

# De $CSP_Z$ para $CSP_M$ : Uma Ferramenta Transformacional Java

Adalberto Farias\*    Alexandre Mota    Augusto Sampaio

Universidade Federal de Pernambuco  
Centro de Informática  
Cx. Postal 7851 - CEP 50.780-580  
Recife, PE  
19 de Junho 2001

## Resumo

Model checking é uma técnica de Métodos Formais bastante usada na indústria pelo fato de ser totalmente automática. Uma outra técnica importante é a integração de formalismos, que resulta em uma linguagem formal mais expressiva e caracterizada pelo reuso quase que total das sintaxes e semânticas das linguagens integrantes. Neste artigo apresentamos uma ferramenta, desenvolvida com tecnologia Java, que transforma  $CSP_Z$ , integração formal entre CSP e Z, em  $CSP_M$ , versão de CSP usada para realizar *model checking*. A ferramenta mecaniza uma metodologia já bem fundamentada.

**Palavras-chave:** Ferramenta, Especificação Formal, Model Checking, Refinamento,  $CSP_Z$ ,  $CSP_M$ , FDR e Java.

## 1 Introdução

Para construir *softwares* de qualidade é essencial adotar um método de desenvolvimento que suporte elaboração de descrições abstratas, precisas e concisas do sistema, bem como detecte e elimine erros antes de sua implementação [4, 14]. Alguns modelos de processo de *software* procuram utilizar Métodos Formais por proverem tais benefícios, alicerçados em Lógica e Matemática [15]. Outro fator para a popularização de Métodos Formais é

---

\*emails: {acf,acm,acas}@cin.ufpe.br

a presença de ferramentas de suporte ao uso de notações matemáticas, bem como a análise das propriedades do sistema.

A linguagem  $CSP_Z$  [6] é uma integração formal entre CSP [7, 8] e Z [11]. CSP é uma álgebra de processos, sendo normalmente representada como um sistema de transições finito (grafo dirigido). Devido a isso, torna-se possível aplicar a técnica de *model checking*, através da ferramenta FDR [10] (model checker). Para tanto, os processos CSP são descritos em  $CSP_M$ , que é a versão ASCII de CSP. Por incluir Z e ser bastante recente,  $CSP_Z$  não possui um *model checker* específico para ela. Felizmente, mostrou-se em [1] que se pode mapear  $CSP_Z$  em  $CSP_M$ , permitindo FDR analisar  $CSP_Z$  também.

Neste trabalho apresentamos uma ferramenta que automatiza a metodologia apresentada em [1], no que diz respeito às questões decidíveis, bem como deixa explíatas ao usuário as questões não-decidíveis. A ferramenta foi desenvolvida em Java [18] após criar bibliotecas para CSP e Z em termos de classes. Foram usadas as ferramentas de suporte JLex <sup>1</sup> e JavaCUP <sup>2</sup> para criar os analisadores léxicos e sintáticos, respectivamente.

Em termos de linguagens para especificação de sistemas distribuídos, outras notações como CSP-OZ [6] e uma combinação envolvendo VDM [12] e CCS [13] podem ser utilizadas. Entretanto, apenas CSP-OZ possui uma proposta de ferramenta para auxiliar o seu *model checking*, porém ainda não implementada.

O conversor apresentado neste artigo é uma tentativa pioneira de fornecer suporte para o *model checking* de especificações escritas em  $CSP_Z$ , utilizando uma técnica de conversão para  $CSP_M$ , a qual possui um *model checker* bastante usado. A ferramenta tem sido utilizada numa disciplina da Pós-Graduação do Centro de Informática da UFPE e está disponível na Internet em <http://www.cin.ufpe.br/~acf>.

Organizamos este trabalho como segue. Na Seção 2 apresentamos CSP,  $CSP_M$  e  $CSP_Z$  através de um exemplo bem simples. Na Seção 3 abordamos a estratégia de conversão utilizada. A ferramenta Java é descrita na Seção 4. Na Seção 5 apresentamos nossas conclusões atuais bem como fazemos menção aos nossos futuros trabalhos na presente área. Para um melhor entendimento deste artigo é importante que o leitor conheça as principais estruturas das linguagens CSP e Z<sup>3</sup>.

---

<sup>1</sup><http://www.ualberta.ca/~maldridg/tutor/JavaTut.html>

<sup>2</sup><http://www.cs.princeton.edu/~appel/modern/java/CUP>

<sup>3</sup>Em virtude da limitação de espaço, não fornecemos uma visão geral das linguagens CSP e Z.

## 2 CSP<sub>Z</sub>: integrando CSP e Z para especificar sistemas

A linguagem CSP<sub>Z</sub> é uma extensão conservativa de CSP e Z no sentido de que a maioria dos aspectos semânticos e sintáticos de ambas notações são preservados [6]. Apresentamos a sintaxe desta linguagem através de um exemplo, comparando-o com suas versões em CSP e CSP<sub>M</sub>.

O exemplo representa um *buffer* que armazena dados em uma seqüência. Dados novos são inseridos no final da seqüência e a retirada se faz pelo início da mesma. Se a seqüência estiver vazia, apenas inserções são oferecidas, caso contrário inserções e remoções poderão acontecer. O *buffer* possui tamanho indefinido e armazena inteiros. Assim, a descrição<sup>4</sup> desse buffer em CSP é:

$$\begin{aligned} Buffer(S) = & in?x \rightarrow Buffer(S \frown \langle x \rangle) \prec S = \langle \rangle \succ \\ & ( in?x \rightarrow Buffer(S \frown \langle x \rangle) \\ & \quad \square out!head(S) \rightarrow Buffer(tail(S)) \\ & ) \end{aligned}$$

O código correspondente em CSP<sub>M</sub><sup>5</sup> é:

```
channel in,out : Int
Buffer(s) = if (s == <>) then
    in?x -> Buffer(s^<x>)
else
    ( in?x -> Buffer(s^<x>)
      []
      out!head(s) -> Buffer(tail(s))
    )
```

Nos trechos acima, observamos que existe manipulação de dados na própria descrição dos processos, o que pode não ser muito adequado caso as estruturas de dados sejam complexas.

Em CSP<sub>Z</sub>, a idéia consiste em separar estruturas de dados e processos em duas partes: uma de CSP, descrevendo as interações entre os processos, e outra de Z, responsável por manipular as estruturas de dados. A especificação em CSP<sub>Z</sub> para o mesmo exemplo fica:

---

<sup>4</sup>A construção  $P \prec b \succ Q$  é semelhante a `if b then P else Q`.

<sup>5</sup>A Tabela 1 (Apêndice A) mostra os principais operadores de CSP<sub>M</sub>

**spec** *Buffer*

**channel** *in, out* : [*x*:*Int*]  
**main** = *in*?*x* → **main** □ *out*!*y* → **main**

$\frac{}{s : \text{seq } \mathbb{Z}}$	$\frac{}{s' : \langle \rangle}$
$\frac{\Delta State \quad x! : \mathbb{N}}{s \neq \langle \rangle \quad s' = \text{tail}(s) \quad x! = \text{head}(s)}$	$\frac{\Delta State \quad x? : \mathbb{N}}{s' = s \frown \langle x? \rangle}$

**end\_spec** *Buffer*

As palavras reservadas **spec**/**end\_spec** limitam o corpo da especificação. A declaração de tipo dos canais (**channel** *in, out* : [*x*:*Int*]) possui um outro significado: tipos são denotados por registros. Em  $\text{CSP}_M$ , os canais possuem um tipo declarado explicitamente (**channel** *in, out* : *Int*).

O processo **main** representa a parte de CSP da especificação, podendo inclusive ser construído a partir de outros processos, como mostrado a seguir:

$$\begin{aligned} \text{main} &= P \parallel \{a\} \parallel Q \\ Q &= d \rightarrow a \rightarrow Q \\ P &= a \rightarrow b \rightarrow P \end{aligned}$$

Os esquemas *State* e *Init* representam, respectivamente, o estado e a inicialização do sistema. Toda especificação em  $\text{CSP}_Z$  deve possuir estes dois esquemas, cujos nomes devem ser preservados.

Para cada evento na parte de CSP representando uma operação do sistema, deve existir um esquema em Z cujo nome é formado pela palavra *com* seguida pelo nome do canal (como em *com\_in* e *com\_out*). Caso algum tipo trafegue pelo canal, variáveis de entrada e saída no esquema representarão estes tipos. O nome destas variáveis deve coincidir com a variável usada na declaração de tipo do canal. No caso do exemplo, *x?* em *com\_in* e *x!* em *com\_out* coincidem com a variável *x* do registro [*x*:*Int*], indicando o tipo dos canais *in* e *out*. Esta coincidência de nomes é essencial para fazer as partes de CSP e Z sincronizarem nos mesmos eventos. Cada esquema *com\_X* realiza alguma ação (entrada/saída) sobre o canal X.

O objetivo do conversor é transformar especificações escritas em  $\text{CSP}_Z$  para  $\text{CSP}_M$ , permitindo assim sua verificação utilizando FDR. Na Seção 3 encontram-se os detalhes da conversão.

A ferramenta implementa a estratégia definida em [1], onde as perguntas a seguir foram respondidas: como descrever o espaço de estados em CSP? Como restringir o comportamento de CSP baseado nos valores dos componentes do estado? Como caracterizar completamente a parte de Z como um processo em CSP? Como combinar e sincronizar as partes de CSP e Z de uma especificação escrita em CSP<sub>Z</sub>? Aqui, mostramos apenas como estas respostas foram dadas em termos de código CSP<sub>M</sub>.

### 3 Estratégia de conversão

Nesta seção, apresentamos rapidamente a conversão de CSP<sub>Z</sub> para CSP<sub>M</sub>, usando o exemplo da Seção 2. Algumas questões sobre limitações dessa conversão são tratadas em 4.1.

O trecho de código abaixo representa a especificação do *buffer* em CSP<sub>M</sub>, resultante da aplicação da estratégia. Comentaremos cada parte da conversão após a especificação.

```
channel in,out : Int
Buffer = let
  -- The interface
  Channels = {|in,out|}
  lChannels = {}
  Interface = union(Channels,lChannels)

  -- The CSP part
  main = in?x -> main [] out?y -> main

  -- The Z part
  State = { s | s <- $Seq type$ }
  Init = { s' | s' <- $Seq type$ , s'==<> }
  com (s,in.x)={s' | s'<-State, s'==s^<x> }
  com (s,out.y)={s' | s'<-State, s!=<>,s'==tail(s),y==head(s)}
  Z_CSP = let
    Z(State)=[](States,Comm):{(com(State,c),c)| c<-Interface}@
      States != {} & |~| State': States @ Comm -> Z(State')
    within |~| iState: Init @ Z(iState)
  within (main [|Interface|] Z_CSP)\lChannels
```

Da parte de CSP, as declarações dos canais tiveram sua sintaxe adequada para CSP<sub>M</sub> e os eventos do tipo *canal!variável* foram modificados para *canal?variável*, o restante permaneceu inalterado. O motivo desta última modificação é que agora a parte de CSP deve ser livre de qualquer tipo de

manipulação de dados. A parte de Z fica responsável por gerar um evento *canal.variável* para sincronizar com a parte de CSP.

O maior enfoque da conversão está na parte de Z. Em primeiro lugar, o estado do sistema passa a ser considerado uma tupla. Para nosso exemplo, apenas um componente é representado: a seqüência de dados (inteiros). O espaço de estados é o conjunto de todas as tuplas definidas por uma compreensão de conjuntos (onde  $\leftarrow$  representa a relação  $\in$ ) que satisfazem um determinado predicado (*invariante*).

$$\text{State} = \{ s \mid s \leftarrow \$\text{Seq type\$} \}$$

A inicialização atribui valores iniciais a cada elemento da tupla que compõe o estado do sistema. Vírgulas entre predicados denotam o conectivo lógico AND.

$$\text{Init} = \{ s' \mid s' \leftarrow \$\text{Seq type\$}, s' == \langle \rangle \}$$

Os esquemas de operações são transformados em funções partindo de pares (*estado, comunicação*) para um conjunto contendo o novo estado do sistema. As pré e pós-condições também são incorporadas nestas funções.

```
com (s,in.x)={s' | s'←State, s' == s^<x> }
com (s,out.y)={s' | s'←State, s != <>,s' == tail(s),
               y == head(s)}
```

A declaração de *Interface* serve para reunir os eventos de sincronização entre as partes de CSP e Z<sup>6</sup>. Em nosso exemplo, a interface de sincronização é constituída de todos os eventos acontecendo nos canais *in* e *out*.

```
Channels = {|in,out|}
lChannels = {}
Interface = union(Channels, lChannels)
```

O processo  $Z(\text{State})$  agrupa todas as funções originadas dos esquemas, de tal forma que todas as operações com pré-condição verdadeira estarão disponíveis para sincronizar com a parte de CSP. O processo  $Z\_CSP$  envolve  $Z(\text{State})$  e os detalhes de inicialização do sistema.

```
Z_CSP = let
  Z(State)=[] (States,Comm):{(com(State,c),c) | c←Interface}@
  States != {} & |~| State': States @ Comm -> Z(State')
  within |~| iState: Init @ Z(iState)
```

---

<sup>6</sup>lChannel representa a declaração dos canais locais da especificação. São canais que não aparecem fora do escopo *spec/end\_spec* e por isso devem ser escondidos ao final da sincronização das partes de CSP e Z.

Por fim, a parte de CSP (`main`) é sincronizada com a parte de Z (`Z_CSP`) e os canais locais são escondidos.

```
channel in,out:Int
Buffer =
  let
    ...
  within (main [|Interface|] Z_CSP)\ lChannels
```

## 4 A Ferramenta

Após termos visto como uma especificação em  $CSP_Z$  pode ser transformada em  $CSP_M$ , apresentamos agora uma ferramenta transformacional Java, cujo propósito principal é automatizar os principais passos dessa conversão.

Decidiu-se adotar a linguagem Java para implementar o conversor devido aos seguintes fatores:

- Independência de plataforma e portabilidade da linguagem;
- A linguagem possui suporte de alto nível para a natureza da aplicação. APIs<sup>7</sup> para leitura de arquivos, interfaces gráficas e desenvolvimento de analisadores léxicos e sintáticos;
- Reuso do analisador sintático de Z, implementado anteriormente utilizando JLex e JavaCUP;
- Simplicidade dos requisitos para a utilização do conversor. Apenas uma Máquina Virtual Java, versão 1.2.x ou superior é suficiente.

Na Figura 1 podemos ver a tela principal do conversor, que apresenta o código em  $CSP_M$  correspondente à conversão de nosso *buffer* em  $CSP_Z$  da Seção 2. Nesta tela é possível editar, em modo texto<sup>8</sup>, a especificação em  $CSP_Z$  e gerar automaticamente o respectivo código em  $CSP_M$ . Caso algum erro de sintaxe seja encontrado na conversão, o mesmo será indicado numa caixa de diálogo, ativada pelo botão 'LOG'.

---

<sup>7</sup>Application Programming Interface

<sup>8</sup>Para ser mais preciso, a parte de Z de uma especificação em  $CSP_Z$  é editada em L<sup>A</sup>T<sub>E</sub>X [17]

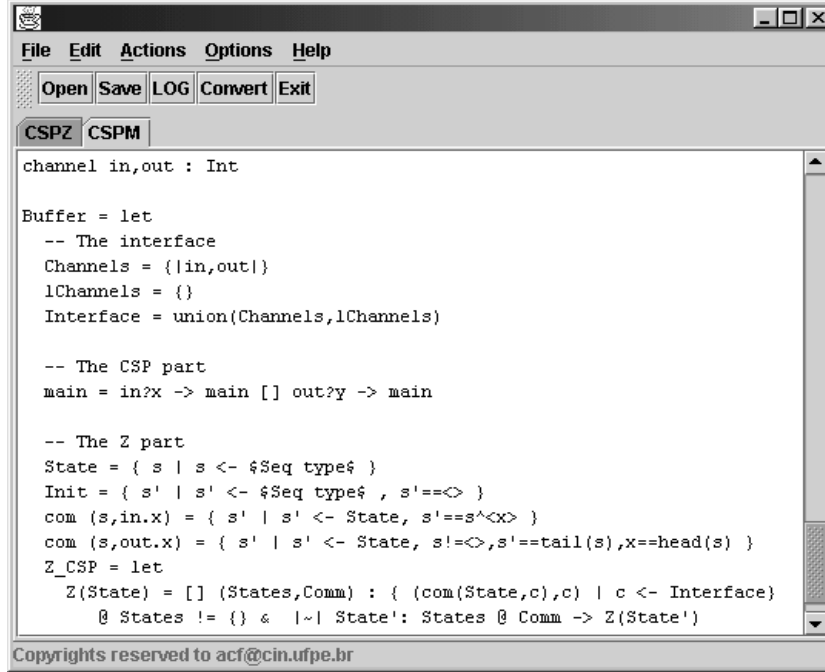


Figura 1: Tela Principal

A verificação da sintaxe de  $CSP_Z$  é feita por dois analisadores sintáticos: um de CSP e outro de Z. Para a implementação do primeiro deles, foi utilizado o trabalho descrito em [16], inclusive com uma gramática proposta para implementação de ferramentas de suporte a CSP. Algumas regras de produção relativas a comandos específicos de FDR foram retiradas e outras pequenas modificações na gramática foram necessárias para a geração do *parser* pelo JavaCUP.

O conversor também provê um sistema de ajuda, que pode ser consultado através do menu 'Help' e apresenta-se sob forma de hipertexto, semelhante à visualização de documentos da Internet.

## 4.1 Questões decidíveis e não-decidíveis

Nem todas as estruturas de Z podem ser automaticamente convertidas em estruturas de  $CSP_M$ . Em geral, isso requer muitos passos de refinamento até que se chegue a uma estrutura mais concreta.

Todas as construções<sup>9</sup> de Z que não possuem correspondentes imediatos ou que levem a construções infinitas em  $CSP_M$  (por exemplo,  $seq \mathbb{Z}$  tem por correspondente  $Seq(Int)$ , uma seqüência infinita) são mapeados em um trecho

<sup>9</sup>Uma visão mais detalhada sobre a conversão de construções mais complexas de Z pode ser encontrada em [5].



delimitado por '\$', ao invés de transformados em alguma estrutura de  $CSP_M$ . O usuário deve modificar esses trechos manualmente, para depois realizar a verificação utilizando FDR.

Apesar de possuir esta limitação, o conversor ajuda bastante em especificações complexas. A necessidade de intervenção do usuário no código gerado é pequena e os principais passos da conversão são feitos automaticamente.

## 4.2 Conversão das principais funções de Z

Com o intuito de fornecer um maior suporte ao usuário, o conversor gera antes da especificação  $CSP_M$ , algumas funções representando os principais operadores e predicados de Z. O usuário pode modificar estas implementações quando desejar, como também definir sua própria função do que ainda não foi implementado. A Figura 2 mostra a tela de definição das funções.

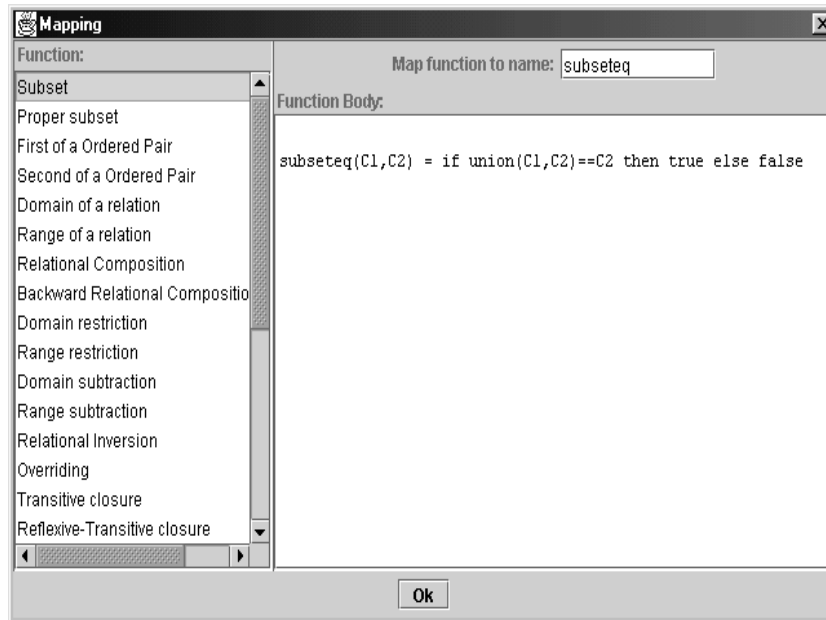


Figura 2 : Mapeamento das funções de Z para  $CSP_M$

A lista do lado esquerdo mostra as principais operações de Z. Ao selecionar uma delas, o respectivo código  $CSP_M$  aparece disponível para edição em 'Function Body' (lado direito da Figura 2).

A utilidade dessas funções consiste em mapear um predicado de Z numa função  $CSP_M$  definida pelo próprio usuário. Como exemplo, a Tabela 2 mostra um operador de Z, sua definição matemática e sua função correspondente em  $CSP_M$ , cujo corpo está definido como mostrado na Figura 2. Esta

função substituirá o operador de  $Z$  no código gerado.

Definição matemática	Operação em $Z$	Função em $CSP_M$
$X \subseteq T$	$X \setminus \text{subseteq } T$	$\text{subseteq}(X, T)$

Tabela 2: Um exemplo de mapeamento entre  $Z$  e  $CSP_M$

## 5 Conclusão

Vimos que o uso de ferramentas é quem difunde Métodos Formais na indústria de *software*. O presente trabalho contribuiu neste aspecto através do conversor, permitindo inclusive que a técnica de *model checking* para CSP seja estendida a  $CSP_Z$  (devido ao artigo [1]), de forma semi-automática, já que o código  $CSP_M$  gerado pode necessitar de uma pequena intervenção do usuário.

O presente trabalho está inserido no contexto do projeto ForMW do LMF (Laboratório de Métodos Formais), como um de seus componentes. Maiores detalhes podem ser encontrados em <http://www.cin.ufpe.br/~lmf>.

A partir da ferramenta atual pretendemos incluir a capacidade de verificar deadlock/livelock local [2]. Esta idéia consiste em particionar especificações  $CSP_Z$  em outras mais simples, preservando sua semântica. As verificações passam então a ser feitas sobre especificações menores, de modo que ao final, propriedades da especificação original possam ser provadas a partir das verificações locais (composicionalidade). Esta teoria sobre uma 'rede' de processos é encontrada em [8] e tratada em termos de implementação em [19], existindo inclusive uma ferramenta implementada em Java com a qual o conversor pode ser integrado.

Um outra extensão deste trabalho consiste em tornar a ferramenta capaz de transformar processos  $CSP_Z$  infinitos (ou que tenham estruturas de dados infinitas) em processos finitos (ou que possuam estrutura de dados finitas). Por exemplo, a especificação mostrada na Seção 2 não pode ser verificada em FDR porque possui uma estrutura de dados infinita ( $Seq(Int)$ ). A seqüência usada pelo *buffer* poderia ser transformada numa estrutura finita sem alteração semântica da especificação original. Essa abordagem é tratada com detalhes em [3, 9].

## Referências

- [1] A. Mota and A. Sampaio. *Model-Checking CSP-Z: strategy, tool support and industrial application*. Science of Computer Programming, Elsevier, Netherlands. (40)1. 2001, pp. 59–96.

- [2] A. Mota. *Formalização e análise do SACI-1 em CSP-Z*. Dissertação de mestrado, Universidade Federal de Pernambuco, 1997, pp. 06–81.
- [3] A. Mota. *Model Checking CSP<sub>Z</sub>: Techniques to Overcome State Explosion*. PhD thesis, Universidade Federal de Pernambuco, to appear.
- [4] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.
- [5] P. Borba and S. Meira. *From model based specifications to functional prototypes*. IEEE TENCON'91 Session on Rapid Prototyping with Functional Programming Languages, 1991.
- [6] C. Fischer. *Combining Object-Z and CSP*. 2<sup>nd</sup> International Conference on Formal Methods for Open Object-based Distributed Systems, Chapman & Hall, London, 1997.
- [7] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.
- [8] A.W.Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
- [9] R. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. D. Phil. Thesis. Oxford University Computing Laboratory, 1999.
- [10] *Failures Divergences Refinement*. FDR2 User Manual, 1997.
- [11] M. Spivey. *The Z Notation: A Reference Manual, 2<sup>nd</sup> Edition*. Prentice Hall International, Englewood Cliffs, NJ, 1992.
- [12] D. Andrews and D. Ince. *Practical Formal Methods with VDM*. McGraw Hill, 1991.
- [13] D. Walker. *Introduction to a Calculus of Communication Systems*. Technical Report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1997.
- [14] R. Pressman. *Software Engineering. A practitioner's approach, 4<sup>th</sup> Edition*. McGraw Hill, 1997.
- [15] J.C.P. Woodcock and M Loomes. *Software engineering mathematics: formal methods demystified*. Pitman. 1988.
- [16] B. Scattergood. *The semantics and Implementation of Machine-Readable CSP*. PhD thesis. University of Oxford, 1998.

- [17] L. Lamport. *LaTeX. A Document Preparation System. Users Guide and Reference Manual*. Addison-Wesley, 1998.
- [18] C.S. Horstmann & G. Cornell. *Core Java 2*. Vols. I e II, Sun Microsystems, Inc. 1999.
- [19] J.M.R. Martin. *The Design and Construction of Deadlock-Free Concurrent Systems*. PhD thesis, University of Buckingham, 1996.

## Apêndice A Principais comandos em CSP<sub>M</sub>

CSP <sub>M</sub>	Explicação
<code>channel in:T</code>	Declaração de canal.
<code>a -&gt; P</code>	Após o acontecimento do evento <code>a</code> , passa a se comportar como o processo <code>P</code>
<code>a?x -&gt; P</code>	Lê um dado de entrada pelo canal <code>a</code> , depois se comporta como o processo <code>P</code>
<code>a!v -&gt; P</code>	Coloca um dado de saída no canal <code>a</code> , depois se comporta como o processo <code>P</code>
<code>P [] Q</code>	Operador de Escolha Externa. <code>P</code> ou <code>Q</code> pode ser executado. Vai depender de quem sincroniza com <code>P [] Q</code>
<code>P  ~  Q</code>	Operador de Escolha Interna. <code>P</code> ou <code>Q</code> pode ser executado. Um deles é escolhido aleatoriamente para acontecer
<code>P \ A</code>	Operador de esconder eventos. O processo <code>P</code> é executado escondendo-se os eventos que ocorrem no conjunto <code>A</code>
<code>if b then P else Q</code>	Comando condicional. Se <code>b</code> for avaliado para <i>true</i> , <code>P</code> é executado, senão <code>Q</code> é executado
<code>P [ X ] Q</code>	Composição paralela. Os processos <code>P</code> e <code>Q</code> sincronizam nos eventos que aparecem em <code>X</code>
<code>P     Q</code>	Entrelaçamento. Os processos <code>P</code> e <code>Q</code> são executados paralelamente e de forma independente
<code>P(s)</code>	Processo parametrizado
<code>let s'=f(s) within P(s')</code>	Declaração local.
<code>[i:T @ P(i)</code>	Indexação de processos.

Tabela 1: Algumas construções de CSP<sub>M</sub>