UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA




# UMA ABORDAGEM PARA EXTRAÇÃO DE ESPECIFICAÇÃO CSP A PARTIR DE UMA IMPLEMENTAÇÃO EM LINGUAGEM C

*• Trabalho de Graduação•*

***Aluno:*** ***Farley Millano de Mendonça Fernandes***

***Orientador:*** ***Alexandre Cabral Mota***

Agosto, 2007

## ASSINATURAS

Este Trabalho de Graduação é resultado dos esforços do aluno Farley Millano de Mendonça Fernandes, sob a orientação do professor Alexandre Cabral Mota, sob o título: "*Uma Abordagem para Extração de Especificação CSP a partir de uma Implementação em Linguagem C*". Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

_____

Farley Millano de Mendonça Fernandes

_____

Alexandre Cabral Mota

*“Está em cima com o céu e o luar*
*Hora dos dias, semanas, meses, anos, décadas*
*E séculos, milênios que vão passar*
*Água-marinha põe estrelas no mar*
*Praias, baías, braços, cabos, mares, golfos*
*E penínsulas e oceanos que não vão secar.”*
(Nando Reis, As coisas tão mais lindas)

*Dedicado à minha família, mãe, Billy e Mateus por todo apoio, toda vida, esforços e amor até chegarmos aqui.*
*Dedicado à Alana por ser o amor da minha vida, por todo carinho, companherismo e ser dela o sorriso que me inspira.*

## *AGRADECIMENTOS*

A Deus por ter me iluminado com minha vida, saúde, força e intelecto suficiente para caminhar ao longo da vida.

A minha família: Elis, Billy e Mateus por serem minha base de tudo provendo todo suporte emocional, educacional e afetivo durante toda vida. Não foi dado um passo sequer na minha vida que não contasse com o apoio deles.

A Alana por todo amor, carinho, companheirismo e apoio, e principalmente por entender da melhor forma minha vida tão corrida e não dar a ela nem metade da atenção que eu acho que deveria dar. A gente vive na certeza de que o melhor está sempre por vir, e assim cuidamos tão bem um do outro.

A minha avó Lourdinha por ser um poço infindável de alegria, carinho, amor e principalmente doação ao próximo, poucas pessoas nesse mundo possuem tal desprendimento e compaixão pelo próximo como ela. Ao meu avô Edílson e aos meus tios: Eliane, Magno, Merinha e Emerson por serem sempre acolhedores comigo.

Ao outro lado da família: meu pai Ricardo que apesar da distância nunca se fez distante afetivamente me dando todo carinho e amor que um filho merece. À minha avó Marinete por ter sempre mostrado um amor incondicional por mim. Ao meu avô Euzamar por sua alegria mostrando como a vida realmente deve ser vivida. À Cida e meu irmão Ricardo Júnior por toda mostra de carinho e receptividade dada a mim quando posso estar por perto. A Diego por ser minha primeira referência de amigo na vida, tornando-se um irmão que a vida me deu.

Ao professor Alexandre Mota pela excelente orientação ao longo do desenvolvimento do trabalho.

Aos amigos 2002.2 Ciências e Engenharia: Allan, Zé Carlos, Geraldo, Leo, Laís, Dudu, Williams, Millena Paulo, Sylvinha (passamos no vestibular!), Joabe por

ter ajudado na escolha do trabalho, e em especial aos meus amigos “empreendedores MobilIT” de longa data Adriano e Vitor por terem sido pessoas com as quais pude sempre compartilhar e dividir os nossos sucessos e conhecimentos, mas também muito esforço e trabalho.

Aos amigos de longa data: Tiago, Gustavo, Herdras, Anderson Scooby e Manoela Pangaré, por terem sido pessoas que me ajudaram e participaram comigo de uma etapa muito importante na vida de cada um.

Aos mestres do ensino médio Ednaldo Ernesto e José Alvino, por terem sido pessoas que me inspiraram por seus conhecimentos e principalmente com a simplicidade com que passam isso.

A Chico Science & Nação Zumbi, Led Zeppelin e Nando Reis por terem sido minha trilhas musicais.

Agradecer é sempre ingrato a quem o faz, então para todos aqueles que minha fraca memória esqueceu, fica meu muito obrigado também!

## *RESUMO*

A implementação de sistemas é uma das últimas etapas no ciclo de desenvolvimento de sistemas e onde se consolidam as intenções de esforços prévios, tais como: a especificação de requisitos, a especificação formal e a modelagem arquitetural. Porém a implementação pode distanciar-se das mesmas por diversos motivos, um deles é o fato de cada etapa possuir sua forma de representação para seus modelos, variando de linguagens naturais, como o inglês, a linguagens de especificação formal, como CSP.

Algumas alternativas, como inspeções de códigos e artefatos, têm como objetivo a minimização de discrepâncias entre especificações e implementação, mas ainda não há forma de se garantir uma boa cobertura por parte destas técnicas. O principal agravante se dá pelo fato da impossibilidade da extração de modelos formais a partir da implementação em certas linguagens ou de certas restrições a sistemas em que isso é possível.

O objetivo maior deste trabalho é a combinação entre as vantagens oferecidas pela linguagem C juntamente com possibilidade de visualização da aderência à modelagem em CSP através de uma extração direta do mesmo, isso representa um esforço de grande valia para indústria de sistemas críticos. Pois assim será possível uma visualização pós-implementação de propriedades (por exemplo: Deadlock, Livelock e Determinismo) atestando propriedades intrínsecas desejáveis aos sistemas, levando a um conseqüente aumento de confiabilidade nos mesmos.

Haverá o desenvolvimento de técnicas sobre um sistema real de metrô para uma documentação das mesmas em seguida, de forma que possam ser incorporadas durante o processo de desenvolvimento de sistemas. Isso fará uma ligação desejável e sutil entre o formalismo das notações em CSP junto à linguagem de programação C.



**Palavras chave:** Engenharia de software, Linguagem C, Linguagem CSP, Métodos Formais

## *ABSTRACT*

The implementation of computer systems is of the final steps within the software development life cycle and, most importantly, it's a stage where all previous efforts, such as: requirements specification, formal specification and architectural modeling are consolidated. However the implementation may get far from them for many reasons, one of them is the fact that each stage has its own representation scheme for its models, varying from natural languages, such as English, to formal specification languages, such as CSP.

Some alternatives, such as code and artifacts inspections, have as the goal of minimizing the discrepancies between specifications and implementations. Unfortunately, these techniques still not provide a good code coverage with respect to the requirements. The most critical fact is that it's not possible to extract formal models from the implementation in some programming languages or some restrictions to the systems when it's possible.

The main objective of this work is the combination between the advantages offered by the C language and the possibility of visualizing the accordance to the CSP modeling, this represents an effort of great value for critical systems industry. Thus, it will be possible have a post-implementation visualization of system properties (such as Deadlock, Livelock and Determinism) previously modeled in the CSP language to verify desirable intrinsic properties to the systems, taking to a significant productivity gaining and consequent increase of trustworthiness on those targeted systems.

There will be the development of techniques over a real subway system for a documentation of them, so that they can be incorporated inside systems development process. This will make a desirable and subtle linking between the formalism of CSP specifications and the C programming language.



**Key words:** Software Engineering, C Language, CSP Language, Formal Methods.

# Índice

# Índice de figuras

# 1. Capítulo 1 – INTRODUÇÃO

A engenharia de software [22] tem buscado cada vez mais a automação e a coesão entre etapas prévias e posteriores em seu ciclo de desenvolvimento, de forma que o software mantenha a consistência com o que se espera dele desde o início de seu planejamento na sua fase de requisitos. Isso traz benefícios não somente para a área técnica, mas também em níveis gerencial e organizacional – tornando o software melhor estimável em termos de escopo, custo e tempo, principalmente confiável para seus clientes.

Métodos formais [24] são em princípio mais usados apenas para sistemas críticos – entenda-se por isso, sistemas onde estão vidas envolvidas e onde todos comportamentos precisam ser analisados previamente –, apesar de todos os benefícios provados, devido a uma questão de cultura organizacional e demanda de pessoal extremamente especializada. A falsa impressão de elevação de custos de um projeto com a inserção de métodos formais é suprimida pelo fato dos custos serem muito maiores em uma mudança de requisito mal avaliada em um estágio avançado do mesmo. E a principalmente vantagem é dada pela possibilidade de geração automática de outros artefatos. Em 1 é mostrada uma ilustração de um gráfico de custo de um defeito ao longo do ciclo de vida de um projeto.

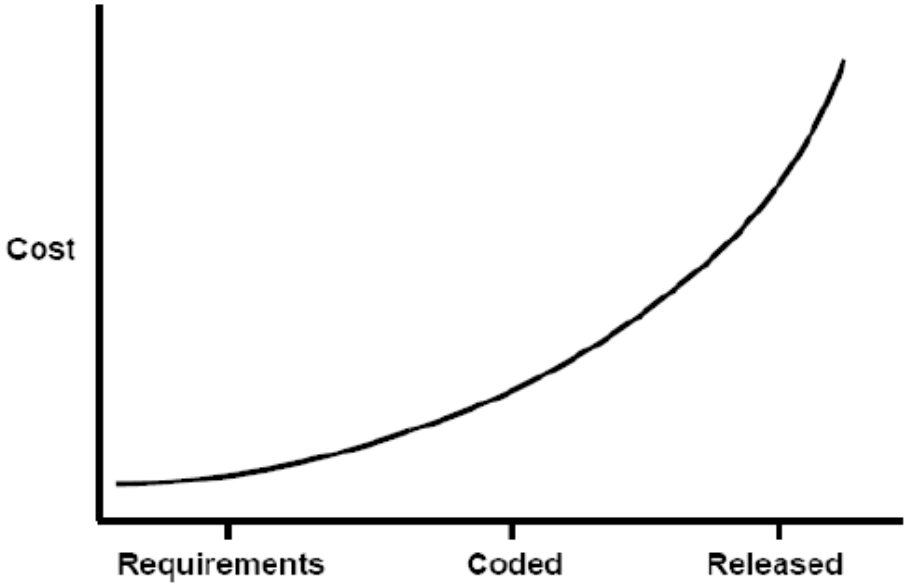



**Figura 1 - Gráfico custo defeito x fase do projeto**

A evolução da área de estudo em métodos formais mostra um caminho que faz com que se consiga inserir de forma sutil modelagens formais nos ciclos de desenvolvimento de software. Automatização na geração de código (implementação) de software tem sido o passo principal dessa evolução, já que as linguagens formais fornecem o poder de mapear propriedades semânticas através de manipulações algébricas e sintáticas.

A ilustração de tal fato se dá no desenvolvimento estudo e suporte ferramental de linguagens de modelagem formal como: Z [9], CSP [6] e Circus[23].

## 1.1. Extração de modelos

Dada a crescente complexidade dos sistemas atualmente em termos de processamento, distribuição e modularização, faz-se necessário criar ferramentas apropriadas para atestar propriedades esperadas de um sistema, principalmente após mudanças de requisitos. Visto que isoladamente os testes de software propriamente ditos não conseguem comprovar tal fato [25].

A linguagem de modelagem CSP [7] (Communicating Sequential Processes) visa modelar com um bom nível de abstração todos os processos que estão inseridos em um sistema e através disso conseguir visualizar propriedades do mesmo. A modelagem escrita nesta linguagem pode ser colocada em análise pela ferramenta FDR[14]. Esta ferramenta além de analisar propriedades (Deadlock, Livelock e Determinismo) verifica também refinamentos de uma especificação.

O existente background ferramental e teórico torna interessante que haja a possibilidade de extração de modelos formais a partir da implementação de um sistema. Isso significa mapear a implementação de um sistema codificado em uma certa linguagem em um modelo formal, o que à primeira vista o que parece ser um passo atrás – pois normalmente modelagens e análises são realizadas antes de implementações –, torna-se extremamente útil já que propriedades esperadas inicialmente, durante a fase de requisitos e modelagem arquitetural, podem ser verificadas após a fase de implementação usando, por exemplo, uma ferramenta como a FDR [14].

## 1.2. O problema

Como mostrado anteriormente no gráfico custo de correção x tempo (figura 1), uma análise de mudança de requisitos ou implementação mal feita é extremamente custosa para um projeto podendo inclusive, dependendo da gravidade, levá-lo a um cancelamento.

É necessário formular-se um “contrato” entre fases prévias e posteriores de um projeto. Uma modelagem em CSP fornece critérios suficientes para tal tarefa, devido ao fato de haver ferramentas que conseguem extrair propriedades e provar refinamentos – conceito este que será explicado em mais detalhe à frente –, da mesma em relação a uma outra modelagem distinta.

## 1.3. Contexto

Este trabalho visa ser parte inicial de um conjunto de trabalhos que procuram fechar o ciclo de verificação de propriedades de um sistema. Tanto no “fluxo natural” partindo de requisitos – geração automática de modelagem formal a partir de requisitos – quanto pós-implementação – extração do modelo após implementação –, foco deste trabalho. De modo mais restritivo será explorada a implementação na linguagem C de um sistema alvo escolhido – sistema de controle de portas de um metrô –, já que esta linguagem possui relação bastante íntima com sistemas críticos por seu nível de performance e daí vem sua necessidade de certo nível de formalismo.

O sistema que terá sua implementação explorada será um controle de portas de um metrô do projeto Santiago Linha 2 no Chile, e a partir dele será definido um padrão para extração do modelo formal específico baseado no entendimento dos módulos componentes do sistema, bem como a estruturação da implementação na linguagem C. Em 2 ilustra-se o conjunto maior de trabalhos que se pretende atingir, com destaque mais escuro para escopo do trabalho aqui discutido.

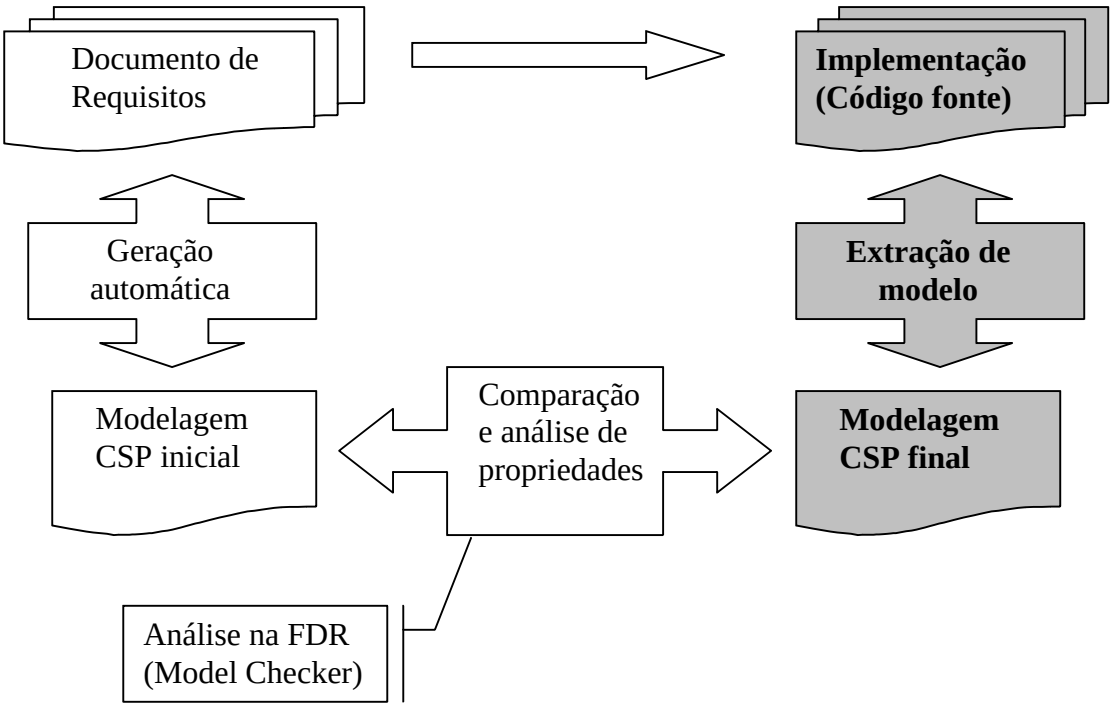



**Figura 2 - Contexto geral de trabalhos**

## 1.4. Contribuições do trabalho

Segue abaixo uma lista com as principais contribuições do trabalho:

- Definição de um conjunto de regras de mapeamento de estruturas sintáticas entre C e CSP voltadas para o domínio do sistema do estudo de caso: Controladora Geral de Portas de um Metrô.

- Análise de propriedades (Deadlock, Livelock e Determinismo) da Controladora Geral de Portas. Será analisada também a Alcançabilidade para estados de operação em navegação interna escolhidos da CGP.

- Comparação isolada das propriedades do modelo CSP extraído por este trabalho com o modelo CSP extraído a partir de requisitos em [17].

- Identificação de modularizações a serem feitas para o sistema do estudo de caso.

## 1.5. Organização do trabalho

A estruturação do trabalho se dará da seguinte forma:

- Capítulo 2 – CSP:

  Neste capítulo são explorados os conceitos envolvidos na linguagem CSP já que nela será representado o resultado final do presente trabalho.

- Capítulo 3 – C:

  O capítulo 3 veio com o objetivo de mostrar um pouco da evolução da linguagem C ao longo de sua história, como também esclarecer termos que são usados na implementação do sistema do nosso estudo de caso.

- Capítulo 4 – Estudo de caso: Metrô:

  O estudo de caso deste capítulo irá traduzir todas as intenções discutidas ao longo do trabalho. Primeiramente haverá uma breve introdução sobre o sistema discutido, em seguida serão mostradas quais funcionalidades serão trabalhadas. Em cima das funcionalidades escolhidas serão definidos os padrões para extração da modelagem bem como o mapeamento em CSP do sistema de controle de portas do metrô. Por fim, será mostrado o sumário de resultados obtidos por avaliação de propriedades e comparação de propriedades isoladas com

a outra vertente do trabalho maior na qual este presente se inclui que é a extração da modelagem em CSP a partir de requisitos [17].

- Capítulo 5 – Conclusão:

Neste capítulo é feito um sumário de todo o trabalho realizado e as contribuições, mostrando o relacionamento com outros trabalhos feitos. E por fim são mencionadas as oportunidades e limitações deixadas por este trabalho a serem desenvolvidas no futuro.

# 2. Capítulo 2 – CSP

CSP [7] surgiu na década de 1970 através do trabalho de C. A.Hoare [6]. Ele introduziu a idéia de processos com variáveis locais que interagem apenas através de trocas de mensagens. Em sua essência, aquela versão era apenas uma linguagem para programação concorrente e não tinha uma semântica matematicamente definida.

Nos anos posteriores, Hoare conseguiu evoluiu sua idéia e apresentou a versão teórica de CSP [7]. Desde então, a teoria de CSP apresentou apenas pequenas mudanças. A motivação para a criação dessas mudanças é a concepção de ferramentas de análise e verificação automáticas. O texto definitivo para a teoria de CSP é apresentado no trabalho de A. W. Roscoe [2].

CSP (Communicating Sequential Processes) é uma linguagem formal utilizada para modelar o comportamento de sistemas concorrentes e distribuídos. Uma forma de entender CSP é imaginar um sistema como uma composição de unidades comportamentais independentes (subsistemas, componentes ou simplesmente rotinas de processamento) que se comunicam entre si e com o ambiente que os cerca [7]. Cada uma destas unidades independentes pode ser formada por unidades menores, combinadas por algum padrão de interação específico. Consideramos o ambiente como todo agente externo que pode interagir com o sistema, como os seus usuários ou outros sistemas.

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

As interações entre processos diferentes, e a maneira que cada processo se comunica com seu ambiente são descritos usando vários operadores algébricos de processos. Usando esta solução algébrica, as descrições de

processos complexos podem ser facilmente construídas a partir de alguns elementos primitivos.

Alguns dos construtores e operadores de CSP são o datatype (datatype D), o event (channel a: D), o prefix (a → P), escolha determinística (P □ Q), escolha não-determinística (P ⊓ Q), interleaving (P ||| Q), a composição paralela (P | [s] | Q, onde s é o conjunto de eventos em que P e Q sincronizam), e o hiding (P \ s, onde s é o conjunto dos eventos a ser escondido). Estes construtores e operadores são detalhados em seguida.

## 2.1. Elementos da linguagem CSP

### Processo

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

O comportamento de um processo de CSP é descrito em termos de eventos, que são operações imediatas, como ABRIR ou FECHAR que pode transmitir informações. Um processo primitivo pode ser compreendido como uma representação de um comportamento básico. Há dois processos primitivos em CSP: *STOP* e *SKIP*. STOP é o processo que não se comunica com nada. Ele é usado para descrever a falha de um sistema, assim como uma situação de deadlock. O SKIP é o processo que indica o término com sucesso do comportamento de um processo. Na figura 3 abaixo detalhamos os operadores algébricos e os processos primitivos de CSP usados para compor a equação de processos complexos.


**Figura 3 - Equações de processos**

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

### Datatype

Sempre que alguma informação necessita ser transmitida é necessário definir seu tipo e valores possíveis a partir do datatype. Conseqüentemente, os

datatypes complexos podem ser definidos e usados nas definições de canais (eventos que transferem dados) para especificar a comunicação dos canais.

### Comunicação

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

Uma comunicação pode ser:

- **Evento** – Um evento é a menor unidade em uma especificação CSP e representa a ocorrência de um fato relevante para entender o comportamento do sistema. Todos os eventos de uma especificação pertencem a um alfabeto (§), e este define o nível de abstração da especificação. Os eventos descrevem as interações mais simples entre os processos. Para um processo RELOGIO, poderíamos definir os eventos tic e tac, que indicam a passagem do tempo.

- **Canal** – Os canais diferem-se dos eventos devido ao fato de transmitirem dados. Os tipos dos dados transmitidos devem estar de acordo com o tipo do canal. Em um sistema que especifica um banco, um processo CAIXA poderia informar o valor debitado da conta de um cliente ao processo SERVIDOR através de um canal debitar, por exemplo. Além de informar a ação do débito, a comunicação informa o valor do débito.

### Alfabeto

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

## 2.2. Operadores

### Prefixo

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

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

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

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

**Figura 4 - Operador Prefixo**

### Recursão

A Recursão em CSP é a habilidade de um processo de incorporar um comportamento repetitivo. O operador ($\rightarrow$), apresentado na seção anterior, pode ser usado para modelar processos recursivos. O comportamento dos processos ($P = x \rightarrow P$) é a repetição indefinida do evento x.

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

$$P = up \rightarrow Q$$

$$Q = down \rightarrow P$$

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

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

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

**Composição Sequencial**

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

$$Q; R$$

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

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

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

a variável x não será percebida pelo processo P.

**Escolha Interna e Externa**

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

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

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

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

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

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

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

### Escolha Condicional

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

*if (b) then P else STOP*

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

**Composição Paralela**

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

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

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

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

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

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

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

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

## 2.3. Semântica

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

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

### Modelo de Traces

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

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

traces(P) sempre contém a seqüência vazia (<>).

Se s^t pertence a traces(P), então s também pertence.

Para ilustrar este modelo, considere o processo S definido por:

*S = a -> b -> c -> STOP*

ele é representado no modelo de traces como:

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

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

*traces(STOP) = { < > }*

*traces(a -> P) = { < > } U { < a > ^ s | s □ traces(P) }*

*traces( P ⊓ Q) = traces(P) U traces(Q)*

*traces( P [] Q) = traces(P) U traces(Q)*

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

**Modelo de Falhas**

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

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

$failures(STOP) = \{ (<>, X) \mid X \subseteq \Sigma\checkmark \}$

$failures(SKIP) = \{ (<>, X) \mid X \subseteq \Sigma \} \cup \{ (<>, X) \mid X \subseteq \Sigma\checkmark \}$

$failures ( P \sqcap Q) = failures (P) \cup failures (Q)$

**Modelo de Falhas e Divergências**

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

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

$divergences(STOP) = \{ \}$

$divergences(SKIP) = \{ \}$

$divergences ( P \sqcap Q) = traces(P) \cup traces(Q)$

*divergences ( P [] Q) = traces(P) U traces(Q)*

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

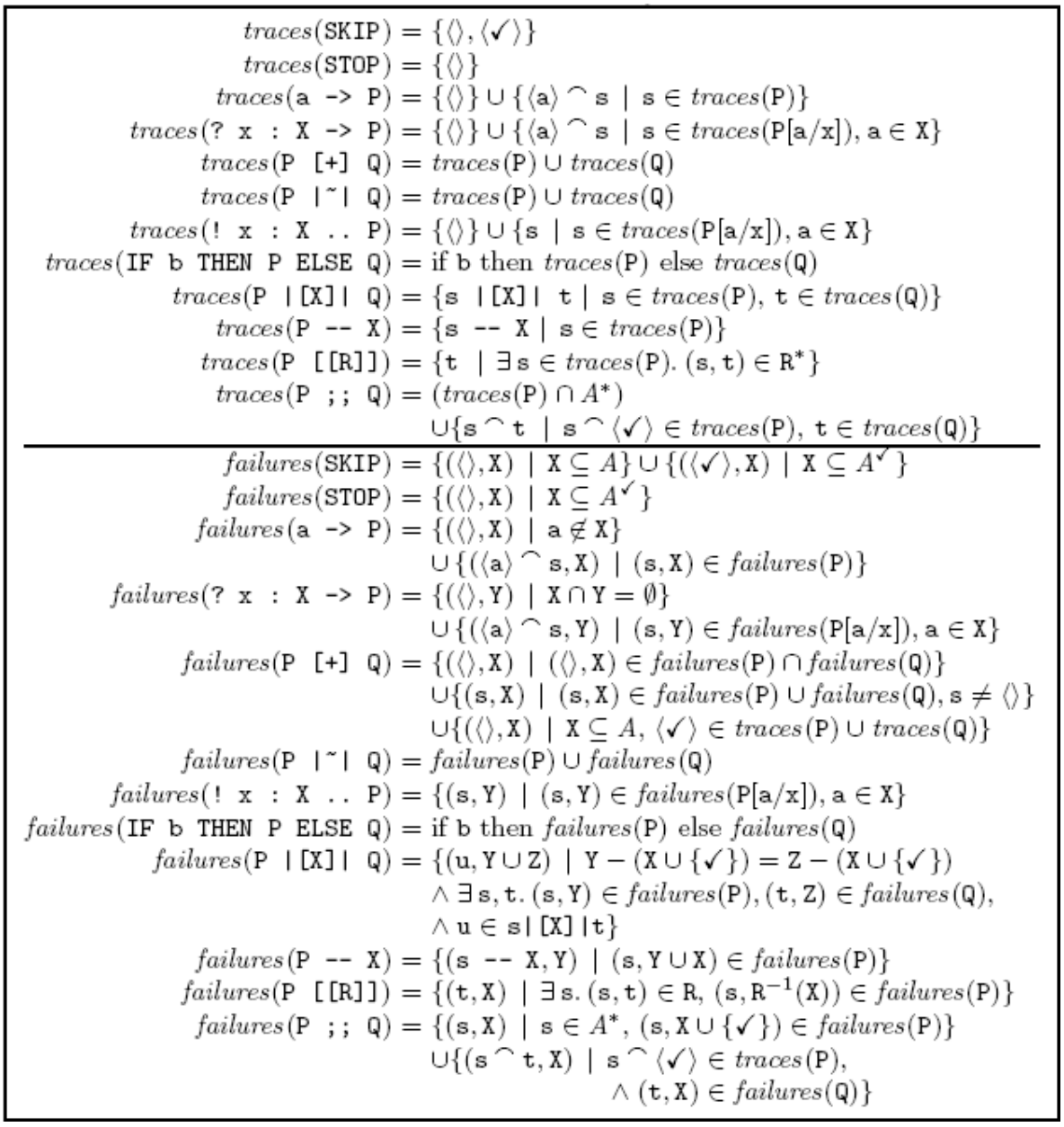
$failures^{\perp}(P) = failures(P) \cup \{ (s, X) \mid s \in divergences(P)\}$

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

$(failures^{\perp}(P) , divergences(P))$

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

A tabela abaixo (figura 5) mostra formas de construir os conjuntos que representam os modelos de traces e failures.



$$
\begin{aligned}
traces(\texttt{SKIP}) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
traces(\texttt{STOP}) &= \{\langle\rangle\} \\
traces(\texttt{a -> P}) &= \{\langle\rangle\} \cup \{\langle \texttt{a}\rangle \frown \texttt{s} \mid \texttt{s} \in traces(\texttt{P})\} \\
traces(\texttt{? x : X -> P}) &= \{\langle\rangle\} \cup \{\langle \texttt{a}\rangle \frown \texttt{s} \mid \texttt{s} \in traces(\texttt{P[a/x]}), \texttt{a} \in \texttt{X}\} \\
traces(\texttt{P [+] Q}) &= traces(\texttt{P}) \cup traces(\texttt{Q}) \\
traces(\texttt{P |~| Q}) &= traces(\texttt{P}) \cup traces(\texttt{Q}) \\
traces(\texttt{! x : X .. P}) &= \{\langle\rangle\} \cup \{\texttt{s} \mid \texttt{s} \in traces(\texttt{P[a/x]}), \texttt{a} \in \texttt{X}\} \\
traces(\texttt{IF b THEN P ELSE Q}) &= \text{if b then } traces(\texttt{P}) \text{ else } traces(\texttt{Q}) \\
traces(\texttt{P |[X]| Q}) &= \{\texttt{s} \mid\![\texttt{X}]\!\mid \texttt{t} \mid \texttt{s} \in traces(\texttt{P}), \texttt{t} \in traces(\texttt{Q})\} \\
traces(\texttt{P -- X}) &= \{\texttt{s -- X} \mid \texttt{s} \in traces(\texttt{P})\} \\
traces(\texttt{P [[R]]}) &= \{\texttt{t} \mid \exists\, \texttt{s} \in traces(\texttt{P}).\ (\texttt{s}, \texttt{t}) \in \texttt{R}^*\} \\
traces(\texttt{P ;; Q}) &= (traces(\texttt{P}) \cap A^*) \\
&\quad \cup \{\texttt{s} \frown \texttt{t} \mid \texttt{s} \frown \langle\checkmark\rangle \in traces(\texttt{P}),\ \texttt{t} \in traces(\texttt{Q})\}
\end{aligned}
$$

$$
\begin{aligned}
failures(\texttt{SKIP}) &= \{(\langle\rangle, \texttt{X}) \mid \texttt{X} \subseteq A\} \cup \{(\langle\checkmark\rangle, \texttt{X}) \mid \texttt{X} \subseteq A^{\checkmark}\} \\
failures(\texttt{STOP}) &= \{(\langle\rangle, \texttt{X}) \mid \texttt{X} \subseteq A^{\checkmark}\} \\
failures(\texttt{a -> P}) &= \{(\langle\rangle, \texttt{X}) \mid \texttt{a} \notin \texttt{X}\} \\
&\quad \cup \{(\langle \texttt{a}\rangle \frown \texttt{s}, \texttt{X}) \mid (\texttt{s}, \texttt{X}) \in failures(\texttt{P})\} \\
failures(\texttt{? x : X -> P}) &= \{(\langle\rangle, \texttt{Y}) \mid \texttt{X} \cap \texttt{Y} = \emptyset\} \\
&\quad \cup \{(\langle \texttt{a}\rangle \frown \texttt{s}, \texttt{Y}) \mid (\texttt{s}, \texttt{Y}) \in failures(\texttt{P[a/x]}), \texttt{a} \in \texttt{X}\} \\
failures(\texttt{P [+] Q}) &= \{(\langle\rangle, \texttt{X}) \mid (\langle\rangle, \texttt{X}) \in failures(\texttt{P}) \cap failures(\texttt{Q})\} \\
&\quad \cup \{(\texttt{s}, \texttt{X}) \mid (\texttt{s}, \texttt{X}) \in failures(\texttt{P}) \cup failures(\texttt{Q}), \texttt{s} \neq \langle\rangle\} \\
&\quad \cup \{(\langle\rangle, \texttt{X}) \mid \texttt{X} \subseteq A,\ \langle\checkmark\rangle \in traces(\texttt{P}) \cup traces(\texttt{Q})\} \\
failures(\texttt{P |~| Q}) &= failures(\texttt{P}) \cup failures(\texttt{Q}) \\
failures(\texttt{! x : X .. P}) &= \{(\texttt{s}, \texttt{Y}) \mid (\texttt{s}, \texttt{Y}) \in failures(\texttt{P[a/x]}), \texttt{a} \in \texttt{X}\} \\
failures(\texttt{IF b THEN P ELSE Q}) &= \text{if b then } failures(\texttt{P}) \text{ else } failures(\texttt{Q}) \\
failures(\texttt{P |[X]| Q}) &= \{(\texttt{u}, \texttt{Y} \cup \texttt{Z}) \mid \texttt{Y} - (\texttt{X} \cup \{\checkmark\}) = \texttt{Z} - (\texttt{X} \cup \{\checkmark\}) \\
&\quad \wedge\, \exists\, \texttt{s}, \texttt{t}.\ (\texttt{s}, \texttt{Y}) \in failures(\texttt{P}), (\texttt{t}, \texttt{Z}) \in failures(\texttt{Q}), \\
&\quad \wedge\, \texttt{u} \in \texttt{s} \mid\![\texttt{X}]\!\mid \texttt{t}\} \\
failures(\texttt{P -- X}) &= \{(\texttt{s -- X}, \texttt{Y}) \mid (\texttt{s}, \texttt{Y} \cup \texttt{X}) \in failures(\texttt{P})\} \\
failures(\texttt{P [[R]]}) &= \{(\texttt{t}, \texttt{X}) \mid \exists\, \texttt{s}.\ (\texttt{s}, \texttt{t}) \in \texttt{R},\ (\texttt{s}, \texttt{R}^{-1}(\texttt{X})) \in failures(\texttt{P})\} \\
failures(\texttt{P ;; Q}) &= \{(\texttt{s}, \texttt{X}) \mid \texttt{s} \in A^*,\ (\texttt{s}, \texttt{X} \cup \{\checkmark\}) \in failures(\texttt{P})\} \\
&\quad \cup \{(\texttt{s} \frown \texttt{t}, \texttt{X}) \mid \texttt{s} \frown \langle\checkmark\rangle \in traces(\texttt{P}), \\
&\qquad\qquad \wedge\, (\texttt{t}, \texttt{X}) \in failures(\texttt{Q})\}
\end{aligned}
$$

**Figura 5 - Tabela de cláusulas semânticas**

## 2.4. Refinamentos

Refinar um sistema ou parte dele consiste em propor uma nova forma de implementar ou modelar o sistema sem nenhum impacto para o restante do mesmo. Ou seja, deseja-se no mínimo manter as propriedades originais.

Isso é bastante usado em linguagens formais, a idéia é partir de um sistema formal mais abstrato e ir adicionando detalhes de implementação aos modelos refinados. Assim, temos um modelo mais próximo da implementação que satisfaz as propriedades do modelo original [8].

Relações de refinamento podem ser definidas para sistemas especificados em CSP de diversas formas, dependendo do modelo semântico que é usado. Há três formas relevantes de refinamento, correspondentes aos três modelos semânticos: refinamento por traces, falhas e por falhas e divergências [5].

**Refinamentos por traces**

Um processo P refina um outro processo Q no modelo de traces ($P \sqsubseteq_T Q$) se e somente se traces(Q) U traces(P). Basicamente isso estabelece que o processo Q (o refinado) não irá realizar seqüências de comunicações que o processo P não realiza. Ou seja, a seqüência de Q é um subconjunto daquelas comunicações.

Partindo desta definição, podemos observar que o processo STOP refina qualquer outro processo no modelo de traces (traces(STOP) = {<>}). De fato, este nível de refinamento não investiga falhas, divergências ou não-determinismos [8].

Podemos também definir a relação de equivalência entre processos através do modelo de traces conjuntamente com o conceito de refinamento [5]. Por exemplo, P é dito equivalente a Q nesta definição se e somente se

$$P =_T Q \equiv P \subseteq_T Q \wedge Q \subseteq_T P.$$

**Refinamentos por falhas**

Um processo P refina um outro processo Q no modelo de falhas ($P \sqsubseteq_F Q$) se e somente se traces(Q) $\subseteq$ traces(P) ^ failures (Q) $\subseteq$ failures (P). Essa expressão é traduzida da seguinte forma: o processo refinado Q não introduz nenhuma falha quando substituído pelo processo P no seu contexto, e também não produz nenhuma seqüência nova de comunicações.

O refinamento de processos no modelo de falhas é mais complexo de ser atingido que no modelo de traces, pois a observação de comportamentos recusáveis introduz um grau maior de complexidade. De maneira geral, entretanto, este refinamento consiste em eliminar comportamentos não-determinísticos e enfraquecer restrições [5]. Apesar disso, divergências – seqüência infinita de ações internas ao processo – não são verificadas nesse refinamento.

**Refinamentos por falhas e divergências**

Este é o refinamento mais importante de CSP. Ele é o único que permite investigar se o processo refinado não introduz mais falhas e divergências. A definição de refinamento neste modelo é dada por:

$$P \subseteq_{FD} Q \equiv failures_{\perp}(Q) \subseteq failures_{\perp}(P) \wedge divergences(Q) \subseteq divergences(P)$$

A escolha de um modo de refinamento ou outro depende de que propriedades se querem provar e também de quanto tempo deseja-se ou pode-se gastar com isso. Pois a ordem de complexidade de elaboração do refinamento cresce com a seguinte seqüência: traces, falhas e falhas-divergência. Em suma, é necessário ter em mente as tolerâncias e as restrições a serem aplicadas aos modelos ao se construir os refinamentos.

## 2.5. Ferramentas

A ferramenta mais conhecida para prova de refinamentos sobre processos CSP é o FDR (Failures and Divergences Refinament) [14]. O ProBE [15] é outra ferramenta útil para animar modelos CSP. Ambas lêem especificações CSP descritas em uma linguagem funcional chamada CSPM [14], que é um acrônimo para machine readable CSP. Esta linguagem tem sintaxe adaptada para representar as construções de CSP, com pequenas variações. Os canais de comunicação, por exemplo, devem ser explicitamente declarados, assim como o tipo dos dados que estes podem transmitir em seus eventos. É possível inclusive definir canais multidimensionais para transmitir mais de um dado simultaneamente.

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

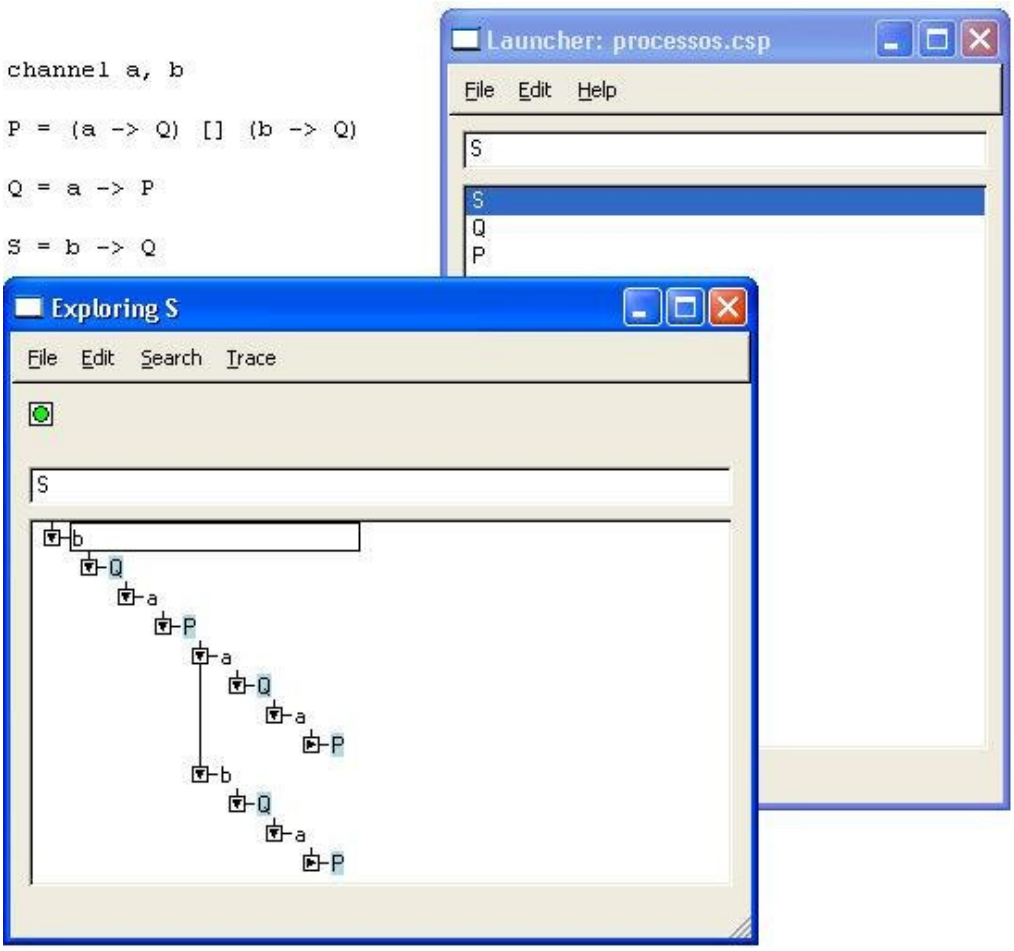



**Figura 6 - Ferramenta ProBE**

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

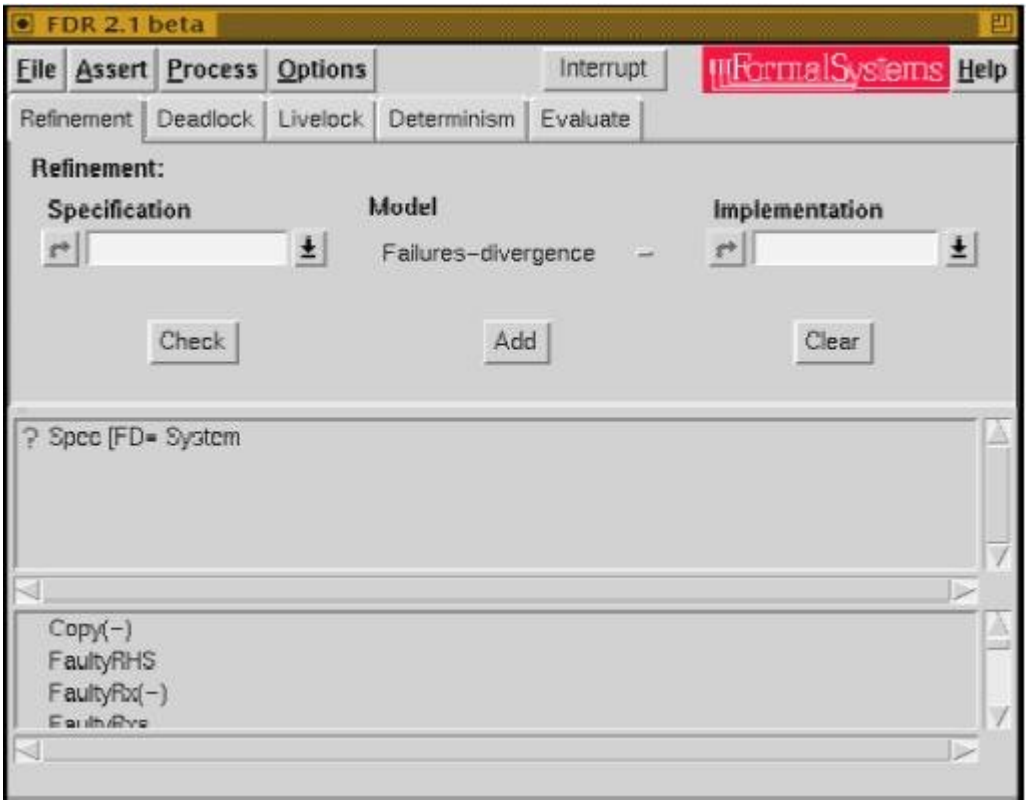



**Figura 7 - Ferramenta FDR**

# 3. Capítulo 3 – C

O surgimento da linguagem C se deu dentro dos laboratórios da AT&T Bell Labs entre 1969 e 1973 para uso sobre a plataforma do sistema operacional Unix. A linguagem – cujo mentor foi Dennis Ritchie – teve seu nome “C” dado por conta da herança de muitas características de uma outra linguagem chamada “B”, esta por sua vez criada por Ken Thompson parceiro de trabalho e contemporâneo de Ritchie [10].

O desenvolvimento de C causou uma revolução sem precedentes até então, pelo fato do aumento de produtividade e um maior nível de entendimento dos programas, o que fez com que, por volta de 1973, a maior parte do sistema operacional Unix fosse escrita em C – anteriormente era escrito em Assembly PDP-11.

## 3.1. Evolução

### K&R C

A primeira tentativa de padronização veio com a publicação do livro *The C Programming Language* – na sua primeira edição de 1978 – por Dennis Ritchie e Brian Kernighan que serviu por muitos anos como a principal referência, mesmo que informal, da linguagem [10]. A versão apresentada no livro ficou conhecida como “K&R C” – uma referência direta ao nome dos autores – e foi considerada por algum tempo como o denominador comum da linguagem conforme novos *portings* e compiladores para novas plataformas eram desenvolvidos. Tal versão era usada quando se desejava o máximo de portabilidade já que versões posteriores possuíam compatibilidade com a mesma.

Detalhes introduzidos pela versão K&R C:

- *struct data types*

Estrutura de dados a ser acessada de forma única

- *long int data type*

Os tipos de dados representando inteiros (variando de tamanho máximo de representação para cada um).

- *unsigned int data type*

Tipo de dados usado para representar número inteiro sem sinal (aumentando o poder de representação do tipo int, já que apenas a parte positiva é considerada).

- Mudança do operador =- para -= (auto-atribuição da operação de subtração) de forma a remover a ambigüidade entre *i=-10* e *i = -10*.

- inferência automática de retorno de função para inteiro, caso a função não fosse declarada antes de seu uso.

**C ANSI e C ISO**

Após o estabelecimento da primeira versão mais conhecida de C (K&R C), algumas características “não oficiais” começaram a ser incorporadas informalmente à linguagem.

Segue abaixo alguns exemplos:

- funções *void* (sem retorno) e tipos de dados *void* * (ausência de tipo explicitamente definido)

- funções que retornam tipos *struct* ou *union*

- campos de nome *struct* num espaço de nome separado para cada tipo *struct*
- atribuição a tipos de dados *struct*
- qualificadores *const* para criar um objeto só de leitura
- uma biblioteca-padrão que incorpora grande parte da funcionalidade implementada por vários vendedores
- enumerações
- um tipo de ponto-flutuante de precisão simples

Diante de tal cenário, viu-se a necessidade de padronizar a linguagem de forma a trazer uma maior compatibilidade entre os programas escritos nela, e também, pelo fato de C ter tomado a dianteira de BASIC como linguagem mais usada para microcomputadores [10].

No ano de 1983 houve a formação de um comitê – chamado X3J11 – pelo o instituto norte-americano de padrões (ANSI) com o objetivo de estabelecer uma especificação do padrão da linguagem C. A introdução de características não-oficiais apesar de largamente usadas foi o principal objetivo do processo de padronização. Todo o processo levou seis anos, sendo finalizado em 1989 e denominado ANSI X3.159-1989 "Programming Language C" [13]. A essa versão de C foi dado o nome popularmente conhecimento como “C ANSI”.

A Organização Internacional de Padrões (ISO) também adotou o padrão C ANSI mesmo com algumas pequenas modificações em 1990, sob a alcunha de ISO/IEC 9899:1990 [10]. Essa versão normalmente é chamada de “C89” ou “C90”.

**C99**

Após as padronizações feitas no final da década de 80, as especificações da linguagem C permaneceram relativamente estáticas por algum tempo – em 1995, a Normative Amendment 1 criou uma versão nova da linguagem C que corrigiu apenas alguns detalhe de C89 e adicionou um melhor suporte a caracteres internacionais [10]. Uma nova padronização foi feita em 1999 levando à publicação da norma ISO 9899:1999. O padrão foi adotado como um padrão ANSI em Março de 2000. Este padrão é normalmente chamado de "C99".

As novas características do C99 incluem:

- Funções inline
- Levantamento de restrições sobre a localização da declaração de variáveis (como em C++)
- Adição de vários tipos de dados novos, incluindo o *long long int* (para minimizar a dor da transição de 32-bits para 64-bits), um tipo de dados *boolean* explícito e um tipo *complex* que representa números complexos
- *Arrays* de dados de comprimento variável
- Suporte oficial para comentários de uma linha iniciados por “//” (como em C++)
- Várias funções de biblioteca novas, tais como snprintf()
- Vários arquivos de cabeçalho novos, tais como stdint.h
- Funções matemáticas de tipo genérico (tgmath.h)
- Suporte melhorado ao ponto flutuante da IEEE
- Suporte a macros *variadic* macros (macros de aridade variada)
- Qualificador *restrict* que permite uma melhor otimização pelo fato de permitir que apenas um ponteiro aponte para um dado

## 3.2. Elementos Sintáticos

C é uma linguagem de programação procedural, imperativa e estruturada pro blocos. Esta seção ilustrará alguns aspectos importantes de sua sintaxe, o objetivo é fazer uma ligação com uso feito no estudo de caso. Ou seja, serão mostrados aspectos sintáticos gerais, mas tendo em vista principalmente os elementos usados na implementação do sistema alvo do estudo de caso.

### 3.2.1. Tipos de Dados

#### Tipos Inteiros

Os tipos inteiros variam sob o aspecto de tamanho em suas representações, necessitando de quantidades diferentes de memória para representá-los. Os modificadores usados para diferenciar os diversos tamanhos de representação são: *short, long* e *long long*.

A amplitude de valores representados dentro de cada tipo varia bastante de plataforma para plataforma, a maneira de verificar isso é analisando o arquivo *limits.h*, nele é mostrado os valores máximo e mínimo para os tipos inteiros. A tabela 1 abaixo ilustra a variação dentre os diversos tipos.

| Implicit Specifier(s) | Explicit Specifier | Bits | Bytes | Minimum Value | Maximum Value |
|---|---|---|---|---|---|
| `signed char` | *same* | 8 | 1 | -128 | 127 |
| `unsigned char` | *same* | 8 | 1 | 0 | 255 |
| `char` | *one of the above* | 8 | 1 | −128 or 0 | 127 or 255 |
| `short` | `signed short int` | 16 | 2 | −32,768 | 32,767 |
| `unsigned short` | `unsigned short int` | 16 | 2 | 0 | 65,535 |
| `long` | `signed long int` | 32 | 4 | −2,147,483,648 | 2,147,483,647 |
| `unsigned long` | `unsigned long int` | 32 | 4 | 0 | 4,294,967,295 |
| `long long`[1] | `signed long long int` | 64 | 8 | −9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 |
| `unsigned long long`[1] | `unsigned long long int` | 64 | 8 | 0 | 18,446,744,073,709,551,615 |

**Tabela 1 - Modificadores para tipos inteiros**

O tipo *int* (sem modificadores) varia muito em sua representação entre as implementações de C, o *Single UNIX Specification* informa que o tipo *int* deve ter no mínimo 32 bits, enquanto o *ISO C* requer apenas 16 bits. O arquivo *limits.h* esclarece a limitações garantidas a estes tipos de dados [12].

| | | |
|---|---|---|
| `asm` | `enum` | `short` |
| `auto` | `extern` | `signed` |
| `_Bool` | `float` | `sizeof` |
| `break` | `for` | `static` |
| `case` | `goto` | `struct` |
| `char` | `if` | `switch` |
| `const` | `_Imaginary` | `typedef` |
| `continue` | `int` | `union` |
| `default` | `long` | `unsigned` |
| `do` | `register` | `void` |
| `double` | `restrict` | `volatile` |
| `else` | `return` | `while` |

**Tabela 2 - Palavras-chave de C**

**Booleanos**

O tipo booleano foi definido a partir do C99 (antes as condições de verdadeiro/falso era feita comparando com zero) sob a nomenclatura *_Bool* para representar o tipo *bool*, isso é definido no arquivo de cabeçalho *stdbool.h*. Quando isso não é possível geralmente são usada uma das duas alternativas:

- Definição de um "novo" tipo usando uma enumeração sob o nome "*boolean*".

```
typedef enum  _boolean { FALSE, TRUE } boolean;

...

boolean b;
```

- Definição de macros de pré-processamento.

```
#define FALSE 0

#define TRUE 1

...

int f = FALSE;
```

### 3.2.2. Controle de fluxo

**IF-ELSE**

A construção da estrutura de controle *if-else*, é dada da seguinte forma:

```
if (<expression>)
  <statement1>
else
  <statement2>
```

É feita a comparação se <expression> nos parênteses é diferente de zero (true), então o controle é passado a <statement1>. Caso exista (a existência é opcional) uma cláusula *else* e ocorra de <expression> ser zero (false), o controle será passado a <statement2>. Por questão de qualidade e legibilidade do código é usado identações adequadas (a sentença *else* ficar no mesmo nível de tabulação do *if*) e o uso de chaves para ficar claro o escopo o bloco de cada um.

**SWITCH**

A construção de uma estrutura de controle com um *switch* faz com que a seqüência de execução seja transferida para algumas das sentenças seguintes, baseando-se no valor de retorno (inteiro) da expressão analisada. Normalmente usa-se a estrutura do *switch* quando se pode prever de antemão os possíveis valores de retorno da expressão, daí vem o uso de *labels* (usa-se nomes intuitivos para cada uma das possíveis condições) para identificar os possíveis fluxos de execução. O exemplo abaixo mostra o uso geral do switch:

```
switch (<expression>) {
  case <label1> :
    <statements 1>
  case <label2> :
```

```
        <statements 2>

        break;

      default :

        <statements 3>

   }
```

Outros detalhes bastante particulares no uso do *switch* correspondem a boas práticas de codificação. O primeiro deles é o uso do *break* ao final de cada fluxo de execução de cada *label,* isso faz com que se defina bem o escopo da execução de cada um deles (caso contrário, ocorre um caso chamado "*fall through*" em que sentenças subseqüentes são executadas) e ao término o fluxo de execução caia fora da estrutura do *switch*. Outro detalhe é o uso do *label default*, pois ele corresponde à execução padrão da estrutura *switch*, mesmo que nenhuma das condições seja alcançada ao menos ele será, muitas vezes a condição *default* está associado a condições de exceção.

### 3.2.3. Funções

Uma definição de uma função na linguagem C consiste de um tipo de retorno (*void* se não há retorno), um nome único, uma lista de parâmetros dentro de parênteses (*void* se não há nenhum), em seguida, uma seqüência de comandos dentro de um bloco delimitado por chaves. Funções que especificam um tipo de retorno devem possuir ao menos um comando *return*.

Segue abaixo um esquema geral para composição de uma função:

```
<return-type> functionName( <parameter-list> )
 {
    <statements>
```

```
    return <expression of type return-type>;
}
```

Onde *<parameter-list>*:

```
<data-type> var1, <data-type> var2, ... <data-type> varN
```

Outra forma de definir função é através de um ponteiro para a mesma usando a seguinte construção:

```
<return-type> (*functionName)(<parameter-list>);
```

Exemplo:

```
#include <stdio.h>

int (*operation)(int x, int y);

int add(int x, int y)
{
   return x + y;
}

int subtract(int x, int y)
{
    return x - y;
}

int main(int argc, char* args[])
{
   int foo = 1, bar = 1;
   operation = add;
   printf("%d + %d = %d\n", foo, bar, operation(foo, bar));
   operation = subtract;
   printf("%d - %d = %d\n", foo, bar, operation(foo, bar));
   return 0;
}
```

# 4. Capítulo 4 – ESTUDO DE CASO

A intenção do capítulo corrente é mostrar passos para que se extraía de forma sistemática a especificação em CSP de um componente implementado na linguagem C, o componente-alvo é um controlador de portas de um metrô.

O primeiro passo foi analisar de forma geral as intenções e objetivos do sistema através de uma breve descrição do mesmo. As funcionalidades escolhidas são descritas de acordo com o documento de requisitos do sistema e serão detalhadas mais adiante.

Em seguida, será mostrado o padrão identificado durante a análise da codificação das funcionalidades do sistema, para que se estabeleça uma base para mapeamento entre as linguagens. Alguns ajustes sintáticos através de refatorações do código são necessários à implementação de forma a se facilitar tanto a codificação quanto a extração da modelagem.

Os mapeamentos desenvolvidos são mostrados de forma abstrata e geral, mas são aplicados em cima de uma estrutura pré-estabelecida que foi pensada voltando-se para o contexto do sistema da Controladora Geral de Portas. Por fim, se mostrará a aplicação das técnicas desenvolvidas em cima do código de implementação do sistema.

## 4.1. Sobre o sistema: Controladora Geral de Portas

O sistema implementado usando a linguagem C faz parte do equipamento de Controle Geral de Portas AeS-0617, armazenado e processado por um

microcontrolador da linha PIC da família 18F da Microchip. A intenção é que o sistema entre em operação no projeto Santiago Linha 2, Chile.

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

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

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

### 4.1.1. Casos de uso escolhidos

As descrições de casos de uso abaixo foram extraídas do documento de requisitos do sistema SRS-0617-1 Controle Geral de Portas [16].

**Função abre portas (CML)**

Caso o trem estiver em CML (Comando Manual), com seleção de cabine líder, chave seletora de lado (ST) habilitada e chave de abertura “KLOAN” (chave responsável por iniciar o processo de abertura de portas) selecionada em modo de preparação será habilitado um flag de temporização de 10s. Caso a velocidade estiver abaixo de 6Km/h ou diminua de 6Km/h dentro desse intervalo e todas as condições anteriores permaneçam inalteradas, será executada abertura de portas no respectivo lado selecionado e o flag de temporização zerado. Caso o operador selecione a chave “KLOAN” em modo de despreparação dentro do período de 10s e antes do trem atingir velocidade inferior a 6Km/h o flag também será zerado. Caso o operador mude a posição da chave de seleção de lado de abertura após a abertura de portas, estas se fecharão sem a necessidade de comando.

**Função fecha portas (CML – prioritária em relação à abertura)**

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

## 4.2. Identificação de padrões

A implementação do sistema encontra-se estruturada de forma que o componente CGP (Controladora Geral de Portas) comporta-se como uma

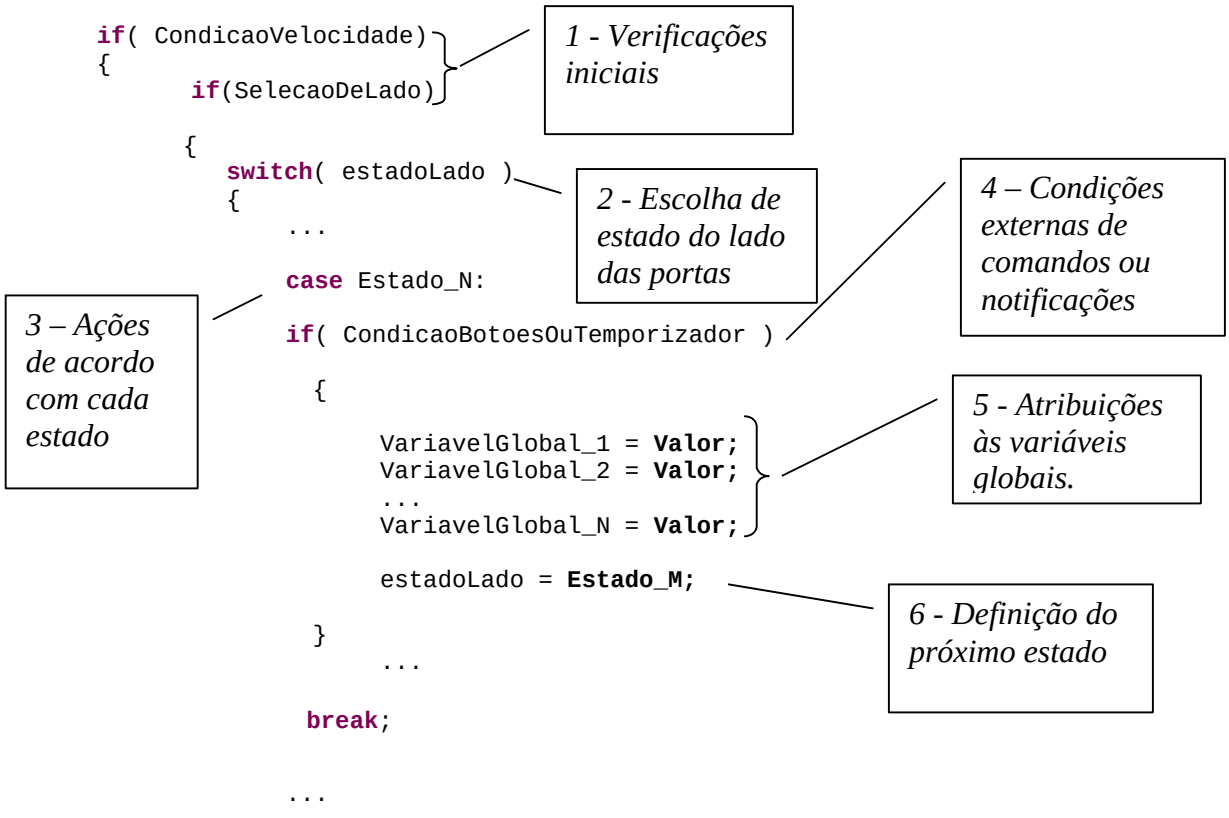
máquina de estados de acordo com lado das portas a ser operado. Dentro de cada estado são feitas atribuições em variáveis globais que podem ser visualizadas tanto pela própria CGP como pelos todos os componentes do sistema. Tais atribuições estabelecem o acionamento de componentes periféricos, como portas e LEDs de sinalização.

O trecho de código original abaixo ilustra um comportamento recorrente da CGP:

```
...
if( VR6 == TRUE )/*   se a velocidade for menor que 3Km/h, faça:*/
{
       if(STDIR == TRUE)/*se chave de seleção de lado direito for */
                    /*        acionada, faça:                    */
      {
         switch( estado_dir )
             /*verifica qual o estado de "estado_dir"*/
            /*para a máquina de estados de operação */
            /*do lado direito do trem:*/
         {
            case 0:/* caso no estado "0"(inicial), faça:     */
               if( ( PREP1 == TRUE ) &&
                    ( BFDC == FALSE ) &&
                    ( BFDL == FALSE ) )
                    /*se um dos botões de abertura for*/
                    /*apertado e nenhum de fechamento, faça:*/
               {
                    /*estado inicial de portas fechadas*/
                    /*do lado esquerdo em CML*/
                  est_fch_cml_d = TRUE;
                    /*estado inicial de portas fechadas   */
                  est_fch       = TRUE;
                    /*ativa o flag de comando de abertura */
                    /* recebido */
                  cmd_fase_dir  = ABRE;
                    /* vai para o estado 1 */
                  estado_dir     = 2;
                    /* não executa comando de gongo*/
                  CMD_GONGO     = FALSE;
                  output_bit(PIN_B2,0);
               }
                    /* se não, faça: */
                    else
               {
                    /* permanece no estado 0 */
                  estado_dir = 0;
               }

                     break;

            ...
```

Após uma análise detalhada do código, foi verificado que as avaliações das variáveis contidas nas expressões *if-else* e *switch* são, em geral, comunicações "de fora para dentro" da CGP – entenda-se por isso que valores trafegados por esses canais serão recebidos através de *input* pela CGP. Enquanto as atribuições contidas dentro das cláusulas mencionadas anteriormente corresponderão por comunicações "de dentro para fora" da CGP – correspondendo ao sentido inverso ao mencionado anteriormente, ou seja, serão enviados através de *output* pela CGP.

O padrão geral identificado foi o seguinte:




Diante de tal cenário decidiu-se estabelecer um padrão de nomenclatura que irá guiar a construção de canais, bem como indicar o sentido de suas comunicações. Na seção 4.2.1 será mostrada essa convenção de nomenclatura,

como também a reorganização do código de forma que toda atribuição às variáveis seja feita antes da indicação do próximo estado.

### 4.2.1. Refatoração do código

As refatorações foram feitas de forma a adequar sintaticamente (sem alteração semântica) a estrutura da função para uma seqüência padrão de mapeamentos. Abaixo são mostradas algumas ações tomadas com uma breve justificativa em seguida:

- Padronização dos nomes das variáveis globais que expressam comunicações entre entidades distintas: <ProcOrigem>_<verbo>EntidadeAcionada_<ProcDestino>;

  A intenção foi seguir uma padronização de forma que o desenvolvedor ao implementar o sistema deixe claro o sentido das comunicações que deseja estabelecer, bem como facilitar o mapeamento. É bem verdade que algumas comunicações serão bidirecionais então nesse caso se escolhe o sentido de *output* da CGP.

- Mudança da assinatura da função trabalho_CML_STL2, alterando seu identificador para o nome do processo a ser extraído CGP_CML.

  A mudança de nome foi feita de forma a se extrair o nome real de quem possui tal comportamento, CGP neste caso.

- Declaração explícita de constantes e das variáveis globais, nessa ordem respectivamente, antes da função, de forma a se construírem as constantes e os canais usados para especificação em CSP;

As especificações CSP, geralmente seguem a ordem: definição de canais e constantes, para em seguida se construir os processos. Além de seguir a convenção, a intenção foi tipar as variáveis e constantes, que se tornarão canais mais adiante. Definiu-se também o tipo enumerado Comando que é usado por algumas variáveis dentro da implementação.

## 4.3. Mapeamento de C para CSP

Agora serão mostradas as estruturas identificadas na implementação em C do componente CGP para em seguida se estabelecer a correspondência em CSP.

Antes de serem mostrado os mapeamentos são estabelecidas algumas considerações por conta do escopo da implementação do sistema e para um melhor entendimento dos mapeamentos.

Considerações:

- Por “bloco” entenda-se um conjunto de construções de qualquer estrutura explicada no Capítulo 3: atribuição a variável, cláusulas *if-else*, cláusulas *switch* até chamada de métodos.
- Os tipos de dados entre as atribuições e os valores são considerados corretos.
- Comentários de código são desconsiderados.
- Todas as variáveis são consideradas globais (visíveis).
- Um processo MEMORIA construído manualmente surgiu pelo fato do sistema ser distribuído e que em CSP cada componente possui sua

memória própria e o elemento analisado fazer acesso a variáveis globais fora dele.

- A chamada à função output_bit(pino, valor) no código da implementação foi abstraída como parte da comunicação CGP_executarComandoGongo_MEMORIA, já que “valor” enviado ao pino PIN_B2 sempre corresponde ao valor atribuído à variável mencionada acima. Em suma, não haverá mapeamento para chamada de função.

Os mapeamentos mostrados foram construídos baseando-se no entendimento do sistema (comportamento, entidades envolvidas e comunicações) e da análise sintática e semântica da codificação. As regras desenvolvidas trazem no primeiro membro uma representação abstrata de um trecho de código em C e o segundo membro – após o símbolo “~>” – seu correspondente em CSP destacado em negrito indicando seqüência de mapeamentos.

**Regra 1:** Definição de função:

AvaliaFuncao[

```
void IdentificadorFuncao (void)
{
        Bloco_funcao;
}
```

]
~>
***IdentificadorFuncao* =** AvaliaBloco[Bloco_funcao]

Ex:

AvaliaFuncao[ void Sender (void){

resultado = 10;

} ]

~>

**Sender =** AvaliaBloco[resultado = 10;]

**Regra 1.1:** Bloco:

AvaliaBloco [bloco]

~>

AvaliaListAtribuicaoValor [listAtr]

AvaliaListIfElse[listIfElse]

AvaliaListSwitch[listSwitch]

AvaliaBloco[bloco \ {listAtr U listIfElse U listSwitch}]

Essa função é aplicada inúmeras vezes por sua recursão até que se consuma todo o bloco considerado, pois nem sempre os bloco estão estruturados dessa forma: lista de atribuições e lista de *if-else´s*. Há variações onde um aparece mais que o outro ou nem aparece. As definições de AvaliaListAtribuicao e AvaliaListIfElse são mostradas mais adiante.

**Regra 2:** *If-else* :

AvaliaIfElse[

if(listExpBool)

listExpBool é da forma:

*processoOrigem1_verboEntidade1_processoDestino1*
*operadorBooleano1*
*processoOrigem2_verboEntidade2_processoDestino2*
*operadorBooleano2*
*...*
*operadorBooleanoN-1*
*processoOrigemN_verboEntidadeN_processoDestinoN*

```
        {
           Bloco_if;
        }
        else
        {
           Bloco_else;
        }

        Bloco_seguinte
]
```

Corresponde ao restante do bloco no qual está inserido o *if-else*

**~>**

***processoOrigem1_verboEntidade1_processoDestino1? verbo1Passado →***
...
***processoOrigemN_verboEntidadeN_processoDestinoN? verboNPassado →***

***if (***AvaliaListExpBool[listExpBool]***) then***
*AvaliaBloco[Bloco_if]* → ***SKIP;***AvaliaSeqBloco [Bloco_seguinte]
***else***
*AvaliaBloco[Bloco_else]* → ***SKIP;***AvaliaSeqBloco [Bloco_seguinte]

Na Regra 2.1 a seguir, é considerada uma estrutura (Bloco_superior) não mostrada acima que é a seqüência de sentenças seguintes ao bloco superior ao if-else avaliado. O exemplo mostrado após a Regra 2.1 ilustra isso.

Por questões de modularização quando um *if-else* possuir 3 níveis ou mais níveis de hierarquia dentro dele – outros if-else´s ou switch – e este *if-else* não possuir um processo imediatamente superior  decidiu-se isolar tal *if-else* em um sub-processo identificado pela variável que o define. Então, no lugar do *if-else* considerado no trecho de código original será chamado o sub-processo criado com ele com o operador seqüencial ";", em seguida para continuar o fluxo de execução do bloco no qual ele estava inserido.

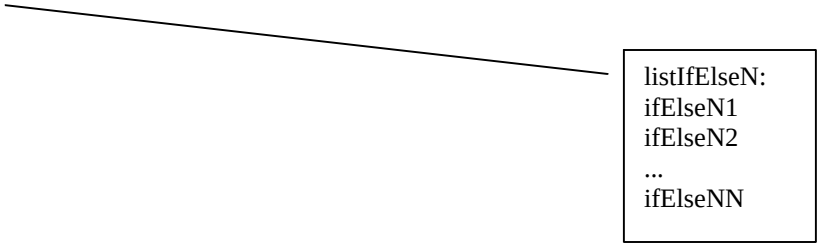
listAtrib1:
Id11 = valor11;
Id12 = valor12;
...
Id1N = valor 1N;

**Regra 2.1:** Blocos seguinte ao if-else:

AvaliaSeqBloco[Bloco_seguinte]
**~>**
AvaliaListAtribuicaoValor[listAtrib1]
AvaliaListIfElse[listIfElse1] AvaliaSeqBloco[Bloco_superior]

listIfElse1:
ifElse11
ifElse12
...
ifElse1N

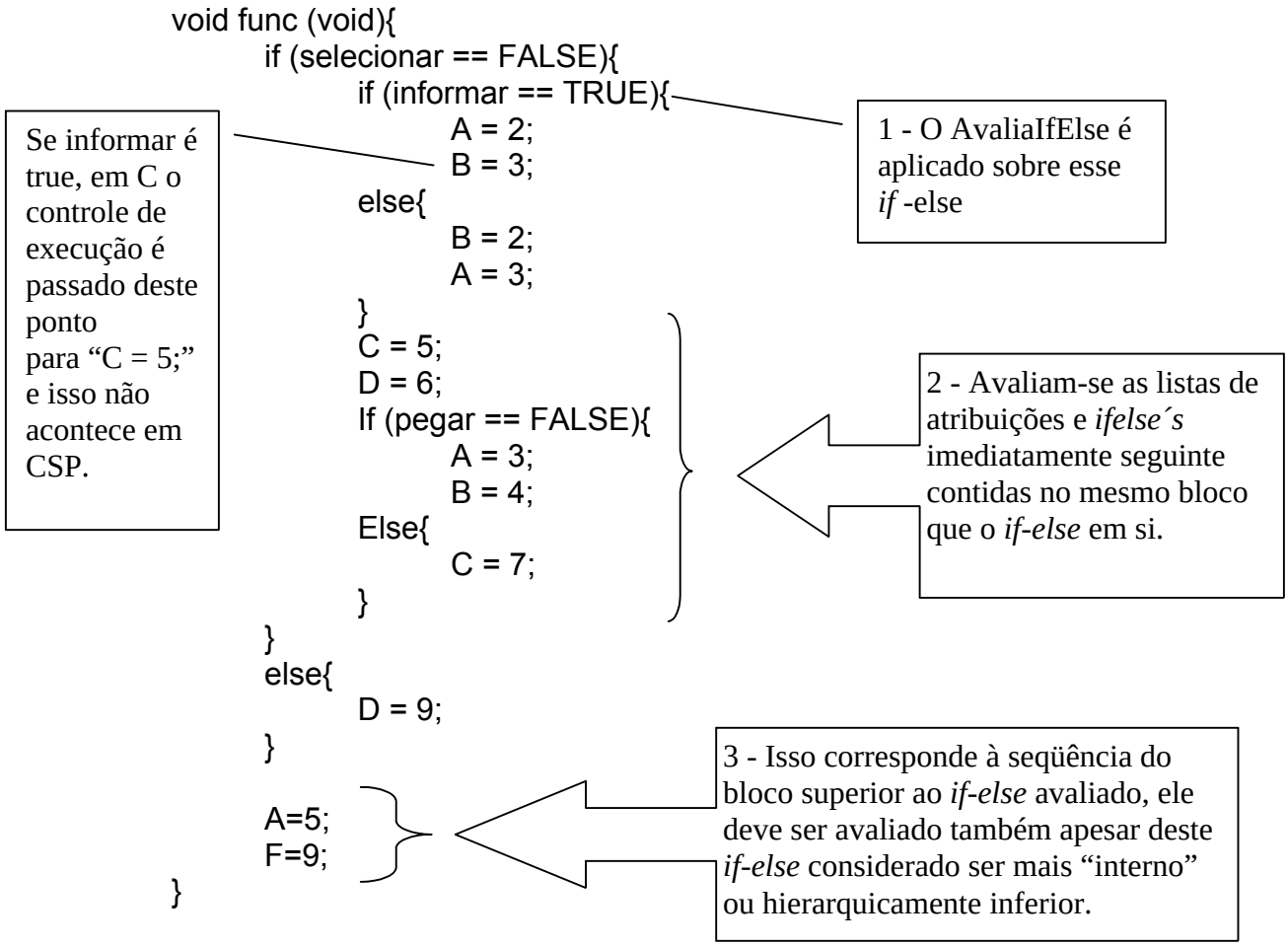
...
AvaliaListAtribuicaoValor [listAtribN]
AvaliaListIfElse[listIfElseN] AvaliaSeqBloco[Bloco_superior]




A primeira parte dessa avaliação foi replicada N vezes para que se consuma todo o bloco de mesmo nível do *if-else* considerado na Regra 2, e foi feita inúmeras vezes pelo fato de bloco não possuir estrutura única – mas é sempre composta por atribuições e outros *if-else´s*. A aplicação é feita até que não reste nenhuma estrutura – atribuição ou outro *if-else* – no mesmo nível que o *if-else* considerado.

Quando o bloco no qual o *if-else* considerado está inserido for completamente consumido, avalia-se a seqüência de sentenças seguintes ao bloco imediatamente superior, no caso chamado de Bloco_superior. A base da função AvaliaSeqBloco é quando se chega ao último nível de bloco – final da função ou processo –, então no caso corresponde a quando for a avaliação é feita sobre vazio e retorna SKIP.

Isso deve ser feito pelo fato de um *if-else* aninhado dentro de outro em C possuir a “volta” do controle do fluxo de execução ao bloco imediatamente superior ao *if-else* e deste para o seu superior imediato também e assim sucessivamente até que se chegue ao último nível, no caso função para este mapeamento. Essa “volta” não possui equivalência em CSP, então deve-se fazer uma “colagem” de comportamentos extraídos posteriores (no mesmo nível do *if-else* e no bloco superior) neste *if-else* mais interno. O exemplo abaixo ilustra a necessidade desta regra:




A consideração dessas sentenças e estruturas posteriores se deve ao fato de a estrutura *if-else* em CSP não existir a passagem do controle de execução ao bloco seguinte ao mesmo, e sim continuando o comportamento do processo no qual ele está inserido. Ou seja, deve-se mapear as sentenças seguintes ao *if-else* avaliado para dentro do fluxo do *if* e também do *else* sobre o qual foi aplicado a Regra 2.

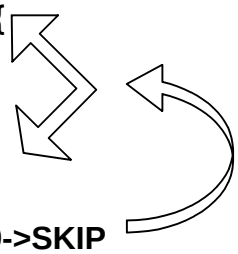
**Regra 2.2:** Lista de if-else:

AvaliaListIfElse[listIfElse]
~>
Avalia[ifElse1];AvaliaListIfElse[listIfElse \ ifElse1]

Ex: É considerado que a Regra 1 já tinha sido aplicada à definição da função.

```
void func (void){
        If (informar == TRUE){

                A = 2;
                B = 3;
                If (selecionar == FALSE){
                        C = 1;
                }
                else {
                        D = 4;
                }

                D = 7;


        }
        Else {
                A = 1;
                B = 2;
        }
        A = 9;
        B = 9;
}
~>
func =
AvaliaIfElse[
        If(informar == TRUE){ A = 2;...}
        Else {A=1;...}

]
~>
func =
informar?informou->
if (informou == true) then
        AvaliaBloco[
        A = 2;
        B = 3;
        If (selecionar == FALSE){
                C = 1;
        }
        else {
                D = 4;
        }
```

```
        D = 7;
        ] -> SKIP;AvaliaSeqBloco [A = 9;B = 9;]
else
        AvaliaBloco[
        A = 1;
        B = 2;
        ] -> SKIP;AvaliaSeqBloco [A = 9;B = 9;]
~>
func =
informar?informou->
if (informou == true) then
        AvaliaListAtribuicaoValor[
        A = 2;
        B = 3;
        ]
        AvaliaListIfElse[
        If (selecionar == FALSE){
                C = 1;
        }
        else {
                D = 4;
        }
        ]
        AvaliaListAtribuicaoValor[
        D = 7;
        ] -> SKIP;AvaliaListAtribuicao[A=9;B=9;] AvaliaSeqBloco []
else
        A.1 -> B.2-> SKIP; AvaliaListAtribuicao[A=9;B=9;]AvaliaSeqBloco[]
~>
func =
informar?informou->
if (informou == true) then
        A.2 -> B.3->
        AvaliaIfElse[
        If (selecionar == FALSE){
                C = 1;
        }
        else {
                D = 4;
        }
        ] ->D.7-> SKIP;A.9 ->B.9->SKIP
else
A.1 -> B.2-> SKIP; A.9 ->B.9->SKIP
~>
func =
informar?informou->
if (informou == true) then
```



Base da recursão, pois o bloco superior é função e não possui mais sentenças após o último if considerado e retorna SKIP

A seqüência detectada para o bloco superior é “colada” em seus internos

```
        A.2 -> B.3->
        selecionar?selecionou->
        if(selecionou == false) then
           AvaliaBloco[C = 1;] ->SKIP; ->D.7-> SKIP;A.9 ->B.9->SKIP

        else
           AvaliaBloco[D = 4;] ->SKIP; ->D.7-> SKIP;A.9 ->B.9->SKIP

else
A.1 -> B.2-> SKIP; A.9 ->B.9->SKIP
~>
func =
informar?informou->
if (informou == true) then
        A.2 -> B.3->
        selecionar?selecionou->
        if(selecionou == false) then
                C.1->SKIP; ->D!7-> SKIP;A.9 ->B.9->SKIP


        else
                D.4 ->SKIP; ->D.7-> SKIP;A.9 ->B.9->SKIP

else
A.1 -> B.2-> SKIP; A.9 ->B.9->SKIP
```

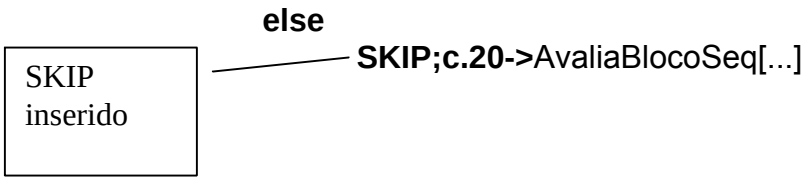
Caso ocorra algum *if* sem a cláusula *else*, uma cláusula *else* é criada em CSP e inserido um processo SKIP com operador seqüencial após ela e o mapeamento restante é feito normalmente.

Ex:

```
}...
        If (informar==TRUE){
                        a = 10;
        }
        c = 20;
}...
```

Quando avaliado vira:

```
Informar?informou -> if (informou == true) then
                                a.10 -> c.20 ->SKIP;AvaliaBlocoSeq[...]
```




**Regra 3:** Definição de variáveis:

AvaliaDefinicaoVariavel[ tipo identificador; ]
~>
***channel identificador:***AvaliaTipo[tipo]

Ex:
AvaliaDefinicaoVariavel[int informarValor;]
~>
***channel informarValor:*** AvaliaTipo[int]
~>
***channel informarValor:Int***

**Regra 4:** Avaliação de tipos de dados:

Os tipos envolvidos são inteiros (int), booleanos (bool) e Comando (definido durante a implementação).

Mapeamentos atômicos:

AvaliaTipo[int]
~>
***Int***

AvaliaTipo[bool]
~>
***Bool***

AvaliaTipo[Comando]
~>
***Comando***

**Regra 5:** Avaliação de valores:

Os valores envolvidos são inteiros (int), booleanos (bool) e Comando.

Mapeamentos atômicos:

- Int:

AvaliaValor[1]
~>
***1***

**...**

AvaliaValor[n]
~>
***n***

- bool:

AvaliaValor[TRUE]
~>
***true***

AvaliaValor[FALSE]
~>
***false***

- Comando:

AvaliaValor[FECHA]
~>
***FECHA***
***...***

AvaliaValor[PAUSA]
~>
***PAUSA***

- Constante:

AvaliaValor[const]
~>
***const***

Para as regras 4 e 5 é assumido que trabalha-se em cima de C99, como mencionado no capítulo 3, esta versão de C possui o tipo booleano definido.

**Regra 6:** Definição de tipos enumerados:

AvaliaEnum[

typedef enum enum_tipo {enum1, enum2, ... , enumN} Tipo;

]
~>
***datatype Tipo = enum1 | enum2 | ... | enumN***

**Regra 7:** Definição de constantes:

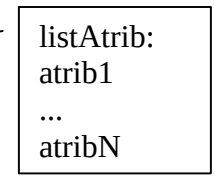
AvaliaConstante[ const tipo identificador = valor; ]
~>
***identificador =*** AvaliaValor[valor]

Nota-se aqui uma sutil diferença entre as linguagens quanto à definição de constantes, enquanto C necessita um tipo explícito, CSP é fracamente tipada.

**Regra 8:** Atribuição de valor:

AvaliaAtribuicaoValor [ identificador = valor; ]
~>
***identificador.*** AvaliaValor[valor] →

**Regra 8.1:** Lista de atribuições de valor:




AvaliaListAtribuicaoValor[listAtrib]
~>
AvaliaAtribuicaoValor[atrib1]
AvaliaListAtribuicaoValor[listAtrib \ atrib1]

Caso listAtrib seja vazia e não haja nenhuma regra a ser aplicada  após ela é retornado **SKIP**.

Ex:

```
int Teclado_enviarValor_Calculadora;
int Tela_enviarImagem_Teclado;
void Teclado (void){
        Teclado_enviarValor_Calculadora = 1;
        Tela_enviarImagem_Teclado = 2;
}
~>
AvaliaDefinicaoVariavel[int Teclado_enviarValor_Calculadora;]
AvaliaDefinicaoVariavel[int Tela_enviarImagem_Teclado;]
AvaliaFuncao[
void Teclado (void){
Teclado_enviarValor_Calculadora = 1;
Tela_enviarImagem_Teclado = 2;
}]
~>
channel Teclado_enviarValor_Calculadora: Int
channel Teclado_enviarValor_Tela: Int
Teclado =
AvaliaBloco[
Teclado_enviarValor_Calculadora = 1;
Tela_enviarImagem_Teclado = 2;
]
```

```
~>
channel Teclado_enviarValor_Calculadora: Int
channel Teclado_enviarValor_Tela: Int
Teclado =
        AvaliaListAtribuicaoValor [
        Teclado_enviarValor_Calculadora = 1;
        Tela_enviarImagem_Teclado = 2;
        ]
        AvaliaListIfElse[]
~>
channel Teclado_enviarValor_Calculadora: Int
channel Teclado_enviarValor_Tela: Int
Teclado =
AvaliaAtribuicaoValor [ Teclado_enviarValor_Calculadora = 1;]
AvaliaAtribuicaoValor [Tela_enviarImagem_Teclado = 2;]
 ~>
channel Teclado_enviarValor_Calculadora: Int
channel Teclado_enviarValor_Tela: Int
Teclado = Teclado_enviarValor_Calculadora. 1 - >
 Tela_enviarImagem_Teclado. 2 -> SKIP
```

Última atribuição da lista sem regra alguma a ser aplicada depois

**Regra 9:** Expressões booleanas:

```
AvaliaListExpBool[listExpBool]
~>
```

listExpBool:
exp1 operador1
exp2 operador2
...
expN

AvaliaExpBool[exp1] AvaliaOperadorBool[operador1]
AvaliaListOperadorBool[listExpBool \ exp1]

**Regra 9.1:** Expressões booleanas unitárias:

AvaliaExpBool[ProcOrigem_verboEntidade_PocDestino == b]
~>
***verboPassado* ==** AvaliaValor[b]

ou

AvaliaExpBool[ProcOrigem_verboEntidade_PocDestino != b]
~>
***verboPassado !=*** AvaliaValor[b]

É assumido que os canais estão seguindo o padrão nomenclatura mostrado acima. Caso verboPassado já exista de algum canal extraído anteriormente, concatena-se um número seqüencial ao mesmo. Ex: se o verbo no passado, “informou” já existe e aparecer novamente pelo mapeamento a segunda ocorrência é identificado por “informou2” e assim sucessivamente.

**Regra 9.2:** Operadores booleanos:

AvaliaOperadorBool [ && ]
~>
***and***

ou

AvaliaOperadorBool [ && ]
~>
***or***

A Regra 10.1 corresponde à base da recursão para a regra 10, pois quando a lista de expressões booleanas for unitária, a regra AvaliaExpBool será aplicada, pois ela trata expressões simples.

Ex:

AvaliaListExpBool[R_selecionarChave_S != 0 && P_informaValor_Q == TRUE] ~>
AvaliaExpBool[R_selecionarChave_S != 0] AvaliaOperador[&&] AvaliaListExp[P_informaValor_Q == TRUE]
~>

Condição para aplicação da base da recursão

***selecionou != 0*** AvaliaValor[0] ***and*** AvaliaExpBool[P_informaValor_Q == TRUE]

~>

***selecionou != 0 and informou ==*** AvaliaValor[TRUE]

~>

***selecionou != 0 and informou == true***

**Regra 10:** Programa:
AvaliaPrograma [ programa ]
~>
AvaliaListaConst[listConst]
AvaliaListaDefinicaoVariaveis[listVar]
AvaliaListaFuncoes[listFunc]

Programa:
ListaConstantes
ListaVariaveisGlobais
ListaFuncoes

As regras descritas acima são generalizações que irão iniciar o processo de mapeamento. Cada avaliação de lista descrita como resultado acima corresponde à aplicação de sua regra unitária à primeira ocorrência unitária de cada lista. E quando a lista for vazia ele retorna vazio também.

**Regra 10.1:** Lista de constantes:

AvaliaListaConst[listConst]
~>
AvaliaConstante[const1]
AvaliaListaConst[listConst \ const1 ]

listConst:
const1;
const2;
...
constN;

**Regra 10.2:** Lista de definição de variáveis:

AvaliaListaDefinicaoVariaveis [listVar]
~>
AvaliaDefinicaoVariavel[var1]

listVar:
var1;
var2;
...
varN;

AvaliaListaDefinicaoVariaveis [listVar \ var1 ]
AvaliaDefinicaoEnum[enum]

**Regra 10.3:** Lista de funções:

AvaliaListaFuncoes [listFunc]
~>
AvaliaFuncaol[func1]
AvaliaListaFuncoes [listFunc \ func1]

listFunc:
void func1
void func2
...
void funcN

**Regra 11:** Switch:

```
AvaliaSwitch[
Switch (estadoLado){
Case 0:
        Bloco_0;
        Break;
...
Case N:
        Bloco_N;
        break;
Default:
        Bloco_default;
        break;
}
]
~>
```

**estadoLado.0 ->** AvaliaBloco[Bloco_0]
**[ ]**
**...**
**estadoLado.N ->** AvaliaBloco[Bloco_N]
**[ ]**
**estadoLado.N+1 ->** AvaliaBloco[Bloco_default]

Aqui é feito um mapeamento de valor fora do escopo (default) para N + 1

**Regra 11.1:** Lista de Switch:

AvaliaListSwitch[ listSwitch]
~>
AvaliaSwitch[switch1]
AvaliaListSwitch[listSwitch \ switch1]

listSwitch:
Switch1
...
SwitchN

## 4.4. Aplicação do mapeamento sobre implementação da CGP

A discussão aqui se limitará a aplicações pontuais dos mapeamentos já que muitas vezes o mesmo processo será bastante repetido, então a intenção aqui é cobrir o maior número de classes de mapeamentos. O resultado completo [18] dos mapeamentos realizados nesta seção será usado apenas para motivo de análise de propriedades.

A estrutura geral do código é a seguinte:

Declaração de constantes
Declaração de Variáveis Globais
Definição de Função
    Cláusula if-else velocidade
        Cláusula if-else lado direito
            Cláusula switch
            Cases
                Cláusulas if-else Botões ou Comandos
                    Atribuições Variáveis Globais
        Clausula if-else lado esquerdo
            Cláusula switch
            Cases
                Cláusulas if-else Botões ou Comandos
                    Atribuições Variáveis Globais

Algumas variações dessa estrutura ocorrem dentro das cláusulas *case*, mas pode-se considerar essa mostrada acima suficiente para prover uma visualização.

A aplicação é iniciada através do mapeamento AvaliaPrograma e nele é submetido todo código da implementação que por sua vez irá dar origem aos

mapeamentos: AvaliaListConstant, AvaliaListaDefinicaoVariaveis e AvaliaListFuncoes, respectivamente.

As aplicações das regras mostradas nas seções subseqüentes são consideradas individualmente, mas do ponto de vista de extração da especificação deve-se considerar que uma anterior é concatenada com a seguinte.

### 4.4.1. Extração de constantes

O primeiro trecho de código é a declaração de constantes e para se extrair a especificação em CSP do mesmo basta aplicar a função AvaliaListConstant.

```
const bool TRUE = true;
const bool FALSE = false;
const int CNT_COUNT1 = 7;
const int CNT_COUNT3 = 7;
const int CNT_COUNT2 = 7;
```

Aplicando-se a regra mencionada anteriormente corresponderá a aplicações individuais de AvaliaConstante:

AvaliaConstante [`const bool TRUE = true;`]
AvaliaConstante [`const bool FALSE = false;`]
AvaliaConstante [`const int CNT_COUNT1 = 7;`]
AvaliaConstante [`const int CNT_COUNT3 = 7;`]
AvaliaConstante [`const int CNT_COUNT2 = 7;`]
~>
**TRUE =** AvaliaValor [true;]
**FALSE =** AvaliaValor [false;]
**CNT_COUNT1 =** AvaliaValor [7;]
**CNT_COUNT3 =** AvaliaValor [7;]
**CNT_COUNT2 =** AvaliaValor [7;]
**~>**
**TRUE = true**
**FALSE = false**
**CNT_COUNT1 = 7**
**CNT_COUNT3 = 7**
**CNT_COUNT2 = 7**

Esse é o início da especificação CSP gerada.

### 4.4.2. Extração de canais e tipos definidos

Nas próximas linhas se visualiza a definição das variáveis globais, e para elas foi definida a função AvaliaDefinicaoVariavel (Regra 3). Será detalhado apenas um mapeamento por tipo de variável – inteiro e booleano –, os seus semelhantes são trabalhados de forma similar.

```
//inteiros
int CGP_informaEstadoInicialRepouso_MEMORIA;
int CGP_iniciarContadorTempoGongo_MEMORIA;
int CGP_iniciarContadorTempoTotal_MEMORIA;
int CGP_inicializarTemporizador_MEMORIA;
int CGP_informarProximoEstadoDireita_MEMORIA;
int CGP_informarProximoEstadoEsquerda_MEMORIA;

//booleanos

bool MEMORIA_informaChaveKLOANSelecionada_CGP;
bool MEMORIA_informaVelocidadeMenorQueTres_CGP;
bool MEMORIA_informaChaveSeletoraLadoDireito_CGP;
bool MEMORIA_informaBotaoFechamentoDireitoConsole_CGP;
bool MEMORIA_informaBotaoFechamentoDireitoLateral_CGP;
bool CGP_estadoInicialPortasFechadasCMLDireita_MEMORIA;
bool CGP_estadoInicialPortasFechadas_MEMORIA;
bool CGP_executarComandoGongo_MEMORIA;
bool CGP_zerarContadorGongo_MEMORIA;
bool CGP_habilitarSinalizacaoInicioFechamento_MEMORIA;
bool CGP_zerarTemporizador_MEMORIA;
bool MEMORIA_informarEstouroTempoTotal_CGP;
bool MEMORIA_zerarTempoTotalOuPortasFechadas_CGP;
bool KISOL;
bool MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP;
bool MEMORIA_informaBotaoFechamentoEsquerdoConsole_CGP;
bool MEMORIA_informaBotaoFechamentoEsquerdoLateral_CGP;
bool CGP_estadoInicialPortasFechadasCMLEsquerda_MEMORIA;
bool CGP_executarComandoGongo_MEMORIA;
```

Aplicando a Regra 3 e em seguida a Regra 4:

AvaliaDefinicaoVariavel[`int CGP_informaEstadoInicialRepouso_MEMORIA;`]

...

AvaliaDefinicaoVariavel[bool MEMORIA_informaChaveKLOANSelecionada_CGP;]

~>

**channel CGP_informaEstadoInicialRepouso_MEMORIA:** AvaliaTipo[int]

...

**channel MEMORIA_informaChaveKLOANSelecionada_CGP:** AvaliaTipo[bool]

~>

**channel CGP_informaEstadoInicialRepouso_MEMORIA: Int**
**...**
**channel MEMORIA_informaChaveKLOANSelecionada_CGP: Bool**

Em seguida aparece a definição de um tipo enumerado, e o uso deste tipo para definição de variáveis:

```
typedef enum enum_comando {ABRE, FECHA, SINALIZA, PAUSA} Comando;
Comando CGP_receberComandoDireito_MEMORIA;
Comando CGP_receberComandoEsquerdo_MEMORIA;
```

Aplicando a Regra 6 para definição do tipo Comando e Regra 3 para demais sentenças:

AvaliaDefinicaoEnum[typedef enum enum_comando {ABRE, FECHA, SINALIZA, PAUSA} Comando;]
AvaliaDefinicaoVariavel[Comando CGP_receberComandoDireito_MEMORIA;]
AvaliaDefinicaoVariavel[Comando CGP_receberComandoEsquerdo_MEMORIA;]

~>

**Datatype Comando = ABRE | FECHA | SINALIZA | PAUSA**
**channel CGP_receberComandoDireito_MEMORIA:** AvaliaTipo [Comando]

**channel CGP_receberComandoEsquerdo_MEMORIA:** AvaliaTipo [Comando]

~>

**Datatype Comando = ABRE | FECHA | SINALIZA | PAUSA**
**channel CGP_receberComandoDireito_MEMORIA: Comando**

**channel CGP_receberComandoEsquerdo_MEMORIA: Comando**

Assim são construídos os canais envolvidos pela CGP e seu comportamento.

### 4.4.3. Extração do processo

O processo extraído é construído a partir da assinatura da função, isso é feito usando-se a Regra 1.

Segue abaixo:

```
void  CGP_CML( void ) {...}
```

Aplicando a Regra 1 (AvaliaFuncao):

AvaliaFuncao [`void  CGP_CML(void) {...}`]

~>

**CGP_CML** = AvaliaBloco[...]

### 4.4.4. Extração do comportamento do processo

A extração do comportamento do processo – ou seu conjunto de eventos – é feito usando-se as regras de mapeamento de acordo com estruturas ocorrentes. A função AvaliaBloco corresponde à aplicação da regras: AvaliaListAtribuicaoValor e AvaliaListIfElse. Aqui serão mostrados apenas os trechos de código gerais e casos que necessitem um olhar mais cuidado, de forma a não detalhar um trabalho repetitivo.

Segue abaixo o início do corpo da função, são mostradas apenas as estruturas mais externas:

```
if( MEMORIA_informaVelocidade_CGP == TRUE )
{
```

```
       Bloco_ifVelocidade

}

else
{
      CGP_receberComandoDireito_MEMORIA = FECHA;
      CGP_receberComandoEsquerdo_MEMORIA = FECHA;
      CGP_informarProximoEstadoDireita_MEMORIA= 0;
      CGP_informarProximoEstadoEsquerda_MEMORIA= 0;

}
```

O bloco do *if* foi abstraído por enquanto por conta de sua complexidade, mas será detalhado mais a frente. Como apenas um if-else é visto logo abaixo da definição da função, foi considerado logo o AvalaIF-Else ao invés de sua lista. Abaixo segue primeiramente a aplicação da Regra 2 (AvaliaIfElse) em seguida Regra 8 (AvaliaAtribuicaoValor) ao trecho de código considerado acima:

AvaliaIfElse[

```
      if( MEMORIA_informaVelocidadeMenorQueTres_CGP == TRUE )
      {

            Bloco_ifVelocidade


      }

      else
      {
            CGP_receberComandoDireito_MEMORIA = FECHA;
            CGP_receberComandoEsquerdo_MEMORIA = FECHA;
            CGP_informarProximoEstadoDireita_MEMORIA = 0;
            CGP_informarProximoEstadoEsquerda_MEMORIA = 0;

      }

    ]

    ~>
```

**MEMORIA_informaVelocidadeMenorQueTres_CGP?informouVel →**
**if( informouVel ==** AvaliaValor[TRUE] **) then**

```
            Bloco_ifVelocidade; -- considerado mais adiante
```

**else**

AvaliaAtribuicaoValor [CGP_receberComandoDireito_MEMORIA = FECHA;]
AvaliaAtribuicaoValor [CGP_receberComandoEsquerdo_MEMORIA = FECHA;]
AvaliaAtribuicaoValor [
CGP_informarProximoEstadoDireita_MEMORIA = 0;]
AvaliaAtribuicaoValor[
CGP_informarProximoEstadoEsquerda_MEMORIA = 0;]

**~>**
**MEMORIA_informaVelocidadeMenorQueTres_CGP? informouVel →**
**if(informouVel == true ) then**

AvaliaBloco[Bloco_ifVelocidade;] -- considerado mais adiante

**else**

**CGP_receberComandoDireito_MEMORIA.**AvaliaValor[FECHA] →
**CGP_receberComandoEsquerdo_MEMORIA.**AvaliaValor[FECHA] →
**CGP_informarProximoEstadoDireita_MEMORIA.**AvaliaValor[0] →
**CGP_informarProximoEstadoEsquerda_MEMORIA.**AvaliaValor[0] → **SKIP**

**~>**
**MEMORIA_informaVelocidadeMenorQueTres_CGP? informouVel →**
**if(informouVel == true ) then**

AvaliaBloco[Bloco_ifVelocidade;] -- considerado mais adiante

**else**

**CGP_receberComandoDireito_MEMORIA.FECHA →**
**CGP_receberComandoEsquerdo_MEMORIA.FECHA →**
**CGP_informarProximoEstadoDireita_MEMORIA.0 →**
**CGP_informarProximoEstadoEsquerda_MEMORIA.0 → SKIP**

A estrutura Bloco_IfVelocidade possui como cláusulas mais externas dois *if-else* com a diferença apenas do lado considerado, como mostrado abaixo:

```
if( MEMORIA_informaChaveSeletoraLadoDireito_CGP == TRUE )
{
      Bloco_IfChaveLadoDireito
{
else
{
```

```
      if( ( CGP_informarProximoEstadoDireita_MEMORIA != 0 ) &&
            ( MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP == TRUE )
                  && ( KISOL == FALSE ) )
            {

                  CGP_habilitarSinalizacaoInicioFechamento_MEMORIA =
                  FALSE;
                  CGP_zerarContadorGongo_MEMORIA         = FALSE;
                  CGP_zerarTemporizador_MEMORIA         = FALSE;
                  CGP_receberComandoDireito_MEMORIA = FECHA;
                  CGP_informaChaveKLOANSelecionada_MEMORIA = FALSE;
                  CGP_informaEstadoInicialRepouso_MEMORIA = 0;
                  CGP_informarProximoEstadoDireita_MEMORIA    = 0;
            }
}
if(MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP == TRUE )
{

     Bloco_IfChaveLadoEsquerdo


}

else
{
         if( ( CGP_informarProximoEstadoEsquerda_MEMORIA != 0 ) &&
            ( MEMORIA_informaChaveSeletoraLadoDireito_CGP == TRUE ) &&
                  ( KISOL == FALSE ) )
         {

                CGP_habilitarSinalizacaoInicioFechamento_MEMORIA =FALSE;
                CGP_zerarContadorGongo_MEMORIA         = FALSE;
                CGP_zerarTemporizador_MEMORIA         = FALSE;
                CGP_receberComandoEsquerdo_MEMORIA = FECHA;
                CGP_informaChaveKLOANSelecionada_MEMORIA = FALSE;
                CGP_informaEstadoInicialRepouso_MEMORIA = 0;
                CGP_informarProximoEstadoEsquerda_MEMORIA    = 0;
           }

}
```

Os blocos correspondentes aos lados da chave seletora são 2 blocos de if-else – ambos com níveis de hierarquia interna maiores ou iguais a 3 – que se sucedem então serão modularizados em 2 processos que se sucedem através do operador seqüencial “;”, então uma visão geral do processo extraído ao final será:

***CGP_CML = MEMORIA_informaVelocidadeMenorQueTres_CGP?informouVel →***

***if( informouVel == true ) then***

***Proc_ChaveLadoDir; Proc_ChaveLadoEsq***

Modularização dos processos de ambos os lados

***else***
***CGP_receberComandoDireito_MEMORIA.FECHA →***
***CGP_receberComandoEsquerdo_MEMORIA.FECHA →***
***CGP_informarProximoEstadoDireita_MEMORIA.0→***
***CGP_informarProximoEstadoEsquerda_MEMORIA.0→SKIP***

Pelo fato de serem estritamente iguais variando apenas o lado, será trabalhado e mapeado apenas o lado direito sendo os mesmos resultados obtidos considerados para o lado esquerdo. O processo Proc_ChaveLadoDir irá corresponder ao mapeamento do *if-else* que verifica a seleção da chave do lado direito, enquanto Proc_ChaveLadoEsq do lado esquerdo. Abaixo segue o bloco, visto apenas em alto nível, responsável por extrair o comportamento do processo Proc_ChaveLadoDir:

```
if( MEMORIA_informaChaveSeletoraLadoDireito_CGP == TRUE )
{
      Bloco_IfChaveLadoDireito
{
else
{

      if( ( CGP_informarProximoEstadoDireita_MEMORIA != 0 ) &&
            ( MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP == TRUE )
                  && ( KISOL == FALSE ) )
            {

                  CGP_habilitarSinalizacaoInicioFechamento_MEMORIA =
                  FALSE;
                  CGP_zerarContadorGongo_MEMORIA         = FALSE;
                  CGP_zerarTemporizador_MEMORIA         = FALSE;
                  CGP_receberComandoDireito_MEMORIA = FECHA;
                  CGP_informaChaveKLOANSelecionada_MEMORIA = FALSE;
                  CGP_informaEstadoInicialRepouso_MEMORIA = 0;
                  CGP_informarProximoEstadoDireita_MEMORIA    = 0;
            }
}
```

Aplicado o AvaliaIfElse ao bloco mostrado acima:

AvaliaIfElse[

```
if( MEMORIA_informaChaveSeletoraLadoDireito_CGP == TRUE )
{
      Bloco_IfChaveLadoDireito
{
else
{
```

```
      if( ( CGP_informarProximoEstadoDireita_MEMORIA != 0 ) &&
            ( MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP == TRUE )
                  && ( KISOL == FALSE ) )
            {

                  CGP_habilitarSinalizacaoInicioFechamento_MEMORIA =
                  FALSE;
                  CGP_zerarContadorGongo_MEMORIA         = FALSE;
                  CGP_zerarTemporizador_MEMORIA         = FALSE;
                  CGP_receberComandoDireito_MEMORIA = FECHA;
                  CGP_informaChaveKLOANSelecionada_MEMORIA = FALSE;
                  CGP_informaEstadoInicialRepouso_MEMORIA = 0;
                  CGP_informarProximoEstadoDireita_MEMORIA    = 0;
            }
}]
~>
```
***MEMORIA_informaChaveSeletoraLadoDireito_CGP?informouLadoDir ->***
***If(informouLadoDir == true) then***
Bloco_IfChaveLadoDireito – considerado mais adiante
***else***
AvaliaIfElse [
```
      if( ( CGP_informarProximoEstadoDireita_MEMORIA != 0 ) &&
            ( MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP == TRUE )
                  && ( KISOL == FALSE ) )
            {

                  CGP_habilitarSinalizacaoInicioFechamento_MEMORIA =
                  FALSE;
                  CGP_zerarContadorGongo_MEMORIA         = FALSE;
                  CGP_zerarTemporizador_MEMORIA         = FALSE;
                  CGP_receberComandoDireito_MEMORIA = FECHA;
                  CGP_informaChaveKLOANSelecionada_MEMORIA = FALSE;
                  CGP_informaEstadoInicialRepouso_MEMORIA = 0;
                  CGP_informarProximoEstadoDireita_MEMORIA    = 0;
            }
 ]
~>
```
***MEMORIA_informaChaveSeletoraLadoDireito_CGP?informouLadoDir ->***
***If(informouLadoDir == true) then***
Bloco_IfChaveLadoDireito – considerado mais adiante
***else***
***CGP_informarProximoEstadoDireita_MEMORIA?informouEstDir ->***
***MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP?informouChaveLadoEsq->***
***KISOL?k ->***
***If(informouEstDir != 0 and informouChaveLadoEsq == true and k == false ) then***
AvaliaBloco[

```
      CGP_habilitarSinalizacaoInicioFechamento_MEMORIA = FALSE;
      CGP_zerarContadorGongo_MEMORIA         = FALSE;
      CGP_zerarTemporizador_MEMORIA         = FALSE;
      CGP_receberComandoDireito_MEMORIA = FECHA;
      CGP_informaChaveKLOANSelecionada_MEMORIA = FALSE;
      CGP_informaEstadoInicialRepouso_MEMORIA = 0;
      CGP_informarProximoEstadoDireita_MEMORIA    = 0;
```

]
***else***

Inserção de else SKIP por conta do if sem else

***SKIP***

***~>***

***MEMORIA_informaChaveSeletoraLadoDireito_CGP?informouLadoDir ->***

***If(informouLadoDir == true) then***

Bloco_IfChaveLadoDireito – considerado mais adiante

***else***

***CGP_informarProximoEstadoDireita_MEMORIA?informouEstDir ->***

***MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP?informouChaveLadoEsq->***

***KISOL?k ->***

***If(informouEstDir != 0 and informouChaveLadoEsq == true and k == false ) then***

AvaliaAtribuicaoValor[
CGP_habilitarSinalizacaoInicioFechamento_MEMORIA = **FALSE**;]
AvaliaAtribuicaoValor[
CGP_zerarContadorGongo_MEMORIA = **FALSE**;]
AvaliaAtribuicaoValor[
CGP_zerarTemporizador_MEMORIA = **FALSE**;]
AvaliaAtribuicaoValor[
CGP_receberComandoDireito_MEMORIA = FECHA;]
AvaliaAtribuicaoValor[
CGP_informaChaveKLOANSelecionada_MEMORIA = **FALSE**;]
AvaliaAtribuicaoValor[
CGP_informaEstadoInicialRepouso_MEMORIA = 0;]
AvaliaAtribuicaoValor[
CGP_informarProximoEstadoDireita_MEMORIA = 0;]

***else***

***SKIP***

***~>***

***MEMORIA_informaChaveSeletoraLadoDireito_CGP?informouLadoDir ->***

***If(informouLadoDir == true) then***

Bloco_IfChaveLadoDireito – considerado mais adiante

***else***

***CGP_informarProximoEstadoDireita_MEMORIA?informouEstDir ->***

***MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP?informouChaveLadoEsq->***

***KISOL?k ->***

***If(informouEstDir != 0 and informouChaveLadoEsq == true and k == false ) then***

***CGP_executarComandoGongo_MEMORIA.false ->***
***CGP_habilitarSinalizacaoInicioFechamento_MEMORIA.false ->***
***CGP_zerarContadorGongo_MEMORIA.false->***
***CGP_zerarTemporizador_MEMORIA.false->***
***CGP_receberComandoDireito_MEMORIA.fecha->***
***CGP_informaChaveKLOANSelecionada_MEMORIA.false->***
***CGP_informaEstadoInicialRepouso_MEMORIA.0->***
***CGP_informarProximoEstadoDireita_MEMORIA.0->SKIP***

***else***

***SKIP***

Conforme mencionado anteriormente o processo mapeado acima foi modularizado sob a alcunha de Proc_ChaveLadoDir:

***Proc_ChaveLadoDir =***

***MEMORIA_informaChaveSeletoraLadoDireito_CGP?informouLadoDir ->***
***If(informouLadoDir == true) then***
Bloco_IfChaveLadoDireito – considerado mais adiante
***else***
***CGP_informarProximoEstadoDireita_MEMORIA?informouEstDir ->***
***MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP?informouChaveLadoEsq->***
***KISOL?k ->***
***If(informouEstDir != 0 and informouChaveLadoEsq == true and k == false ) then***

***CGP_executarComandoGongo_MEMORIA.false ->***
***CGP_habilitarSinalizacaoInicioFechamento_MEMORIA.false ->***
***CGP_zerarContadorGongo_MEMORIA.false->***
***CGP_zerarTemporizador_MEMORIA.false->***
***CGP_receberComandoDireito_MEMORIA.fecha->***
***CGP_informaChaveKLOANSelecionada_MEMORIA.false->***
***CGP_informaEstadoInicialRepouso_MEMORIA.0->***
***CGP_informarProximoEstadoDireita_MEMORIA.0->SKIP***
***else***
***SKIP***

Resta agora apenas a última parte do Proc_ChaveLadoDir que é o bloco interno ao *if* ele é constituído de um *switch* que trata a navegação para o estado adequado ao lado, neste caso especificamente o direito. O bloco é extremamente longo então será mostrado o mapeamento de apenas dois estados do mesmo (estado 0 e estado 1). Pela regra do AvaliaIfElse, deverá ser aplicado o AvaliaBloco [Bloco_IfChaveLadoDireito], e este por sua vez aplicará AvaliaSwitch ao bloco:

AvaliaSwitch[

```
switch( CGP_informarProximoEstadoDireita_MEMORIA )
{
 case 0:
   if( (MEMORIA_informaChaveKLOANSelecionada_CGP == TRUE ) &&
       (MEMORIA_informaBotaoFechamentoDireitoConsole_CGP == FALSE) &&
       (MEMORIA_informaBotaoFechamentoDireitoLateral_CGP == FALSE ) ){
              CGP_estadoInicialPortasFechadasCMLDireita_MEMORIA = TRUE;
              CGP_estadoInicialPortasFechadas_MEMORIA = TRUE;
              CGP_receberComandoDireito_MEMORIA  = ABRE;
              CGP_executarComandoGongo_MEMORIA  = FALSE;
                 output_bit(PIN_B2,0);
                 CGP_informarProximoEstadoDireita_MEMORIA     = 1;
              }
              else
              {
                 CGP_informarProximoEstadoDireita_MEMORIA = 0;
              }
              break;
           case 1:
              if(MEMORIA_informaChaveKLOANSelecionada_CGP == FALSE )
```

Chamada de função abstraída para CGP_executarComandoGongo_MEMORIA

```
            {
                  CGP_informarProximoEstadoDireita_MEMORIA = 2;
            }
         else
         {

            CGP_receberComandoDireito_MEMORIA = ABRE;
            CGP_informarProximoEstadoDireita_MEMORIA    = 1;
         }

       break;...

]

~>
```

***CGP_informarProximoEstadoDireita_MEMORIA.0 ->*** AvaliaBloco[Bloco_0]
***[ ]***
***CGP_informarProximoEstadoDireita_MEMORIA.1 ->***AvaliaBloco[Bloco_1]
***…***
**~>**
***CGP_informarProximoEstadoDireita_MEMORIA.0 ->*** AvaliaIfElse[

```
if( (MEMORIA_informaChaveKLOANSelecionada_CGP == TRUE ) &&
 (MEMORIA_informaBotaoFechamentoDireitoConsole_CGP == FALSE) &&
 (MEMORIA_informaBotaoFechamentoDireitoLateral_CGP == FALSE ) ){
        CGP_estadoInicialPortasFechadasCMLDireita_MEMORIA = TRUE;
        CGP_estadoInicialPortasFechadas_MEMORIA = TRUE;
        CGP_receberComandoDireito_MEMORIA  = ABRE;
        CGP_executarComandoGongo_MEMORIA  = FALSE;
        output_bit(PIN_B2,0);
        CGP_informarProximoEstadoDireita_MEMORIA     = 1;
 }
 else
 {
     CGP_informarProximoEstadoDireita_MEMORIA = 0;
 }

]
```

***[ ]***
***CGP_informarProximoEstadoDireita_MEMORIA.1 ->*** AvaliaIfElse[

```
if(MEMORIA_informaChaveKLOANSelecionada_CGP == FALSE )
{
     CGP_informarProximoEstadoDireita_MEMORIA = 2;
}
else
{

      CGP_receberComandoDireito_MEMORIA = ABRE;
      CGP_informarProximoEstadoDireita_MEMORIA    = 1;
}
```

***…***
]
**~>**

***CGP_informarProximoEstadoDireita_MEMORIA.0 ->***
***MEMORIA_informaChaveKLOANSelecionada_CGP?informouChave->***
***MEMORIA_informaBotaoFechamentoDireitoConsole_CGP?informouBotaoCon->***
***MEMORIA_informaBotaoFechamentoDireitoLateral_CGP?informouBotaoLat->***
***If(informouChave == true and informouBotaoCon == false and informouBotaoLat == false ) then***

AvaliaBloco[

```
CGP_estadoInicialPortasFechadasCMLDireita_MEMORIA = TRUE;
CGP_estadoInicialPortasFechadas_MEMORIA = TRUE;
CGP_receberComandoDireito_MEMORIA  = ABRE;
CGP_executarComandoGongo_MEMORIA  = FALSE;
output_bit(PIN_B2,0);
CGP_informarProximoEstadoDireita_MEMORIA     = 1;
```

]
}
***else***
AvaliaBloco[

```
CGP_informarProximoEstadoDireita_MEMORIA = 0;
```

]
***[ ]***
***CGP_informarProximoEstadoDireita_MEMORIA.1 ->***
***MEMORIA_informaChaveKLOANSelecionada_CGP?informouChave->***
***If(informouChave == false) then***

AvaliaBloco[

```
CGP_informarProximoEstadoDireita_MEMORIA = 2;
```

]
***else***
AvaliaBloco[

```
CGP_receberComandoDireito_MEMORIA = ABRE;
CGP_informarProximoEstadoDireita_MEMORIA    = 1;
```

]
***...***
***~>***
***CGP_informarProximoEstadoDireita_MEMORIA.0 ->***
***MEMORIA_informaChaveKLOANSelecionada_CGP?informouChave->***
***MEMORIA_informaBotaoFechamentoDireitoConsole_CGP?informouBotaoCon->***
***MEMORIA_informaBotaoFechamentoDireitoLateral_CGP?informouBotaoLat->***
***If(informouChave == true and informouBotaoCon == false and informouBotaoLat == false ) then***
***CGP_estadoInicialPortasFechadasCMLDireita_MEMORIA.true->***
***CGP_estadoInicialPortasFechadas_MEMORIA.true->***
***CGP_receberComandoDireito_MEMORIA.ABRE->***
***CGP_executarComandoGongo_MEMORIA.false->***
***CGP_informarProximoEstadoDireita_MEMORIA.1->SKIP***

***else***

***CGP_informarProximoEstadoDireita_MEMORIA. 0->SKIP***

***[ ]***

***CGP_informarProximoEstadoDireita_MEMORIA.1 ->***
***MEMORIA_informaChaveKLOANSelecionada_CGP?informouChave->***
***If(informouChave == false) then***

```
            CGP_informarProximoEstadoDireita_MEMORIA. 2->SKIP
      else
            CGP_receberComandoDireito_MEMORIA.ABRE->
             CGP_informarProximoEstadoDireita_MEMORIA.1->SKIP
      ...
```

Sumarizando todo processo CGP_CML mapeado até agora:

```
TRUE = true
FALSE = false
CNT_COUNT1 = 7
CNT_COUNT3 = 7
CNT_COUNT2 = 7

channel CGP_informaEstadoInicialRepouso_MEMORIA: Int
...
channel MEMORIA_informaChaveKLOANSelecionada_CGP: Bool

Datatype Comando = ABRE | FECHA | SINALIZA | PAUSA
channel CGP_receberComandoDireito_MEMORIA: Comando
channel CGP_receberComandoEsquerdo_MEMORIA: Comando


CGP_CML = MEMORIA_informaVelocidadeMenorQueTres_CGP?informouVel →

            if( informouVel == true ) then

                  Proc_ChaveLadoDir; Proc_ChaveLadoEsq

            else
                  CGP_receberComandoDireito_MEMORIA.FECHA →
                  CGP_receberComandoEsquerdo_MEMORIA.FECHA →
                  CGP_informarProximoEstadoDireita_MEMORIA.0→
                  CGP_informarProximoEstadoEsquerda_MEMORIA.0→SKIP


Proc_ChaveLadoDir =
MEMORIA_informaChaveSeletoraLadoDireito_CGP?informouLadoDir ->
If(informouLadoDir == true) then

      CGP_informarProximoEstadoDireita_MEMORIA.0 ->
      MEMORIA_informaChaveKLOANSelecionada_CGP?informouChave->
      MEMORIA_informaBotaoFechamentoDireitoConsole_CGP?informouBotaoCon->
      MEMORIA_informaBotaoFechamentoDireitoLateral_CGP?informouBotaoLat->
      If(informouChave == true and informouBotaoCon == false and informouBotaoLat ==
      false ) then
            CGP_estadoInicialPortasFechadasCMLDireita_MEMORIA.true->
            CGP_estadoInicialPortasFechadas_MEMORIA.true->
            CGP_receberComandoDireito_MEMORIA.ABRE->
            CGP_executarComandoGongo_MEMORIA.false->
            CGP_informarProximoEstadoDireita_MEMORIA.1->SKIP
```

Modularização dos processos de ambos os lados

```
 else

        CGP_informarProximoEstadoDireita_MEMORIA.0->SKIP

    [ ]

    CGP_informarProximoEstadoDireita_MEMORIA.1 ->
    MEMORIA_informaChaveKLOANSelecionada_CGP?informouChave->
    If(informouChave == false) then
        CGP_informarProximoEstadoDireita_MEMORIA. 2->SKIP
    else
        CGP_receberComandoDireito_MEMORIA.ABRE->
         CGP_informarProximoEstadoDireita_MEMORIA.1->SKIP
    ...  - - demais estados do lado direito


else
  CGP_informarProximoEstadoDireita_MEMORIA?informouEstDir ->
  MEMORIA_informaChaveSeletoraLadoEsquerdo_CGP?informouChaveLadoEsq->
  KISOL?k ->
    If(informouEstDir != 0 and informouChaveLadoEsq == true and k == false ) then

        CGP_executarComandoGongo_MEMORIA.false ->
        CGP_habilitarSinalizacaoInicioFechamento_MEMORIA.false ->
        CGP_zerarContadorGongo_MEMORIA.false->
        CGP_zerarTemporizador_MEMORIA.false->
        CGP_receberComandoDireito_MEMORIA.FECHA->
        MEMORIA_informaChaveKLOANSelecionada_CGP.false->
        CGP_informaEstadoInicialRepouso_MEMORIA.0->
        CGP_informarProximoEstadoDireita_MEMORIA.0->SKIP
    else

        SKIP
```

O restante dos estados para o processo Proc_ChaveLadoDir, como também Proc_ChaveLadoEsq serão mostrados apenas na especificação final [18]. Mas de antemão é dito que o Proc_ChaveLadoEsq corresponderá ao processo Proc_ChaveLadoDir sofrendo apenas renomeação de algumas variáveis através do reuso do processo. Pois eles possuem comportamentos iguais, variando apenas algumas comunicações específicas para cada lado.

```
Proc_ChaveLadoEsq =

Proc_ChaveLadoDir [e1Dir <- e1Esq, ... ,eNDir <- eNEsq]
```

## 4.5. Avaliação de propriedades

Nesta seção serão mostrados os resultados obtidos da especificação em CSP extraída da CGP. Deve-se lembrar que tal avaliação é apenas experimental, pois a interação só é mostrada com um processo criado manualmente chamado MEMORIA que representa apenas um repositório de valores, donde lê-se e escreve-se os mesmos.

Isso foi feito pelo fato do escopo do trabalho se limitar à aplicação das técnicas de mapeamento sobre o código da CGP, conseguindo apenas extrair o comportamento deste. Então para uma avaliação completa de resultados devem-se considerar todos os processos envolvidos no sistema.

É válido mencionar que algumas adaptações foram feitas à especificação extraída devido à ferramenta FDR exigir que para comparações de tipos inteiros se restrinja a um conjunto de valores, ou seja, teve que se reduzir a cardinalidade do canal. Isso foi feito sobre as variáveis que representam os estados de ambos os lados. Em 8 um screenshot do aviso dado pela ferramenta:

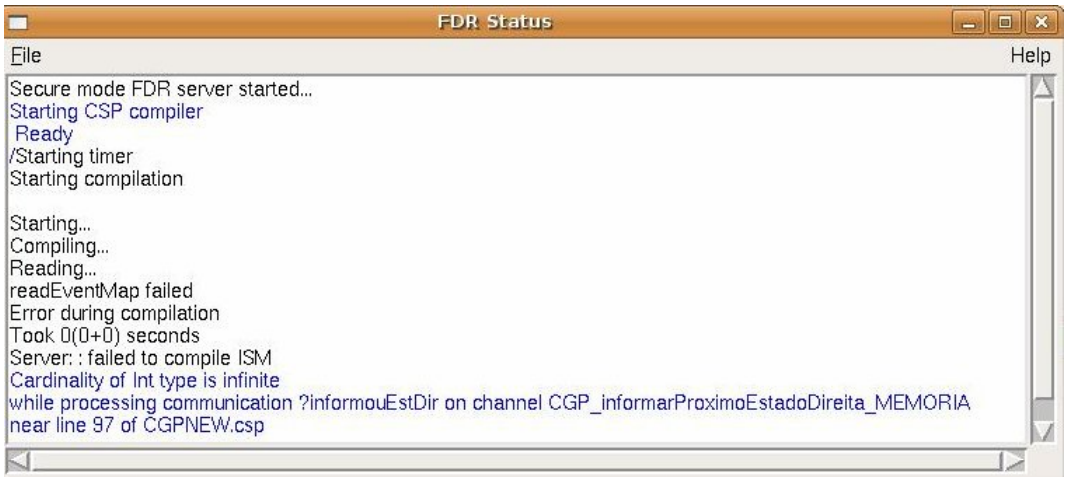



**Figura 8 - Ajuste da cardinalidade do canal**

O screenshot da FDR abaixo mostra os resultados das propriedades do sistema composto pela CGP e pela Memória: Deadlock, Livelock e Determinismo.

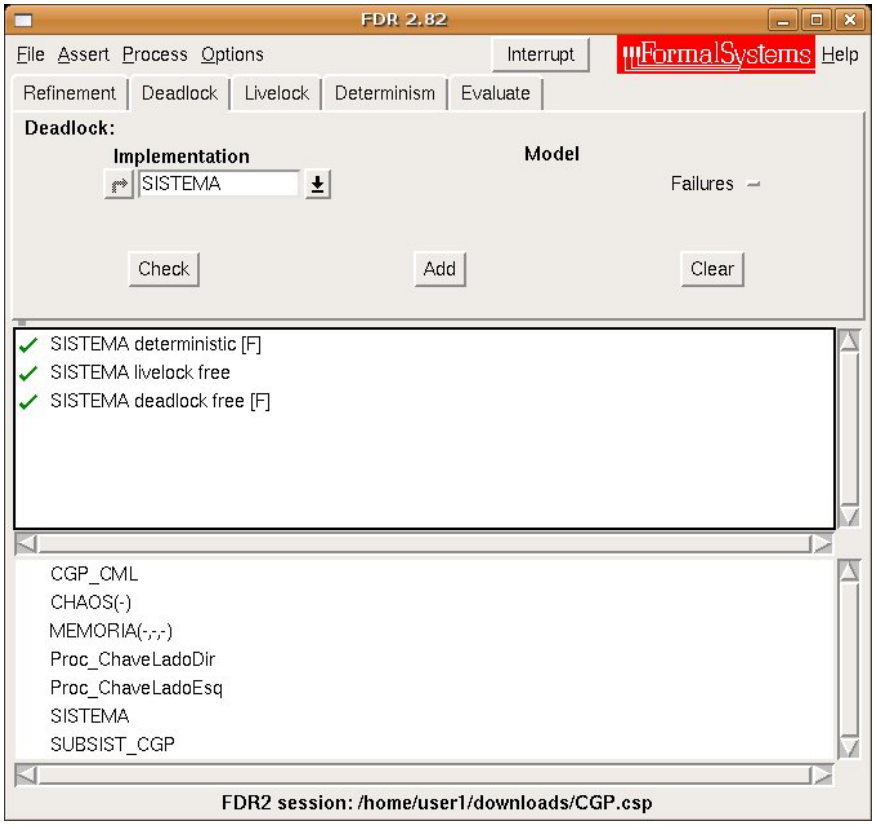



**Figura 9 - Resultados da análise do modelo extraído**

Outra propriedade analisada foi a alcançabilidade, nela se verifica se todos os possíveis comportamentos de um processo são atingíveis. O screenshot em 10 mostra que o estado 6 da operação de cada lado das portas não é atingível:

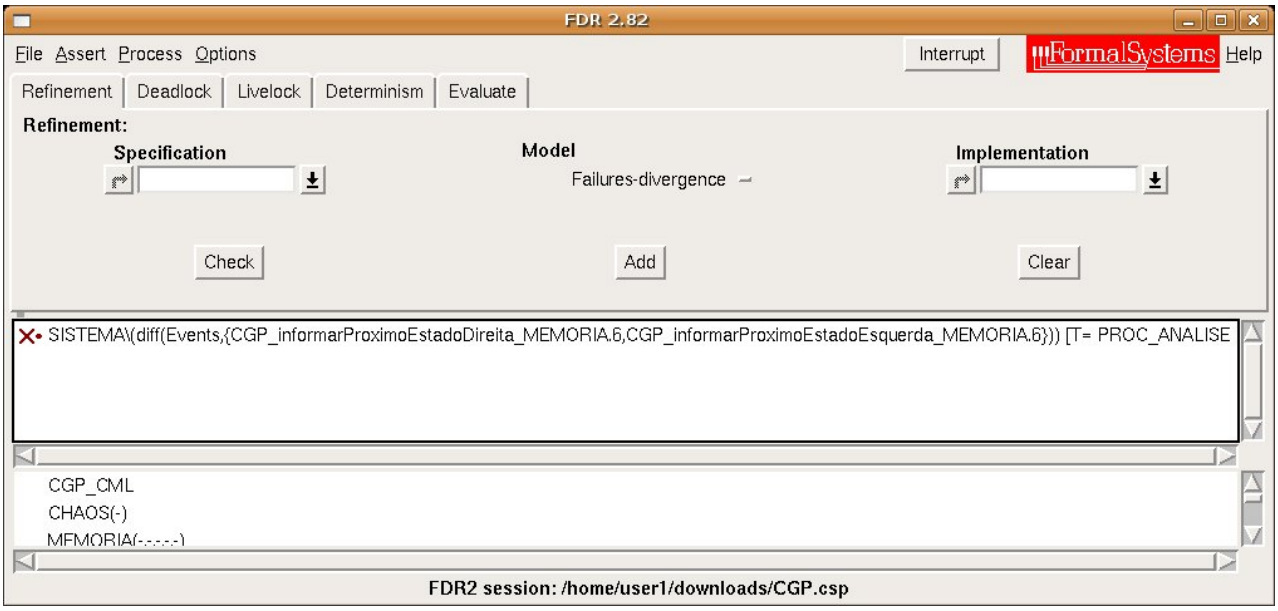



**Figura 10 - Avaliação da alcançabilidade do estado 6**

Para se mostrar isso se fez necessário criar um processo auxiliar de análise (PROC_ANALISE) que correspondia a uma escolha externa ([ ]) dos eventos inatingíveis seguidos de STOP. Para explicitar a inatingibilidade aplicou-se um *hidding* de todos os eventos exceto os dois inatingíveis (*CGP_ informaProximo EstadoDireita_MEMORIA.6* e *CGP_informaProximoEstadoEsquerda_ MEMORIA.6*) em cima do processo SISTEMA (CGP e MEMORIA). E após isso, foi provado um não refinamento pelo modelo de traces (seqüência de eventos) como mostrado pela asserção abaixo:

*assert SISTEMA\(diff(Events,{CGP_informaProximoEstadoDireita_MEMORIA.6, CGP_informaProximoEstadoEsquerda_MEMORIA.6})) [T= PROC_ANALISE*

*PROC_ANALISE = CGP_informaProximoEstadoDireita_MEMORIA.6 ->STOP*

*[]*

*CGP_informaProximoEstadoEsquerda_MEMORIA.6 ->STOP*

Isso significa que o conjunto de eventos (traces) do processo SISTEMA escondendo-se todos os outros exceto os inatingíveis não contem o conjunto de eventos de PROC_ANALISE que corresponde a: vazio e os dois eventos

inatingíveis. Assim prova-se a inatingibilidade desses eventos, que no caso correspondem estados inalcançáveis da CGP.

### 4.5.1. Comparação de resultados

Este tópico apresenta a verificação das propriedades do sistema realizadas em [17], com objetivo de comparar os resultados obtidos em ambos os trabalhos. Como já mencionamos, o trabalho em [17] gera a especificação em CSP a partir de documentos de requisitos textuais usando uma linguagem controlada.

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

Assim, a comparação dos resultados com [17], se limitou à verificação isolada das propriedades de ambos os trabalhos, visto que encontrar uma relação de refinamento não foi possível, a primeira vista, devido aos alfabetos de ambas as extrações possuírem representações distintas. Trabalhos futuros tentarão compatibilizar as versões das especificações para que refinamentos possam ser avaliados.

Em 11 são mostrados os resultados das propriedades: Deadlock, Livelock e Determinismo do trabalho em [17]. Nele verificaram-se as mesmas propriedades tal qual o presente trabalho.

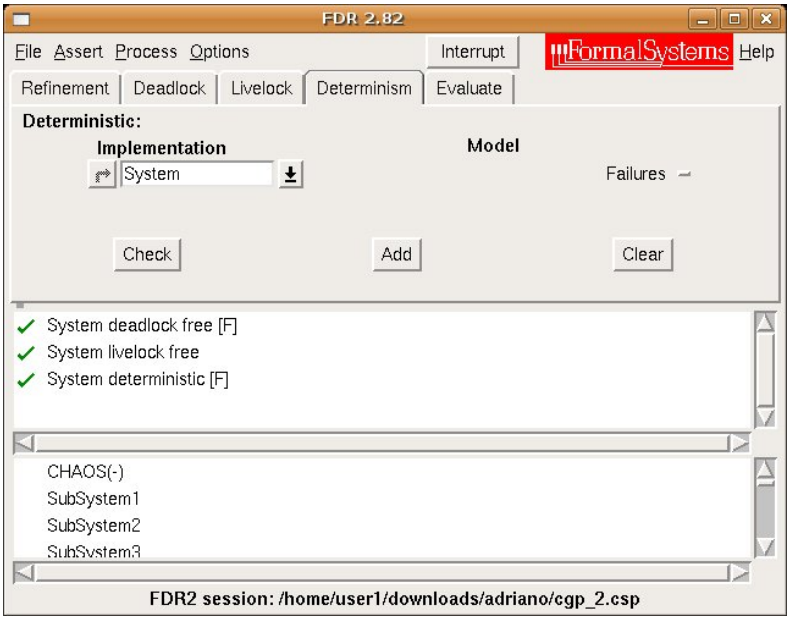



**Figura 11 - Resultados das propriedades do trabalho relacionado**

O problema da alcançabilidade também foi encontrado pelo trabalho [19], pois nele foi uma análise de ferramentas de *model checking* que verificam propriedades em cima da implementação diretamente. E através do uso da ferramenta CBMC fazendo uma asserção que mostra que o valor de uma determinada variável CGP_informarProximoEstadoDireita_MEMORIA é sempre menor ou igual a 5. Em 12 é mostrado o resultado das asserções.

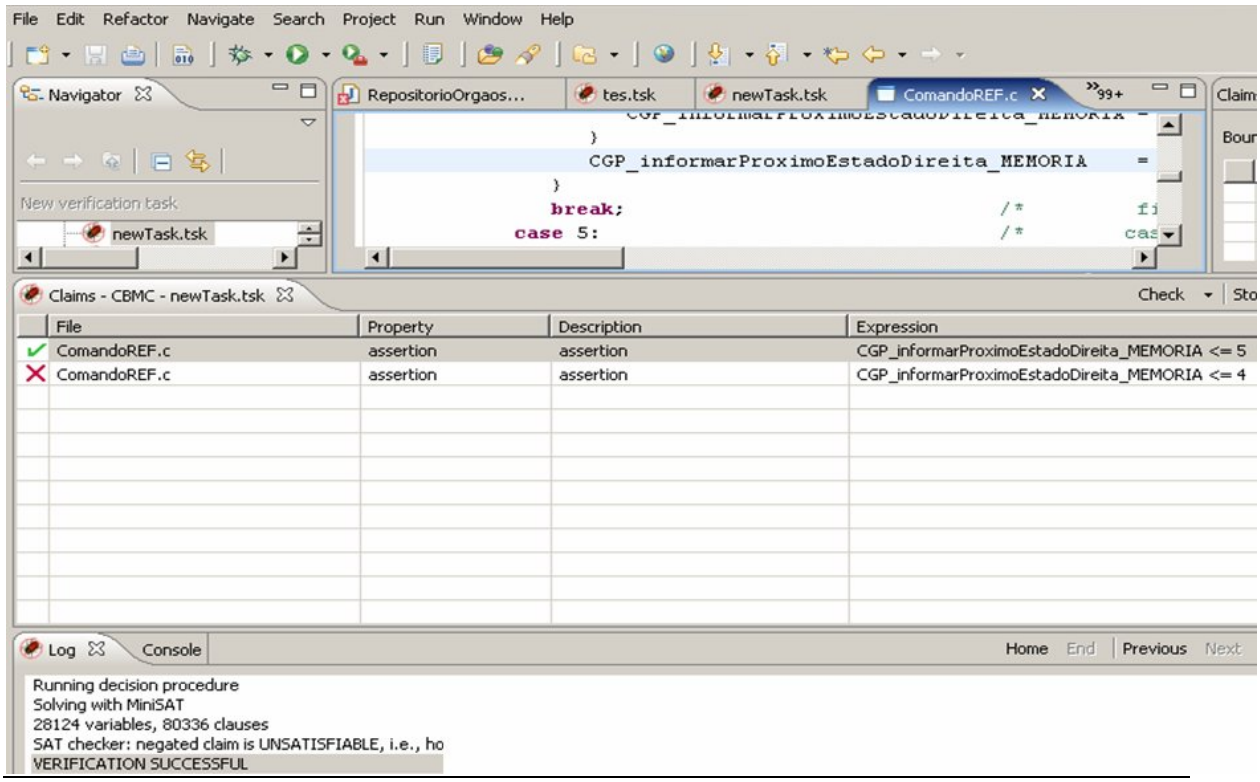



**Figura 12 - Verificação de alcançabilidade por trabalho relacionado**

Um check (√) é mostrado para a asserção que a variável em teste é sempre menor ou igual a 5. Enquanto um (x) é exibido para informar que ela pode assumir valor maior que 4. Em outras palavras, o valor máximo atingido por ela é 5.

# Capítulo 5 – Conclusão

Neste trabalho foi proposto um mecanismo para extração da especificação formal em CSP de um sistema implementado usando a linguagem C. De posse desse modelo pode-se avaliar propriedades do sistema tais como: Deadlock, Livelock e Determinismo, através da submissão do mesmo à análise em ferramentas como a FDR [14].

Antes de mostrar o núcleo do trabalho realizado foram discutidos aspectos relevantes às duas linguagens: CSP e C, respectivamente. O objetivo foi esclarecer termos, conceitos e definições usadas durante a implementação bem como do resultado do mapeamento gerado.

Em seguida foi mostrada a descrição do sistema da Controladora Geral de Portas, ele corresponde a um módulo responsável por controlar a abertura e fechamento das portas de um metrô. Esse sistema foi desenvolvido no projeto Santiago Linha 2 no Chile. O alvo do trabalho foram dois casos de uso responsável por abertura e fechamento de portas no modo manual. Eles definem todas as comunicações e acionamentos periféricos feitos durante esses processos, tais como: inicialização de *timers* e sinalizações.

Foi feita uma análise detalhada dos aspectos sintáticos e semânticos das funcionalidades escolhidas, em vista de se obter um levantamento de padrões encontrados na implementação do sistema da Controladora Geral de Portas. Foi definido um padrão de nomenclatura para aplicação de uma refatoração que facilite o entendimento do programa, bem como o sentido das comunicações. Com isso conseguiu-se definir o conjunto de regras de mapeamentos – voltadas exclusivamente ao domínio do problema – necessárias para aplicação sobre todo o código do sistema.

Na definição das regras de mapeamentos são mostrados exemplos de trechos de códigos que ilustram bem o resultado da aplicação das mesmas. Isso levou a um aperfeiçoamento das regras, bem como a identificação de pontos onde poderiam ser feitas modularizações das regras para que se facilitem ajustes necessários no futuro.

A aplicação foi feita sobre a implementação do sistema CGP dando resultado a uma especificação em CSP que posteriormente foi submetida à análise na FDR para identificação de propriedades. O maior objetivo do trabalho é fornecer um meio de atestar propriedades do sistema após sua implementação.

As principais contribuições deste trabalho foram:

- Definição de um conjunto de regras – considerando as estruturas sintáticas que aparecem nas funcionalidades trabalhadas – que possibilitam a geração de especificação em CSP para um domínio específico de aplicação, no caso deste trabalho a Controladora Geral de Portas de um Metrô.

- Análise de propriedades (Deadlock, Livelock e Determinismo) da Controladora Geral de Portas. Em particular foi feita uma análise de Alcançabilidade para os estados de ambos os lados de operação da CGP (esquerdo e direito) e foi visto que o estado 6 não é atingível.

- Comparação de propriedades isoladamente com modelagens extraídas na etapa de requisitos em [17] do ciclo de desenvolvimento de software.

- Identificação de pontos de modularização no sistema da CGP. Para isso então se extraiu um processo e em seu uso futuro aplicou-se apenas renomeação dos canais diferentes.

- Verificação e prova de estados inatingíveis no sistema, ao menos por navegação interna da maquina de estados proposta por ele. Isso também foi atestado por outro trabalho relacionado [19].

## 4.6. Trabalhos relacionados

A seguir são mostrados alguns trabalhos, abaixo de cada um deles são dados uma breve descrição deles e também os relacionamentos destes com o presente trabalho:

- *Analisando uma abordagem para extração de uma modelagem em CSP à partir de especificações em linguagem natural*

  A abordagem em [17] também foi voltada para o sistema CGP, nele foi definida uma gramática de onde se extraiu todas as entidades bem como seus comportamentos. O relacionamento com o presente trabalho se dá pelo fato de possuírem um sistema alvo em comum e com isso abre-se a possibilidade de comparação das propriedades dos mesmos.

- *Model Checkers: Uma análise de ferramentas para a linguagem de programação C*

  Em [19] focou na análise de ferramentas que analisam propriedades da implementação, através da submissão do código implementado usando a linguagem C. O objetivo foi selecionar e agrupar as ferramentas disponíveis por suas características. A relação com o trabalho aqui discutido é dada pelo fato de se pode atestar propriedades semelhantes por meios distintos (implementação e modelo extraído).

- *Verificação de modelos para programas em um subconjunto de JCSP*

O objetivo deste trabalho em [5] é estabelecer padrões de mapeamentos entre a linguagem de implementação Java através de parte de sua biblioteca JCSP. A aproximação com o presente trabalho se dá pelo fato de ambos extraírem modelos formais a partir de uma linguagem de implementação.

- *C++CSP2*

Esta biblioteca [20, 21] é uma equivalência para C++ tal qual JCSP é para Java. Ou seja, procura emular todas as estruturas da linguagem CSP para uma linguagem de implementação facilitando a introdução de concorrência na linguagem destino.

## 4.7. Trabalhos futuros

À primeira vista um ponto a ser ajustado neste trabalho é fornecer um maior poder de extensibilidade ao mesmo, através da cobertura de todo o sistema no qual a Controladora Geral de Portas se inclui e extração de comportamentos e interações de todos os processos envolvidos no mesmo. Ao final disso verificar-se a completude e a corretude das regras geradas, atualmente isso foi feito experimentalmente visualizando-se a especificação gerada e comparando-se com a semântica esperada.

A linguagem C possui muito mais estruturas a serem exploradas do que as mapeadas por esse trabalho, então outro ponto de melhoria seria a ampliação de estruturas da linguagem cobertas, tais como: ponteiros, estruturas de dados, tipos de pontos flutuantes, etc. Juntamente a isso gerar uma especificação em CSP mais legível identificando mais pontos de modularização nos mapeamentos.

O conjunto de trabalhos no qual este se insere conta também com a geração automática de especificação a partir de requisitos [17], então no estágio atual de ambos os trabalhos só foi possível comparar propriedades isoladamente. A intenção é definir um pareamento de ambas as técnicas de forma a se conseguir provar refinamentos entre elas.

O principal trabalho a ser desenvolvido é integrar os mapeamentos através de *plugin* a uma IDE (*Integrated Development Environment*) como Eclipse, de forma a se automatizar esse processo evitando falhas humanas ao seu uso. E possibilitar a visualização de propriedades a cada marco do estágio de desenvolvimento, por exemplo, avaliando o impacto de mudanças nos requisitos ou apenas validar a implementação feita até o momento.

# *REFERÊNCIAS BIBLIOGRÁFICAS*


[1] **CSP**. Wikipedia
Disponível em: < http://en.wikipedia.org/wiki/Communicating_sequential_processes >
Acesso em: 15 jul. 2007.

[2] ROSCOE, A. W.; **The Theory and Practice of Concurrency**. 1997

[3] RICHARDSON, D.; **CSP - Communicating Sequential Processes.**
Disponível em: < http://ei.cs.vt.edu/~cs5204/sp99/csp.html >
Acesso em: 15 jul. 2007.

[4] CABRAL, G. F. L.; **Formal Specification Generation from Requirements Documents.** 2006

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

[6] HOARE, A. C.; **Communicating Sequential Processes.** 1978

[7] HOARE, A. C.; **Communicating Sequential Processes.** Prentice-Hall**,**1985

[8] JÚNIOR, J. O. C. L.; **Verificando Refatorações Automaticamente.** 2006

[9] Woodcock, J., Davis, J.; **Using Z – Specification, Refinement andProof.** Prentice - Hall, 1996.

[10] **C (Programming Language)**. Wikipedia
Disponível em: < http://en.wikipedia.org/wiki/C_(programming_language) >
Acesso em: 15 jul. 2007.

[11] RITCHIE, D. M.; **The Development of C Language**. Bell Labs / Lucent Technology. 1993
Disponível em: < http://cm.bell-labs.com/cm/cs/who/dmr/chist.html >
Acesso em: 15 jul. 2007.

[12] **C Syntax**. Wikipedia
Disponível em: < http://en.wikipedia.org/wiki/C_syntax >
Acesso em: 15 jul. 2007.

[13] **Programar em C: História da linguagem**. C / C++ Brasil
Disponível em: < http://www.cbrasil.org/wiki/index.php?title=Programar_em_C:_Hist%C3%B3ria_da_Linguagem_C >
Acesso em: 15 jul. 2007.

[14] GOLDSMITH, M; **FDR: User Manual and Tutorial, version 2.77**. Formal Systems (Europe) Ltd, August 2001.

[15] **PROBE Users Manual.**
Disponível em: < http://www.fsel.com/software.html. >
Acesso em: 15 jul. 2007.

[16] **SRS-0617-1 Controle Geral de Portas.** AeS – Automação e Sistemas Ltda.

[17] GOMES, A.; **Verificando a Aplicabilidade de uma Abordagem de Geração de Especificação Formal a Partir de Documentos de Requisitos.** Agosto 2007

[18] MILLANO, F.; **Especificação em CSP da Controladora Geral de Portas.** Agosto 2007

Disponível em: < http://www.cin.ufpe.br/~fmmf/TG/. >

[19] MONTENEGRO, P.; **Model Checkers: Uma análise de ferramentas para a linguagem de programação C.** Agosto 2007

[20] **C++CSP.** University of Kent

Disponível em: < http://www.cs.kent.ac.uk/projects/ofa/c++csp >

[21] BROWN, N., WELCH, P.;**An introduction to the Kent C++CSP Library.** University of Kent

Disponível em: < http://www.cs.kent.ac.uk/projects/ofa/c++csp >

[22] SOMMERVILLE, I.; **Engenharia de Software** – 6ª Edição, Prentice-Hall, 2003.

[23] OLIVEIRA, M., CAVALCANTI, A.; **From Circus to JCSP**. November 2004.

[24] CLARKE, E., WING, J.; **Formal Methods: State of the Art and Future Directions**

[25] BECK, K. **Test-Driven Development by Example.** Addison Wesley, 2003