield (id,id)>

<!ELEMENT sheets    (cpnsheet | textsheet)*>

<!ELEMENT string    (with)?>

<!ELEMENT subpageinfo (%objatts;)>
<!ATTLIST subpageinfo id          ID          #IMPLIED
                      name        CDATA       #IMPLIED>

<!ELEMENT subset    (id?, (with | by))>

<!ELEMENT subst     (subpageinfo?)>
<!ATTLIST subst     subpage     IDREF       #IMPLIED
                    portsock    CDATA       #IMPLIED>

<!ELEMENT text      (#PCDATA)>
<!ATTLIST text      tool CDATA "CPN Tools"
                    version CDATA #IMPLIED>

<!ELEMENT textattr  EMPTY>
<!ATTLIST textattr  colour      %cols;      #IMPLIED
                    bold        %boolean;   #IMPLIED>

<!ELEMENT textsheet (zorder?)>
<!ATTLIST textsheet id          ID          #IMPLIED
                    panx        %number;    #IMPLIED
                    pany        %number;    #IMPLIED
                    zoom        %number;    #IMPLIED
                    decl        IDREF       #REQUIRED>

<!ELEMENT time      (%objatts;, text)>
<!ATTLIST time      id          ID          #IMPLIED>

<!ELEMENT timed     EMPTY>

<!ELEMENT token     EMPTY>
<!ATTLIST token     x           %number;    #IMPLIED
                    y           %number;    #IMPLIED>

<!ELEMENT trans     (%objatts;, text, box, subst?,binding?, (time | cond | code-key | code |
channel)*)>
<!ATTLIST trans     id          ID          #IMPLIED
                    explicit    %boolean;   #IMPLIED>

<!ELEMENT transend  EMPTY>
<!ATTLIST transend  idref       IDREF       #IMPLIED>

<!ELEMENT type      (id | (%objatts;, text))>
<!ATTLIST type      id          ID          #IMPLIED>

<!ELEMENT union     (unionfield)+>

<!ELEMENT unionfield (id, (type)?)>

<!ELEMENT unit      (with)?>

<!ELEMENT value     (text)>

<!ELEMENT var       (type, id+, layout?)>
<!ATTLIST var       id          ID          #IMPLIED>

<!ELEMENT vguideline (guideline_elm)*>
<!ATTLIST vguideline id          ID          #IMPLIED
                     x           %number;    #IMPLIED>

<!ELEMENT with      ((ml, (ml?|and))|(id,(id)?))>
```

```
<!ELEMENT workspaceElements (generator, cpnet)>

<!ELEMENT zorder    (position)*>
```

Apêndice B

Escrevendo o XML

```
//Primeiro, as transições
        Enumeration enumCpnTrans = newTransSet.elements();
        while (enumCpnTrans.hasMoreElements()) {

                CPNTransition writeTrans = ((CPNTransition) enumCpnTrans
                                .nextElement());

                Element trans = new Element("trans");
                trans.setAttribute(new Attribute("id",
                                "id" + writeTrans.getId()));
                //Element: posattr
                Element posattr = new Element("posattr");
                posattr.setAttribute(new Attribute("x", "0"));
                posattr.setAttribute(new Attribute("y", "0"));
                trans.addContent(posattr);

                //Element: fillattr
                Element fillattr = new Element("fillattr");
                fillattr.setAttribute(new Attribute("colour", "White"));
                fillattr.setAttribute(new Attribute("pattern", "solid"));
                fillattr.setAttribute(new Attribute("filled", "false"));
                trans.addContent(fillattr);

                //Element: lineattr
                Element lineattr = new Element("lineattr");
                lineattr.setAttribute(new Attribute("colour", "Teal"));
                lineattr.setAttribute(new Attribute("thick", "1"));
                lineattr.setAttribute(new Attribute("type", "solid"));
                trans.addContent(lineattr);

                //Element: textattr
                Element textattr = new Element("textattr");
                textattr.setAttribute(new Attribute("colour", "Teal"));
```

```
        textattr.setAttribute(new Attribute("bold", "false"));
        trans.addContent(textattr);

        //Element: text
        Element text = new Element("text");
        text.setText(writeTrans.getNameTrans());
        System.out.println("name" + writeTrans.getNameTrans());
        trans.addContent(text);

        //Element: box
        Element box = new Element("box");
        box.setAttribute(new Attribute("w", "20"));
        box.setAttribute(new Attribute("h", "13"));
        trans.addContent(box);
        page.addContent(trans);

}

//Segundo os lugares

Enumeration enumCpnPlace = newPlaceSet.elements();
while (enumCpnPlace.hasMoreElements()) {

        CPNPlace writePlace = (CPNPlace) enumCpnPlace.nextElement();
        Element place = new Element("place");
        place.setAttribute(new Attribute("id",
                "id" + Integer.toString(idGeneration++)));

        //Element: posattr
        Element posattr = new Element("posattr");
        posattr.setAttribute(new Attribute("x", "0"));
        posattr.setAttribute(new Attribute("y", "0"));
        place.addContent(posattr);

        //Element: fillattr
        Element fillattr = new Element("fillattr");
        fillattr.setAttribute(new Attribute("colour", "White"));
        fillattr.setAttribute(new Attribute("pattern", "solid"));
        fillattr.setAttribute(new Attribute("filled", "false"));
        place.addContent(fillattr);

        //Element: lineattr
        Element lineattr = new Element("lineattr");
        lineattr.setAttribute(new Attribute("colour", "Teal"));
        lineattr.setAttribute(new Attribute("thick", "1"));
        lineattr.setAttribute(new Attribute("type", "solid"));
        place.addContent(lineattr);

        //Element: textattr
```

```
Element textattr = new Element("textattr");
textattr.setAttribute(new Attribute("colour", "Teal"));
textattr.setAttribute(new Attribute("bold", "false"));
place.addContent(textattr);

//Element: text
Element text = new Element("text");
text.setText(writePlace.getName());
place.addContent(text);

//Element: ellipse
Element ellipse = new Element("ellipse");
ellipse.setAttribute(new Attribute("w", "20"));
ellipse.setAttribute(new Attribute("h", "13"));
place.addContent(ellipse);

//Element: type
Element type = new Element("type");
type.setAttribute(new Attribute("id",
               "id" + writePlace.getType()));

//Element: posattr
Element posattrType = new Element("posattr");
posattrType.setAttribute(new Attribute("x", "0"));
posattrType.setAttribute(new Attribute("y", "0"));
type.addContent(posattrType);

//Element: fillattr
Element fillattrType = new Element("fillattr");
fillattrType.setAttribute(new Attribute("colour", "White"));
fillattrType.setAttribute(new Attribute("pattern",
                                                    "solid"));
fillattrType.setAttribute(new Attribute("filled", "false"));
type.addContent(fillattrType);

//Element: lineattr
Element lineattrType = new Element("lineattr");
lineattrType.setAttribute(new Attribute("colour", "Teal"));
lineattrType.setAttribute(new Attribute("thick", "1"));
lineattrType.setAttribute(new Attribute("type", "solid"));
type.addContent(lineattrType);

//Element: textattr
Element textattrType = new Element("textattr");
textattrType.setAttribute(new Attribute("colour", "Teal"));
textattrType.setAttribute(new Attribute("bold", "false"));
type.addContent(textattrType);

//Element: text
```

```
        Element textType = new Element("text");
        if (writePlace != null) {
               textType.setText(writePlace.getType());
        }
        type.addContent(textType);
        place.addContent(type);

        //Element: initimark
        Element initmark = new Element("initmark");
        initmark.setAttribute(new Attribute("id", "id"
                      + Integer.toString(idGeneration++)));

        //Element: posattr
        Element posattrMark = new Element("posattr");
        posattrMark.setAttribute(new Attribute("x", "0"));
        posattrMark.setAttribute(new Attribute("y", "0"));
        initmark.addContent(posattrMark);

        //Element: fillattr
        Element fillattrMark = new Element("fillattr");
        fillattrMark.setAttribute(new Attribute("colour", "White"));
        fillattrMark.setAttribute(new Attribute("pattern",
                                                        "solid"));
        fillattrMark.setAttribute(new Attribute("filled", "false"));
        initmark.addContent(fillattrMark);

        //Element: lineattr
        Element lineattrMark = new Element("lineattr");
        lineattrMark.setAttribute(new Attribute("colour", "Teal"));
        lineattrMark.setAttribute(new Attribute("thick", "1"));
        lineattrMark.setAttribute(new Attribute("type", "solid"));
        initmark.addContent(lineattrMark);

        //Element: textattr
        Element textattrMark = new Element("textattr");
        textattrMark.setAttribute(new Attribute("colour", "Teal"));
        textattrMark.setAttribute(new Attribute("bold", "false"));
        initmark.addContent(textattrMark);

        //Element: text
        Element textMark = new Element("text");
        textMark.setText(writePlace.getType());
        initmark.addContent(textMark);
        place.addContent(initmark);
        page.addContent(place);

}

//Por fim, os arcos
```

```
Enumeration enumCpnArc = newArcSet.elements();
while (enumCpnArc.hasMoreElements()) {

        CPNArc writeArc = (CPNArc) enumCpnArc.nextElement();

        Element arc = new Element("arc");
        arc.setAttribute(new Attribute("id",
                        "id" + Integer.toString(idGeneration++)));
        arc.setAttribute(new Attribute("orientation",
                                    writeArc.getOrientation()));
        arc.setAttribute(new Attribute("order","0"));

        //Element: posattr
        Element posattr = new Element("posattr");
        posattr.setAttribute(new Attribute("x", "0"));
        posattr.setAttribute(new Attribute("y", "0"));
        arc.addContent(posattr);

        //Element: fillattr
        Element fillattr = new Element("fillattr");
        fillattr.setAttribute(new Attribute("colour", "White"));
        fillattr.setAttribute(new Attribute("pattern", "solid"));
        fillattr.setAttribute(new Attribute("filled", "false"));
        arc.addContent(fillattr);

        //Element: lineattr
        Element lineattr = new Element("lineattr");
        lineattr.setAttribute(new Attribute("colour", "Teal"));
        lineattr.setAttribute(new Attribute("thick", "1"));
        lineattr.setAttribute(new Attribute("type", "solid"));
        arc.addContent(lineattr);

        //Element: textattr
        Element textattr = new Element("textattr");
        textattr.setAttribute(new Attribute("colour", "Teal"));
        textattr.setAttribute(new Attribute("bold", "false"));
        arc.addContent(textattr);

        //Element: arrowattr
        Element arrowattr = new Element("arrowattr");
        arrowattr.setAttribute(new Attribute("headsize",
                                                "1.000000"));
        arrowattr.setAttribute(new Attribute("currentcyckle", "2"));
        arc.addContent(arrowattr);

        //Element: transend
        Element transend = new Element("transend");
        transend.setAttribute(new Attribute("idref", "id"
```

```
                + writeArc.getTransend()));
arc.addContent(transend);

//Element: placeend
Element placeend = new Element("placeend");
placeend.setAttribute(new Attribute("idref", "id"
                + writeArc.getPlaceend()));
arc.addContent(placeend);

//Element: annot
Element annot = new Element("annot");
annot.setAttribute(new Attribute("id",
         "id" + Integer.toString(idGeneration++)));

//Element: posattr
Element posattrAnnot = new Element("posattr");
posattrAnnot.setAttribute(new Attribute("x", "0"));
posattrAnnot.setAttribute(new Attribute("y", "0"));
annot.addContent(posattrAnnot);

//Element: fillattr
Element fillattrAnnot = new Element("fillattr");
fillattrAnnot.setAttribute(new Attribute("colour",
                                          "White"));
fillattrAnnot.setAttribute(new Attribute("pattern",
                                          "solid"));
fillattrAnnot.setAttribute(new Attribute("filled",
                                           "false"));
annot.addContent(fillattrAnnot);

//Element: lineattr
Element lineattrAnnot = new Element("lineattr");
lineattrAnnot.setAttribute(new Attribute("colour", "Teal"));
lineattrAnnot.setAttribute(new Attribute("thick", "1"));
lineattrAnnot.setAttribute(new Attribute("type", "solid"));
annot.addContent(lineattrAnnot);

//Element: textattr
Element textattrAnnot = new Element("textattr");
textattrAnnot.setAttribute(new Attribute("colour", "Teal"));
textattrAnnot.setAttribute(new Attribute("bold", "false"));
annot.addContent(textattrAnnot);

//Element: text
Element textAnnot = new Element("text");
textAnnot.setText(writeArc.getText());
annot.addContent(textAnnot);
arc.addContent(annot);
page.addContent(arc);
```

```
}

//Finalmente, cria-se a instanciação da página para ser exibida na
// CPNTools
//Instance
Element instance = new Element("instances");
Element instanceChild = new Element("instance");
instanceChild.setAttribute(new Attribute("id", "ID777777"));
instanceChild.setAttribute(new Attribute("page", pageId));
instance.addContent(instanceChild);
cpnet.addContent(instance);

//Options
Element option = new Element("options");
Element optionChild = new Element("option");
optionChild.setAttribute(new Attribute("name",
                                    "outputdirectory"));
Element optionValue = new Element("value");
Element optionText = new Element("text");
optionText.setText("&lt;same as model&gt;");
optionValue.addContent(optionText);
optionChild.addContent(optionValue);
option.addContent(optionChild);
cpnet.addContent(option);

//Binders
Element binder = new Element("binders");
Element cpnBinder = new Element("cpnbinder");
cpnBinder.setAttribute(new Attribute("id", "ID1002814115"));
cpnBinder.setAttribute(new Attribute("x", "202"));
cpnBinder.setAttribute(new Attribute("y", "154"));
cpnBinder.setAttribute(new Attribute("width", "624"));
cpnBinder.setAttribute(new Attribute("height", "602"));
Element sheets = new Element("sheets");
Element cpnSheets = new Element("cpnsheet");
cpnSheets.setAttribute(new Attribute("id", "ID1002814108"));
cpnSheets.setAttribute(new Attribute("panx", "-6.000000"));
cpnSheets.setAttribute(new Attribute("pany", "184.000000"));
cpnSheets.setAttribute(new Attribute("zoom", "1.000000"));
cpnSheets.setAttribute(new Attribute("instance", "ID555394"));
Element zorder = new Element("zorder");
Element zorderPosition = new Element("position");
zorderPosition.setAttribute(new Attribute("value", "0"));
zorder.addContent(zorderPosition);
cpnSheets.addContent(zorder);
sheets.addContent(cpnSheets);

Element zorderBinder = new Element("zorder");
Element zorderBinderPosition = new Element("position");
```

```
zorderBinderPosition.setAttribute(new Attribute("value", "0"));
zorderBinder.addContent(zorderBinderPosition);

cpnBinder.addContent(sheets);
cpnBinder.addContent(zorderBinder);
binder.addContent(cpnBinder);
cpnet.addContent(binder);

//MonitorBlock
Element monitor = new Element("monitorblock");
monitor.setAttribute(new Attribute("iname", "Monitors"));
cpnet.addContent(monitor);

//IndexNode
Element index = new Element("IndexNode");
index.setAttribute(new Attribute("expanded", "true"));
Element indexFirst = new Element("IndexNode");
indexFirst.setAttribute(new Attribute("expanded", "false"));
Element indexSecond = new Element("IndexNode");
indexSecond.setAttribute(new Attribute("expanded", "false"));
Element indexThird = new Element("IndexNode");
indexThird.setAttribute(new Attribute("expanded", "false"));
Element indexFourth = new Element("IndexNode");
indexFourth.setAttribute(new Attribute("expanded", "false"));
indexThird.addContent(indexFourth);
Element indexFifth = new Element("IndexNode");
indexFifth.setAttribute(new Attribute("expanded", "false"));
Element indexSixty = new Element("IndexNode");
indexSixty.setAttribute(new Attribute("expanded", "false"));
indexFifth.addContent(indexSixty);

Element indexSeventy = new Element("IndexNode");
indexSeventy.setAttribute(new Attribute("expanded", "true"));

Element indexOne = new Element("IndexNode");
indexOne.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexOne);
Element indexTwo = new Element("IndexNode");
indexTwo.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexTwo);
Element indexThree = new Element("IndexNode");
indexThree.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexThree);
Element indexFour = new Element("IndexNode");
indexFour.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexFour);
Element indexFive = new Element("IndexNode");
indexFive.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexFive);
```

```
Element indexSix = new Element("IndexNode");
indexSix.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexSix);
Element indexSeven = new Element("IndexNode");
indexSeven.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexSeven);
Element indexEight = new Element("IndexNode");
indexEight.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexEight);
Element indexNine = new Element("IndexNode");
indexNine.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexNine);
Element indexTen = new Element("IndexNode");
indexTen.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexTen);
Element indexEleven = new Element("IndexNode");
indexEleven.setAttribute(new Attribute("expanded", "false"));
indexSeventy.addContent(indexEleven);

Element indexEighty = new Element("IndexNode");
indexEighty.setAttribute(new Attribute("expanded", "false"));
Element indexNinety = new Element("IndexNode");
indexNinety.setAttribute(new Attribute("expanded", "true"));

index.addContent(indexFirst);
index.addContent(indexSecond);
index.addContent(indexThird);
index.addContent(indexFifth);
index.addContent(indexSeventy);
index.addContent(indexEighty);
index.addContent(indexNinety);
cpnet.addContent(index);
```