

# Ações Semânticas de CSP<sub>Z</sub>

Adalberto Cajueiro de Farias  
Centro de Informática  
Universidade Federal de Pernambuco, Brasil

## Relatório Técnico

17 de outubro de 2001

### Resumo

A integração entre linguagens formais para especificar sistemas é uma técnica de Métodos Formais que ganhou muita importância na indústria de software nos últimos tempos, devido ao reuso, quase que total, da sintaxe e semântica das linguagens integrantes, resultando numa linguagem formal mais expressiva. Uma vez criada a linguagem, é necessário definir sua semântica (denotacional, operacional, axiomática, algébrica). Diversos *frameworks* e notações formais podem ser utilizados com esse propósito. Este trabalho representa uma primeira tentativa de definição a semântica operacional de CSP<sub>Z</sub>, em termos de Semântica de Ações.

## 1 Visão geral de Semântica de Ações

Linguagens de programação (LP) podem ter seu significado definido em termos de algum formalismo consolidado. Utilizar notações matemáticas para descrever o significado de uma LP elimina diversos problemas de implementação. Linguagens que possuem sua semântica formalmente definida são implementadas a partir de um documento rigoroso, impossibilitando o surgimento de diferentes interpretações [13].

Notação de Ações é um formalismo para definição da semântica operacional de linguagens de programação através de Semântica de Ações. Definindo suas próprias entidades semânticas, a Notação de Ações define estruturas responsáveis por uma determinada computação. Estas entidades são:

- **Action** - entidades computacionais essencialmente dinâmicas. Sua execução reflete uma determinada computação. Pode ser compreendida como uma máquina abstrata que executa alguma computação.
- **Data** - entidades matemáticas estáticas. Representam informações processadas durante uma computação.
- **Yielder** - representa um item de dado não avaliado, cujo valor depende da informação corrente.

O processamento de uma ação pode produzir os seguintes resultados:

- **Complete** - indicando a terminação normal da ação.
- **Escapes** - correspondendo a uma terminação excepcional, sem falhas.
- **Fail** - representando a terminação anormal da ação, falha.
- **Diverges** - correspondendo a não terminação sem progresso da computação. Por exemplo: um loop infinito.

Em sua execução, uma ação pode processar diversos tipos de informação:

- **Transient** - informação temporária. Dados criados que serão destruídos num curto espaço de tempo.

- **Scoped** - *bindings* de tokens para dados. Associações entre identificadores e valores.
- **Stable** - dados armazenados em células, correspondente aos valores das variáveis.
- **Permanent** - dados comunicados entre ações.

Os diferentes tipos de informação manipulados pelas ações dão origem a diversas facetas<sup>1</sup> das ações semânticas:

- **Básica** - estritamente relacionada aos conceitos fundamentais de controle de fluxo. As ações dessa faceta não processam qualquer tipo de informação. Exemplos:  
*complete* - ação que simplesmente completa sua execução.  
*a<sub>1</sub> or a<sub>2</sub>* - escolhe internamente se executa *a<sub>1</sub>* ou *a<sub>2</sub>*. Caso haja alguma falha na execução da ação escolhida, sua execução é desfeita (exceto mudanças na memória) e a outra ação é executada.  
*a<sub>1</sub> and a<sub>2</sub>* - executa *a<sub>1</sub>* e *a<sub>2</sub>*, mas sem ordem definida, podendo haver inclusive entrelaçamento na execução.  
*a<sub>1</sub> and then a<sub>2</sub>* - executa sequencialmente *a<sub>1</sub>* e depois *a<sub>2</sub>*.
- **Funcional** - engloba ações que processam informações transitórias. Exemplos:  
*give 5* - produz como resultado o dado transitório 5.  
*choose a natural* - escolhe um número natural e o retorna como um valor transitório.  
*a<sub>1</sub> then a<sub>2</sub>* - executa sequencialmente *a<sub>1</sub>* e depois *a<sub>2</sub>*, passando os transitórios produzidos em *a<sub>1</sub>* para *a<sub>2</sub>*.
- **Declarativa**. inclui ações para processar (recebem e produzem) *bindings*. Exemplos:  
*bind x to 5* - produz uma associação entre o identificador *x* e o valor 5 ( $x \mapsto 5$ ).  
*unbind x* - produz uma associação entre *x* e o valor especial *unknown* ( $x \mapsto unknown$ ).  
*a<sub>1</sub> hence a<sub>2</sub>* - semelhante à composição funcional. Repassa os *bindings* de *a<sub>1</sub>* para *a<sub>2</sub>*.
- **Imperativa** - envolve ações para processamento de informações de memória: alocação/liberação de células da memória e mudança no conteúdo das mesmas.  
Exemplos:  
*allocate a cell* - aloca uma célula livre da memória.  
*store 5 in the given cell* - armazena 5 numa célula previamente fornecida.
- **Reflexiva** - contém ações para representar abstrações e suas ativações. Uma abstração é um dado que encapsula uma ação, possibilitando dessa forma ações serem passadas como parâmetro para outras ações. Exemplos:  
*enact the given abstraction* - desencapsula e executa a ação contida na abstração recebida.
- **Diretiva** - engloba ações que processam informações armazenadas em células e *bindings*. Introduce o conceito de indireções, permitindo modelar estruturas circulares (recursivas). Exemplos:  
*recursively bind "fat" to the given abstraction* - cria um *bind* do token "fat" para uma abstração fornecida, onde dentro dessa abstração existe uma referência para o próprio identificador "fat". Esse tipo de ação é útil para definição de estruturas recursivas.
- **Comunicativa** - inclui ações para especificar processamento de informação permanente, dados a serem transmitidos entre diferentes agentes. Exemplos:  
*send a message [to Y<sub>1</sub>][containing 5]* - envia uma mensagem para um outro agente identificado por Y<sub>1</sub> contendo o dado 5.  
*remove a message [at N]* - remove uma mensagem de número serial N do *buffer* do agente em execução.  
*offer a contract [to Y][containing abstraction of A]* - oferece um contrato ao agente Y, contendo uma abstração da ação A. Um contrato é um dado que encapsula uma abstração e pode ser transmitido entre agentes.  
*patiently A* - executa A indivisivelmente. Caso A falhe, então a ação tenta executá-la novamente.

---

<sup>1</sup>Existem ainda ações híbridas, resultantes da combinação entre ações de diferentes facetas.

## 2 Visão geral de CSP

CSP [6, 4] pode ser vista de duas formas: (i) uma notação para descrever sistemas concorrentes ou (ii) uma teoria para estudar processos que interagem por comunicação. O objeto principal dessa notação é um evento de comunicação, pertencente ao conjunto  $\Sigma$ , o conjunto de todos os possíveis eventos de comunicação em CSP. Uma comunicação pode ser entendida como uma transação ou uma sincronização entre processos. Dessa forma, um evento de comunicação acontece apenas se todos os participantes interessados no evento estão preparados para executá-lo. Em outras palavras, quando as partes interessadas no evento estão prontas para executá-lo, então o evento acontece e diz-se que houve uma sincronização. A comunicação entre processos em CSP acontece através de canais, entidades que suportam eventos.

### 2.1 Conceitos fundamentais

#### Processos Básicos

STOP - representa um processo que está em deadlock.

SKIP - representa um processo que termina com sucesso.

#### Prefixação

Construção da linguagem para especificar processos e seus eventos. Exemplo:

$$up \rightarrow down \rightarrow up \rightarrow down \rightarrow STOP$$

O processo realiza a sequência de eventos  $\langle up, down, up, down \rangle$  e depois entra em deadlock. Construções recursivas também são permitidas:

$$P = a \rightarrow P$$

$$Q = b \rightarrow R$$

$$R = c \rightarrow Q$$

P é um processo que realiza indefinidamente o evento  $a$ . O processo Q realiza o evento  $b$  e depois passa a se comportar como o processo R. Por sua vez, R realiza o evento  $c$  e passa a se comportar como Q.

#### Eventos com valores

Um evento de comunicação pode envolver valores de entrada/saída em canais de comunicação:  $c?x$  representa um evento consistindo em receber um dado  $x$  no canal  $c$  e  $c!x$  significa colocar um valor  $x$  no canal  $c$ . O processo abaixo poderia representar um processo que recebe um dado por um canal ( $in$ ) e o repassa para outro canal ( $out$ ).

$$P = in?x \rightarrow out!x \rightarrow P$$

### 2.2 Operadores básicos de CSP

#### Escolha externa

A combinação de processos utilizando a escolha externa representa a execução de apenas um deles. Dessa forma:

$$P = Q \square R$$

significa que o processo P vai se comportar ou como Q ou como R. A decisão do comportamento a ser seguido vai ser determinada por quem sincronizar com P.

#### Escolha não-determinística

CSP possui termos para representar uma escolha não-determinística. Processos não-determinísticos decidem internamente o seu comportamento. Por exemplo:

$$P = Q \sqcap R$$

significa que o processo P pode se comportar como Q ou como R, não importando o fato de existir outro processo desejando sincronizar com P. Em duas execuções do processo P é possível que o mesmo se comporte como Q na primeira vez e como R na segunda. A escolha é feita de forma inesperada, incerta, não-determinística.

### Escolha condicional

Representa uma decisão determinística baseada na validade de uma condição, semelhante ao comando condicional encontrado em linguagens de programação (*if then else*).

$$P = Q \prec b \succ R$$

Na execução do processo P, se a condição  $b$  for verdadeira, então o processo vai se comportar como Q, caso contrário, se comporta como R.

### Operadores de paralelismo

#### *Paralelismo síncrono*

Combinador de processos representando sincronização em todos os eventos. Por exemplo:

$$\begin{aligned} P &= a \rightarrow b \rightarrow P \\ Q &= a \rightarrow b \rightarrow Q \\ R &= a \rightarrow c \rightarrow R \\ P \parallel Q \end{aligned}$$

Na execução de qualquer evento, P e Q devem progredir juntos, produzindo como resultado a sequência de eventos  $\langle a, b, a, b, a, b, a, b, \dots \rangle$ .  $P \parallel R$  seria equivalente ao processo  $a \rightarrow STOP$ , pois P e R concordam apenas no evento  $a$ .

#### *Paralelismo alfabetizado*

Combinador de processos que permite sincronização dos mesmos apenas em eventos específicos. Sincronizar processos na forma:

$$A \parallel_X B$$

significa que, nos eventos contidos em  $X \cap Y$ , os processos A e B devem sincronizar. Nos eventos ocorrendo em  $X \setminus Y$  o processo A executa independente. Da mesma forma, nos eventos em  $Y \setminus X$  o processo B executa independente.

#### *Entrelaçamento*

Combinador de processos onde cada processo componente executa de forma independente, ou seja, os processos não sincronizam em evento algum.

$$P \parallel\parallel Q$$

Os processos P e Q executam normalmente sem influência mútua, podendo inclusive a execução ocorrer de forma paralela (em processadores diferentes).

#### *Paralelismo generalizado*

Combinador de processos que estabelece paralelismo de forma geral, ou seja, generaliza os demais operadores de paralelismo de CSP. A combinação dos processos:

$$P \parallel_x Q$$

possui o seguinte significado: seja A o alfabeto<sup>2</sup> de P ( $A = \alpha P$ ) e B o alfabeto Q ( $B = \alpha Q$ ). Então:

- Nos eventos ocorrendo em X, a execução é semelhante a  $P \parallel Q$ .
- Nos eventos ocorrendo em  $(A \cap B) \setminus X$ , a execução é semelhante a  $P \sqcap Q$ .

---

<sup>2</sup>Conjunto dos eventos comunicados por um processo

- Nos eventos ocorrendo em  $(A \setminus B) \setminus X$ , apenas P progride.
- Nos eventos ocorrendo em  $(B \setminus A) \setminus X$ , apenas Q progride.

A partir do operador de Paralelismo Generalizado algumas igualdades podem ser estabelecidas:

$$P \parallel\!\!\parallel Q = P \parallel_{\{\}} Q$$

$$P_{X||Y} Q = P_{X \cap Y} || Q$$

### 3 Visão Geral de Z

A notação Z [12] é baseada na Teoria dos Conjunto e Lógica-Matemática. A Teoria dos Conjuntos é utilizada para especificar estruturas de dados de forma abstrata. A Lógica-Matemática é um cálculo de predicados de primeira ordem.

Através do uso dessas duas teorias, objetos matemáticos e suas propriedades podem ser reunidos em *esquemas*, estruturas contendo declarações e restrições escritas em forma de predicados. Assim, esquemas podem ser utilizados para modelar o estado e as operações de um sistema. A grande vantagem dessa abordagem é a possibilidade de realizar refinamentos no sistema baseando-se no refinamento de cada esquema (composicionalidade). Os refinamentos podem ser feitos sobre dados ou operações. O nível em que as especificações são escritas permitem um tratamento abstrato do problema, de forma muito próxima à matemática convencional, fazendo com que especificações escritas nesta linguagem sejam abstratas, concisas, completas, não-ambíguas, fáceis de manter, compreensíveis e de custo efetivo. A estrutura geral de uma especificação em Z é a seguinte:

## Tipos

Definição dos novos tipos de dados a serem utilizados na especificação.

[*Mensagem*]

$$Mensagem ::= Sucesso \mid Erro$$

Os exemplos acima mostram duas formas de declaração de um novo tipo de dado. Na primeira delas os valores assumidos pelo tipo não são especificados, enquanto que na segunda o tipo pode assumir apenas um dos valores indicados.

## Funções

### Definição das funções auxiliares utilizadas na especificação.

$$\frac{[X, Y] \quad First : X \times Y \rightarrow X}{\forall x : X; y : Y \bullet First(x, y) = x}$$

A função *First* mostrada acima foi definida de forma genérica e pode ser utilizada na especificação.

## Estado

Esquema contendo a descrição do estado do sistema (componentes e restrições).

$pairs : \mathbb{P}\mathbb{N}$
$\forall x \in pairs \bullet x \bmod 2 = 0$

O estado do sistema é constituído por um componente (*pairs*) que é um subconjunto dos números naturais, onde todos os elementos pertencentes a *pairs* são divisíveis por 2. Esta restrição, conhecida por *invariante*, deve ser preservada pelas operações do sistema.

## Inicialização

Esquema de inicialização do estado.

<i>Init</i>
<i>State'</i>
$\text{pairs}' = \emptyset$

A inicialização apenas atribui valores *default* aos componentes do estado.

## Operações

Esquemas representando as operações providas pelo sistema.

<i>add_number</i>	<i>search_number</i>
$\Delta State$	$\Xi State$
$x? : \mathbb{N}$	$x! : \mathbb{N}$
$x? \bmod 2 = 0$	$\exists y : \mathbb{N} \bullet y \in \text{pairs}$
$\text{pairs}' = \text{pairs} \cup \{x?\}$	$x! = y$

Nas operações apresentadas acima, o invariante do sistema é garantido pela pré-condição de cada esquema. Em *add\_number* ocorre a mudança do estado do sistema ( $\Delta State$ ) e existe a garantia de que apenas números pares ( $x? \bmod 2 = 0$ ) serão inseridos no conjunto ( $\text{pairs}' = \text{pairs} \cup \{x?\}$ ). A operação *search\_number* não altera o estado do sistema ( $\Xi State$ ) e retorna com resposta um número pertencente a *pairs*.

Existem diversos operadores definidos em Z. Tendo por base a Teoria dos Conjuntos, todas as demais construções herdam as propriedades dos conjuntos na seguinte ordem: conjuntos, relações, funções, seqüências, bags.

Apesar de possuir uma ampla expressividade para representação de dados e operações, a linguagem Z não possui operadores que permitam modelar interações entre processos, tornando a linguagem inadequada para modelagem de sistemas distribuídos.

## 4 Descrição informal de $CSP_Z$

A linguagem  $CSP_Z$  [2, 1] é uma integração semântica entre CSP e Z tal que CSP modela a descrição e interação entre os processos e Z manipula as estruturas de dados. O propósito principal da linguagem é utilizar as notações componentes de forma complementar. Por ser uma Álgebra de Processos, CSP é adequada para descrever interações entre os mesmos. Por outro lado, a linguagem Z não possui termos suficientes para representar ordem das operações, porém mostra-se bastante adequada para modelar estruturas de dados complexas. Valendo-se da natureza complementar entre CSP e Z,  $CSP_Z$  procura utilizar as linguagens de forma que uma supere a baixa expressividade da outra. Separando os processos das estruturas de dados, uma especificação em  $CSP_Z$  é construída em duas partes:

- Parte de CSP: contém a declaração de canais e processos que compõem o sistema. Esta parte é livre de manipulação de qualquer estrutura de dados.
- Parte de Z: contém um esquema representando o estado do sistema, um esquema de inicialização e esquemas representando as operações.

O objetivo deste trabalho consiste em especificar como estas partes executam de forma sincronizada. Os agentes são os principais elementos utilizados na definição do comportamento de uma especificação em  $CSP_Z$ .

Seguindo a estrutura proposta em [9], a forma geral de uma especificação em  $CSP_Z$  é a seguinte:

### Especificação

#### Parte de CSP

##### Declaração de canais

## Declaração de processos

### Declaração do processo principal

#### Parte de Z

## 4.1 Especificação

### Sintaxe

(1) Specification =  $\llbracket$  “spec” Identifier CspPart ZPart “end\_spec” Identifier  $\rrbracket$

### Semântica

A execução de uma especificação consiste da inicialização, seguida da habilitação de todos os esquemas que possuem pré-condição verdadeira. Um deles é escolhido de forma não-determinística para ser executado. Cada esquema pode gerar um evento que sincroniza com a parte de CSP. Dessa forma, os esquemas de Z analisam dados e fazem mudança no estado, enquanto a parte de CSP diz qual evento deve acontecer num determinado instante.

### Exemplos

spec *Buffer*

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

<i>State</i>
$s : \text{seq } \mathbb{Z}$
<i>com_out</i>
$\Delta State$
$x! : \mathbb{N}$
$s \neq \langle \rangle$
$s' = \text{tail}(s)$
$x! = \text{head}(s)$

<i>Init</i>
<i>State'</i>
$s' : \langle \rangle$
<i>com_in</i>
$\Delta State$
$x? : \mathbb{N}$
$s' = s \cap \langle x? \rangle$

end\_spec *Buffer*

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.

Pelos canais *in* e *out* traferam tuplas contendo apenas um dado (inteiro). O processo *main* representa a parte de CSP, onde dois eventos podem acontecer (*in*?*x* ou *out*!*y*). Esta escolha vai depender de quem sincroniza com o processo *main*. Após acontecer um destes eventos, o *buffer* passa a se comportar como *main*, oferecendo os dois eventos novamente.

O esquema *State* define o estado do sistema sendo composto por apenas um componente (*s*), neste caso, uma seqüência de inteiros. *Init* é o esquema que inicializa todos os componentes do estado. Os esquemas *com\_in* e *com\_out* representam as operações de inserir e retirar um elemento da seqüência, respectivamente. Cada uma das operações em Z sincronizará com a parte de CSP em um evento particular (*com\_in* com *in*?*x* e *com\_out* com *out*!*y*). Caso um evento na parte de CSP necessite de sincronização com a parte de Z, então o esquema deve possuir como nome o prefixo “com\_” seguido pelo nome do canal. Essa coincidência de nomes é importante para indicar que cada esquema poderá sincronizar apenas em um evento específico.

O esquema *com\_out* possui as seguintes características:

1. Declarações: não modifica o estado do sistema ( $\Delta State$ ) e declara uma variável de saída ( $x!$ ).
2. Pré-condição: verifica se a seqüência possui elementos ( $s \neq \langle \rangle$ ).

3. Pós-condição: retira o primeiro elemento da seqüência ( $s' = tail(s)$ ) e o devolve como um valor de saída ( $x! = head(s)$ ).

Pelo fato deste esquema sincronizar com um evento da parte de CSP, devolver um valor de saída significa que este valor ( $x$ ) será colocado no respectivo canal ( $out$ ). Dessa forma, isso pode ser visto com se a parte de Z tivesse gerando um evento  $out!x$ .

## 4.2 Parte de CSP

### Sintaxe

- (1)  $CspPart = \llbracket ChannelDeclaration^* ProcessDeclaration^* MainDeclaration \rrbracket$

### Semântica

A parte de CSP de uma especificação escrita em  $CSP_Z$  contém as declarações dos canais utilizados na comunicação, definições dos diferentes processos que fazem parte da especificação e o processo principal (*main*). Canais em CSP servem como uma ponte de ligação entre processos que interagem entre si. Com esta abstração, dois processos podem sincronizar em eventos que acontecem em determinado canal sem interferir nos demais processos do sistema. Processos sem CSP são construções da linguagem que servem para descrever uma computação a ser realizada. O processo *main* representa a parte de CSP de uma especificação em  $CSP_Z$  e sincronizará com a parte de Z.

### 4.2.1 Declaração de canais

#### Sintaxe

- (1)  $ChannelDeclaration = \llbracket \text{"channel"} Identifier \langle , Identifier \rangle^* \text{" : " Type} \rrbracket$
- (2)  $Type = \llbracket \text{"["} \rrbracket \mid \llbracket \text{"["} Identifier \text{" : " Identifier} \langle , Identifier \text{" : " Identifier} \rangle^* \text{" ]"} \rrbracket$

#### Semântica

Uma declaração de canais quando elaborada realiza a criação de um canal de comunicação através do qual processos podem trocar mensagens.

#### Exemplos

- channel data : [ ]
- channel c : [ x:T, y:T ]
- channel in, out : [ x:Int ]

### 4.2.2 Declaração de processos

#### Sintaxe

- (1)  $Process = \llbracket STOP \rrbracket \mid \llbracket SKIP \rrbracket \mid \llbracket Identifier \rrbracket \mid \llbracket Event \rightarrow Process \rrbracket \mid \llbracket Process \parallel Process \rrbracket \mid \llbracket Process \square Process \rrbracket \mid \llbracket Process \sqcap Process \rrbracket \mid \llbracket Process \parallel \underset{SetEvent}{Process} \rrbracket$



- (2) Event =  
 $\llbracket \text{Identifier} \rrbracket$   
 $\llbracket \text{Identifier} \text{ "?" Identifier} \rrbracket$   
 $\llbracket \text{Identifier} \text{ "!" Identifier} \rrbracket$
- (3) SetEvent =  
 $\llbracket \text{"{" "}" } \rrbracket$   
 $\llbracket \text{"{" Event } \langle , \text{Event} \rangle^* \text{"}" } \rrbracket$   
 $\llbracket \text{"{" | " Event } \langle , \text{Event} \rangle^* \text{" |"} \rrbracket$

## Semântica

Uma declaração de processo consiste em criar processos identificados por um nome. Tais processos podem ser elementares (STOP, SKIP), processos comunicando eventos ( $a \rightarrow b \rightarrow STOP$ ), processos recursivos ou até mesmo uma combinação entre demais processos. Quando um processo P qualquer é executado, o mesmo comunica um evento através de canais. Se não existir alguém interessado no mesmo evento, então o processo P admite que o evento aconteceu e continua sua execução. Caso exista algum outro processo Q interessado em interagir com P num evento específico, o canal funcionará como elemento de ligação, onde P avisa que está pronto para aceitar o evento. Neste caso, P só progride quando algum outro processo também avisa que está pronto para aceitar o mesmo evento. Os operadores de paralelismo foram abordados em 2.2.

- STOP e SKIP são os processos mais simples de CSP. STOP é um processo que está em deadlock, e SKIP representa um processo que terminou com sucesso.
- A construção  $Process = Identifier$  constrói um agente cujo comportamento é idêntico ao processo indicado pelo identificador.
- A construção  $Process = Event \rightarrow Process$  representa a criação de um agente que vai comunicar um determinado evento (*Event*) depois vai se comportar novamente como um processo. Construções recursivas são permitidas.
- A construção  $Process = Process \parallel Process$  constrói um agente que vai computar o entrelaçamento entre os processos componentes.
- A construção  $Process = Process \square Process$  constrói um agente que vai computar a escolha externa entre os processos componentes.
- A construção  $Process = Process \sqcap Process$  constrói um agente que vai computar a escolha interna (não determinística) entre os processos componentes.
- A construção  $Process \parallel_{SetEvent} Process$  constrói um agente que vai computar o paralelismo generalizado entre os processos componentes.
- *SetEvent* indica um conjunto de eventos.  $\{a\}$  é um conjunto de eventos contendo apenas o evento *a*.  $\{ | a, b, c | \}$  é um conjunto representando todos os eventos acontecendo nos canais canal *a*, *b* e *c*.

## Exemplos

- $P = a \rightarrow b \rightarrow STOP$
- $Q = c \rightarrow SKIP$
- $R = a \rightarrow b \rightarrow R$
- $BufferServer = in?x \rightarrow BufferServer \square out!x \rightarrow BufferServer$
- $BufferClient = in!x \rightarrow BufferClient \sqcap out?x \rightarrow BufferClient$
- $Buffer = BufferServer \mid \{ | in, out | \} \mid BufferClient$
- $TwoBufferServer = BufferServer \parallel BufferServer$

### 4.2.3 Declaração do processo principal da parte de CSP

#### Sintaxe

- (1) MainDeclaration =  $\llbracket$  "main" "=" Process  $\rrbracket$

#### Semântica

Representa a criação de um processo responsável por toda a parte de CSP. Este processo sincronizará com a parte de Z.

#### Exemplos

- $main = in?x \rightarrow main \square out!y \rightarrow main$
- $main = P \square Q$   
 $P = a \rightarrow b \rightarrow P$   
 $Q = c \rightarrow d \rightarrow Q$

## 4.3 Parte de Z

#### Sintaxe

- (1) ZPart =  $\llbracket$  Paragraph\*  $\rrbracket$
- (2) Paragraph =  $\llbracket$  "\begin{schema}" "{" Identifier "}" SchemaText "\end{schema}"  $\rrbracket$
- (3) SchemaText =  $\llbracket$  DeclPart  $\langle$  "\where" Predicate  $\rangle$   $\rrbracket$
- (4) DeclPart =  $\llbracket$  Declaration  $\langle$  NL<sup>3</sup> DeclPart  $\rangle$   $\rrbracket$
- (5) Declaration =  
 $\llbracket$  Expression ":" Expression  $\rrbracket$  |  
 $\llbracket$  Expression  $\rrbracket$
- (6) Predicate =  
 $\llbracket$  "true"  $\rrbracket$  |  
 $\llbracket$  "false"  $\rrbracket$  |  
 $\llbracket$  Expression  $\rrbracket$  |  
 $\llbracket$  Predicate NLPredicate  $\rrbracket$
- (7) Expression =  
 $\llbracket$  Identifier  $\rrbracket$  |  
 $\llbracket$  Expression "'"  $\rrbracket$  |  
 $\llbracket$  Expression "?"  $\rrbracket$  |  
 $\llbracket$  Expression "!"  $\rrbracket$  |  
 $\llbracket$  Expression "=" Expression  $\rrbracket$  |  
 $\llbracket$  "\Delta" Expression  $\rrbracket$  |  
 $\llbracket$  "\Xi" Expression  $\rrbracket$  |  
 $\llbracket$  NUMBER  $\rrbracket$  |  
 $\llbracket$  ArithExpression  $\rrbracket$  |  
 $\llbracket$  RelationalExpression  $\rrbracket$
- (8) ArithExpression =  
 $\llbracket$  Expression "\*" Expression  $\rrbracket$  |  
 $\llbracket$  Expression "/" Expression  $\rrbracket$  |  
 $\llbracket$  Expression "+" Expression  $\rrbracket$  |  
 $\llbracket$  Expression "-" Expression  $\rrbracket$

---

<sup>3</sup>Representa o caracter 'New Line'

- (9)  $\text{RelationalExpression} =$   
 $\llbracket \text{Expression} "<" \text{Expression} \rrbracket \mid$   
 $\llbracket \text{Expression} "\leq" \text{Expression} \rrbracket \mid$   
 $\llbracket \text{Expression} ">" \text{Expression} \rrbracket \mid$   
 $\llbracket \text{Expression} "\geq" \text{Expression} \rrbracket \mid$   
 $\llbracket \text{Expression} "\neq" \text{Expression} \rrbracket$

## Semântica

A parte de Z é responsável por definir o estado do sistema, a inicialização do mesmo e as operações a serem executadas. A ordem das operações será determinada pela parte de CSP enquanto na parte de Z, apenas as modificações em termos de mudança de estado estarão especificadas. Toda as construções são baseadas em parágrafos, predicados e expressões.

- $\backslash begin\{schema\}$  e  $\backslash end\{schema\}$  representam a criação de um esquema de Z. Um esquema pode ser compreendido como um bloco de descrição representando o estado do sistema ou uma operação presente no mesmo.
- *SchemaText* constitui o corpo de um esquema, contendo declarações, pré e pós-condições.
- *DeclPart* representa a parte de declaração de um esquema, que podem ser tipadas (*Expression* : *Expression*) ou não (*Expression*).
- *Predicate* permite construir os predicados (pré e pós-condições) dos esquemas.
- *true* e *false* representam o conjunto dos possíveis valores booleanos em Z.
- *Predicate NL Predicate* representa uma seqüência de predicados entendida como uma operação lógica AND.
- *Expression* representa os diferentes tipos de expressões que podem ser construídas em Z.
- *Expression'* representa o estado final de uma variável (ou esquema).
- *Expression!* é usada nas declarações de parâmetros de saída (*output*) gerado por um esquema.
- *Expression?* é usada nas declarações de parâmetros de entrada (*input*) de um esquema.
- *Expression = Expression* representa uma atribuição.
- $\backslash Delta \text{ Expression}$  significa a mudança do estado do sistema.
- $\backslash Xi \text{ Expression}$  significa a preservação do estado do sistema.
- *NUMBER* representa um número .
- *ArithExpression* representa as expressões aritméticas (soma, subtração, multiplicação e divisão).
- *RelationalExpression* representa as expressões relacionais (menor, menor que, maior, maior que, diferente).

## 5 $CSP_Z$ como máquina de estados

Uma especificação de  $CSP_Z$  pode ser vista como uma máquina de estados. Cada esquema representa uma operação que é realizada dada a validade de sua pré-condição.

A parte de CSP estabelece a ordem dos processos e a parte de Z trata das mudanças de estado. Na execução de  $CSP_Z$ , primeiramente o esquema de inicialização é executado. Em seguida, o sistema procura executar as duas partes independentemente sincronizando-as quando necessário. Após a inicialização, todos os esquemas de Z que possuem a pré-condição verdadeira num determinado estado ficam habilitados para execução. O sistema escolhe um deles, executa, muda o estado (possivelmente) e comunica um evento. Essa comunicação acontece de forma sincronizada com a parte de CSP, ou seja, apenas esquemas que produzem eventos aceitos pela parte de CSP são executados. Dessa forma, a parte de CSP

indiretamente escolhe qual esquema executar, através da espera por um evento específico. Se dois ou mais esquemas de  $Z$  podem comunicar esse evento e estão habilitados (pré-condição verdadeira) então um dos dois poderá executar (escolha não-determinística).

A parte de  $Z$  limita-se apenas a tratar estruturas de dados, parâmetros das operações e mudar o estado do sistema. A cada execução de um esquema, todas as operações do sistema (exceto a inicialização) tornam-se disponíveis novamente.

## 5.1 Exemplo simples

Utilizando o exemplo mostrado em 4.1, podemos ter uma visão de  $CSP_Z$  como uma máquina de estados.

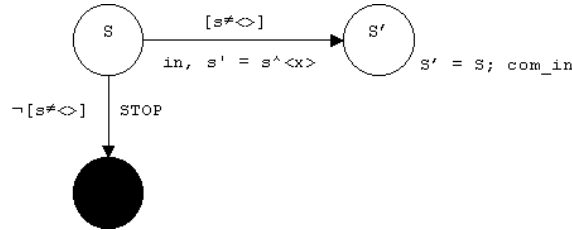


Figura 1: Visão de Máquina de Estados

A execução de um esquema *com\_in* pode ser vista da seguinte forma: num estado  $S$ , caso a pré-condição de *com\_in* ( $s! = \langle \rangle$ ) seja verdadeira e a parte de CSP esteja pronta para aceitar o evento *in*, então *com\_in* comunica esse evento e muda o estado do sistema ( $s' = s \wedge \langle x \rangle$ ). Caso a pré-condição não seja verdadeira, então o esquema não fica habilitado para executar, ou seja, se a parte de CSP estiver esperando por um evento a ser gerado por *com\_in*, neste caso, originará uma situação de deadlock (STOP).

## 6 Sintaxe Abstrata

### 6.1 Especificação

- (1) Specification =  $\llbracket$  "spec" Identifier CspPart ZPart "end\_spec" Identifier  $\rrbracket$

### 6.2 Parte de CSP

- (1) CspPart =  $\llbracket$  ChannelDeclaration\* ProcessDeclaration\* MainDeclaration  $\rrbracket$
- (2) ChannelDeclaration =  $\llbracket$  "channel" Identifier  $\langle$  , Identifier  $\rangle^*$  ":" Type  $\rrbracket$
- (3) ProcessDeclaration =  $\llbracket$  Identifier "=" Process  $\rrbracket$
- (4) MainDeclaration =  $\llbracket$  "main" "=" Process  $\rrbracket$
- (5) Type =  $\llbracket$  "[" "]"  $\rrbracket$  |  $\llbracket$  "[" Identifier ":" Identifier  $\langle$  , Identifier ":" Identifier  $\rangle^*$  "]"  $\rrbracket$
- (6) Process =  $\llbracket$  STOP  $\rrbracket$  |  $\llbracket$  SKIP  $\rrbracket$  |  $\llbracket$  Identifier  $\rrbracket$  |  $\llbracket$  Event  $\rightarrow$  Process  $\rrbracket$  |  $\llbracket$  Process ||| Process  $\rrbracket$  |  $\llbracket$  Process  $\square$  Process  $\rrbracket$  |  $\llbracket$  Process  $\sqcap$  Process  $\rrbracket$  |  $\llbracket$  Process || Process  $\rrbracket$    
 SetEvent

- (7) Event =  
 [[ Identifier ]] |  
 [[ Identifier "?" Identifier ]] |  
 [[ Identifier "!" Identifier ]]
- (8) SetEvent =  
 [[ "{" ]]  
 [[ "{" Event < , Event > "\*" "]" ]  
 [[ "{ | " Event < , Event > "\*" " | }" ]]

### 6.3 Parte de Z

- (1) ZPart = [[ Paragraph\* ]]
- (2) Paragraph = [[ "\begin{schema}" "{" Identifier "}" SchemaText "\end{schema}" ]]
- (3) SchemaText = [[ DeclPart < "\where" Predicate > ]]
- (4) DeclPart = [[ Declaration < NL DeclPart > ]]
- (5) Declaration =  
 [[ Expression ":" Expression ] |  
 [[ Expression ]]
- (6) Predicate =  
 [[ "true" ] |  
 [[ "false" ] |  
 [[ Expression ] |  
 [[ Predicate NL Predicate ]]
- (7) Expression =  
 [[ Identifier ] |  
 [[ Expression "'" ] |  
 [[ Expression "?" ] |  
 [[ Expression "!" ] |  
 [[ Expression "=" Expression ] |  
 [[ "\Delta" Expression ] |  
 [[ "\Xi" Expression ] |  
 [[ NUMBER ] |  
 [[ ArithExpression ] |  
 [[ RelationalExpression ]]
- (8) ArithExpression =  
 [[ Expression "\*" Expression ] |  
 [[ Expression "/" Expression ] |  
 [[ Expression "+" Expression ] |  
 [[ Expression "-" Expression ]]
- (9) RelationalExpression =  
 [[ Expression "<" Expression ] |  
 [[ Expression "<=" Expression ] |  
 [[ Expression ">" Expression ] |  
 [[ Expression ">=" Expression ] |  
 [[ Expression "≠" Expression ]]

## 7 Entidades semânticas

**includes:** Action Notation

A especificação de  $CSP_Z$  utilizando Semântica de Ações exigiu a definição de algumas entidades. Outras tiveram sua existência simplesmente assumida. Isso pode representar perda completa do formalismo. Uma vez que a linguagem vai ser definida em termos de ações, para toda entidade presente em  $CSP_Z$ , existe uma ação correspondente fornecendo-lhe um significado preciso. Essa “quebra” de formalidade será justificada ao final do trabalho.

### 7.1 Ações

**introduces:** put, create-channel-agent, create-process-agent, choose next event in, create-schema-abstraction, create-body-of-z-agent .

- **put**  $\_$  [in  $\_$  status] :: yielder, yielder  $\rightarrow$  action.  
Modifica o *status* de um determinado agente. os possíveis valores para o *status* de um process-agent são: ACTIVE e WAIT.
- **create-channel-agent** [identified by token  $\_$ ] [accepting type  $\_$ ] :: yielder, yielder  $\rightarrow$  action.  
Ação que realiza a criação de um channel-agent, porém sem colocá-lo em execução. Cada agente representando um canal será identificado por um token e poderá aceitar um determinado tipo de dado. Assim, **create-channel-agent** [identified by token  $c$ ] [accepting type *Int*] criará um agente representando o canal  $c$  através do qual trafegam valores inteiros (*Int*).
- **create-process-agent** [identified by token  $\_$ ] [containing as body  $\_$ ] :: yielder, yielder  $\rightarrow$  action.  
Cria um agente para representar um processo em CSP identificado por um token e contendo uma computação a ser executada. Por exemplo, **create-process-agent** [identified by token  $P$ ] [containing as body  $P'$ ] criará um agente representando o processo  $P$  cuja computação é  $P'$ . Neste caso,  $P'$  contém todos os passos a serem executados pelo agente (eventos a serem comunicados, dados a serem lidos/escritos etc).
- **create-schema-abstraction** [identified by token  $P$ ] [containing as body  $P'$ ]  
Cria uma abstração de um esquema, identificada pelo nome do esquema e contendo o corpo de esquema a ser executado. Por exemplo: **create-schema-abstraction** [identified by Init] [containing as body an abstraction], cria uma abstração representando o esquema de inicialização cuja semântica está encapsulada em uma abstração.
- **choose next event in**  $\_$  :: yielder  $\rightarrow$  action.  
Retorna o proximo evento de um processo a ser executado/comunicado. Em CSP, os processos são construídos através de eventos que os mesmos comunicam. A seqüência de eventos de um processo pode ser infinita e a execução deve seguir uma determinada ordem. Exemplo:

$$P = a?x \rightarrow b!y \rightarrow P$$

Executando a ação **choose next event in alphabet of P** duas vezes retornará os eventos  $a?x$  e  $b!y$  nesta ordem.

- **create-body-of-z-agent**  
Cria o corpo do agente que vai representar a parte de Z da especificação ao. Esta ação compõe-se de outras ações mais simples da seguinte forma:
  - create-body-of-z-agent :: action .

```

(1) create-body-of-z-agent =
    give abstraction of
    | unfolding
    | | give current z-abstractions
    | then
    | | | choose an z-abstraction [in the given list] [with pre-condition true] [neither State nor Init]
    | | | and then
    | | | | enact the given abstraction
    | | | then
    | | | | unfold
    | or
    | | unfold

```

## 7.2 Dados

**needs:** data

**introduces:**

channel-agent, process-agent, z-agent, z-abstraction, z-abstractions, sync-message, event, agent .

- **channel-agent** : agent.

Agente representando um canal de comunicação. Este agente trata de todos os detalhes de sincronização entre processos, ou seja, é responsável por mandar mensagens aos processos interessados nos eventos que ele suporta. Cada channel-agent é identificado por um nome e suporta o tráfego de algum tipo de dado. Existem diversas formas de se representar um canal de comunicação. Em [14], canais são modelados como células da memória, tornando o tratamento de controle de concorrência mais fácil para a linguagem modelada (ML). Já em [10], canais possuem sua modelagem abstraída e um processo comunica-se diretamente com outro processo. Embora seja muito difícil, na prática, encontrar sistemas que precisem sincronizar mais de dois processos, a última abordagem limita a comunicação apenas a dois processos.

- **process-agent** : agent.

Agente representando um processo em CSP. Um processo é composto de eventos a serem executados por agentes. Os eventos são enviados para um channel-agent que será encarregado por comunicar o evento a outros processos interessados. Uma vez esperando pela sincronização de um evento, um process-agent passa a ter um *status* (ACTIVE ou WAIT). Em WAIT, o agente só progride quando recebe uma mensagem de sincronização relativo a um evento específico. Em ACTIVE, o agente progride sua execução independente de sincronização.

- **z-agent** : agent.

Agente representando toda a parte de Z da especificação. Este agente deve conhecer todas as operações do sistema construídas na parte de Z, para então executar alguma delas quando necessário. Ele também sincroniza com um process-agent representando a parte de CSP. Para o exemplo mostrado em 4.1, existe um process-agent (*main*) aceitando os eventos *in?x* e *out!y* e um z-agent (*z\_csp*) contendo os esquemas *com\_in* e *com\_out*, que executam de forma sincronizada através dos eventos *in?x* e *out!y*, respectivamente.

- **z-abstraction** : abstraction.

Um *subsort* de abstraction. Define abstrações de esquemas (abstrações identificadas por um nome e que contêm o corpo do esquema a ser executado). Serve para representar as operações definidas na parte de Z.

- **z-abstractions** = list of z-abstraction.

Uma lista de contendo todas as abstrações dos esquemas definidos na parte de Z. Esta estrutura é utilizada por um z-agent quando vai ser posto em execução (as operações são oferecidas pelo sistema devem ser reconhecidas pelo z-agent).

- **sync-message** : message.

Mensagem de sincronização enviada por um process-agent a um channel-agent, para informar que

o processo deve executar determinado evento de forma sincronizada. Depois disso, o channel-agent se encarrega de sincronizar os demais processos interessados no evento.

- **event**  $\leq$  distinct-datum.  
Tipo de dado representando eventos de comunicação. Um evento é um dado que guarda o nome do canal onde deve ocorrer e o tipo de evento (entrada, saída, ambos).
- **agents** = list of agent.  
Lista contendo todos os agentes de uma especificação em  $CSP_Z$  (channel-agent, process-agent, z-agent).

### 7.3 Produtores

**introduces:** current agents, current z-abstractions, channel of, alphabet of .

- **current agents.**  
Retorna a lista contendo todos os agentes criados a partir da especificação.
- **current z-abstractions.**  
Retorna a lista contendo todas as abstrações de esquemas declarados na especificação.
- **channel of** \_ .  
Retorna o channel-agent representando o canal associado a um evento.
- **alphabet of** \_ .  
Retorna uma lista ordenada de eventos de um determinado processo.

## 8 Funções Semânticas

**needs:** Sintaxe Abstrata , Entidades Semânticas .

### 8.1 Especificações

**introduces:** run \_ , init \_ , execute-agents \_ , execute-channel-agents \_ , execute-channel-agent \_ , execute \_ , execute-z-agent \_ .

- run \_ :: Specification  $\rightarrow$  action .

(1) run  $\llbracket$  "spec"  $I_1$ :Identifier  $C$ :CspPart  $Z$ :ZPart "end\_spec"  $I_2$ :Identifier  $\rrbracket$  =

```

| elaborate-csp C
| and then
| elaborate-z Z
| then
| | give current z-abstractions
| | then
| | | init the given list
| | | and then
| | | | give current agents
| | | | then
| | | | execute-agents the given list

```

- init :: yielder  $\rightarrow$  action .

(2) init L =

```

| choose an abstraction [in L] [identified by "Init"]
| then
| enact the given abstraction

```



- `execute-agents :: yielder → action .`

```
(3) execute-agents L =
    | execute-channel-agents in L and regive
    then
    | | choose an agent [in the given list] [identified by "main"]
    | | then
    | | execute the given agent
    and
    | | choose an agent [in the given list] [identified by "z_csp"]
    | | then
    | | execute-z-agent the given agent
```

- `execute-channel-agents :: list of agents → action .`

```
(4) execute-channel-agents L =
    | check (L is () ) then complete
    or
    | | check (not L is () )
    | | and then
    | | | give first L
    | | | and then
    | | | | check (the given agent is a channel-agent)
    | | | | and then
    | | | | | execute-channel-agent the given agent
    | | | | | and then
    | | | | | give rest L then execute-channel-agents the given list
    | | or
    | | | check (not the given agent is a channel-agent) then complete
```

- `execute-channel-agent _ :: channel-agent → action .`

```
(5) execute-channel-agent A =
    unfolding
    | | receive a sync-message [from any agent] [containing an avent]
    | | and then
    | | | give contents of the given message
    | | and then
    | | | choose an agent [interested in the given event]
    | | | and then
    | | | | send a message [to the given agent] [containing the given event]
    | | | | then
    | | | | unfold
    | | or
    | | | send a message [to the sender agent] [containing the given event]
    | | | then
    | | | unfold
```

- `execute _ :: agent → action .`

```
(6) execute A =
    unfolding
    | choose next event in alphabet of A
    then
    | | check (status of A is ACTIVE)
    | | and then
    | | | send a sync-message [to channel of the given event] [containing the given event]
    | | | then
    | | | put [agent A] [in WAIT status ]
    | | | then
    | | | unfold
    | or
    | | check (status of A is WAIT)
    | | and then
    | | | receive a message [from a channel-agent] [containing the given event]
    | | | then
    | | | put [agent A] [in ACTIVE status]
    | | | and then
    | | | unfold
```

- `execute-z-agent _ :: agent → action .`

```
(7) execute-z-agent A =
    | give the abstraction [contained in A]
    then
    | enact it
```

## 8.2 Parte de CSP

**introduces:** `elaborate-csp _ , elaborate-channel _ , elaborate-process _ , elaborate-main _ .`

- `elaborate-csp _ :: CspPart → action .`

```
(1) elaborate-csp [ [ Ch:ChannelDeclaration* Pr:ProcessDeclaration* M:MainDeclaration ] ] =
    | elaborate-channel Ch
    then
    | | elaborate-process Pr
    | | then
    | | elaborate-main M
```

- `elaborate-channel _ :: ChannelDeclaration → action .`

```
(2) elaborate-channel [ [ "channel" I:Identifier ⟨ , Identifier ⟩* ":" T:Type ] ] =
    create-channel-agent [identified by token I] [accepting type T]
```

- `elaborate-process _ :: ProcessDeclaration → action .`

```
(3) elaborate-process [ [ I:Identifier "=" P:Process ] ] =
    | give alphabet of P
    then
    | create-process-agent [ identified by token I ] [ containing as body the given alphabet ]
```

- `elaborate-main _ :: ChannelDeclaration → action .`

```
(4) elaborate-main [ [ "main" "=" P:Process ] ] =
    | give alphabet of P
    then
    | create-process-agent [identified by token "main"] [containing as body the given alphabet ]
```

### 8.3 Parte de Z

**introduces:** `elaborate-z _` , `elaborate-par _` .

- `elaborate-z _ :: ZPart → action` .

(1) `elaborate-z [ [ P:Paragraph* ] ] =`  
`| elaborate-par P`  
`and then`  
`| | create-body-of-z-agent`  
`| then`  
`| | create-process-agent [ identified by token "z_csp" ] [ containing as body the given abstraction ]`

- `elaborate-par _ :: Paragraph → action` .

(2) `elaborate-par [ [ "\begin{schema}" " {" I:Identifier " } " S:SchemaText "\end{schema}" ] ] =`  
`| abstraction-from-schema-text S`  
`then`  
`| create-schema-abstraction [identified by token I] [containing as body then given abstraction]`

A função `abstraction-from-schema-text` não foi especificada porque envolve primeiro a elaboração das declarações e predicados contidos no corpo de um esquema. Além da gramática de Z ser muito extensa, o objetivo deste trabalho é apenas especificar a sincronização entre as partes de CSP e Z de uma especificação escrita em *CSP<sub>Z</sub>*.

## 9 Exemplo completo

Nesta seção, o exemplo mostrado em 4.1, é convertido em Ações Semânticas através das Funções Semânticas apresentadas em 8. A parte de Z da especificação é escrita em  $\text{\LaTeX}$  [11].

```
run [
spec Buffer
  channel in, out : [x:Int]
  main = in?x -> main [] out!y -> main
  \begin{schema}{State}
    s : \seq \num
  \end{schema}
  \begin{schema}{com\_out}
    \Delta State \\\
    x! : \nat
  \where
    s \neq \emptyseq \\\
    s' = tail(s) \\\
    x! = head(s)
  \end{schema}
  \begin{schema}{Init}
    State'
  \where
    s' : \emptyseq
  \end{schema}
  \begin{schema}{com\_in}
    \Delta State \\\
    x? : \nat
  \where
    s' = s \cat \lseq x? \rseq
  \end{schema}
end_spec Buffer
```

]] =

```

| elaborate-csp [[
|   channel in, out : [x:Int]
|   main = in?x → main [] out!y → main]]
and then
| elaborate-z [[
|   \begin{schema}{State}
|     s : \seq \num
|   \end{schema}
|
|   \begin{schema}{Init}
|     State'
|   \where
|     s' : \emptyseq
|   \end{schema}
|
|   \begin{schema}{com\_out}
|     \Delta State \\\
|     x! : \nat
|   \where
|     s \neq \emptyseq \\\
|     s' = tail(s) \\\
|     x! = head(s)
|   \end{schema}
|
|   \begin{schema}{com\_in}
|     \Delta State \\\
|     x? : \nat
|   \where s' = s \cat \lseq x? \rseq
|   \end{schema} ]]
then
| give current z-abstractions
then
| init the given list
and then
| give current agents
then
| execute-agents the given list

```

Antes de tudo, as declarações das partes de CSP e Z precisam ser elaboradas. Logo em seguida a inicialização deve ser realizada para depois os agentes executarem de forma síncrona.

```

elaborate-csp [[
  channel in, out : [x:Int]
  main = in?x → main [] out!y → main
]] =
| elaborate-channel [[ channel in, out : [x:Int] ]]
then
| elaborate-process [[ ]]
then
| elaborate-main [[ main = in?x → main [] out!y → main ]]

```

A elaboração da parte de CSP compreende a elaboração dos canais de comunicação, seguida da elaboração dos processos auxiliares e finalmente a elaboração do processo principal.

```

elaborate-channel [ channel in, out : [x:Int] ] =
  create-channel-agent [identified by token in] [accepting type Int]

```

A elaboração de canais compreende a criação de agentes representando os canais de comunicação declarados. Neste caso, dois agentes serão criados: *in* e *out* para representar os canais de comunicação *in* e *out*, respectivamente. Apenas encontra-se omissa a criação do agente *out*.

```

elaborate-main [ main = in?x → main [] out!y → main ] =
  | give alphabet of (in?x → main [] out!y → main)
  then
  | create-process-agent [identified by token "main"] [containing as body the given alphabet ]

```

A elaboração do processo *main* cria um agente para representar o processo principal da parte de CSP. A elaboração dos processos auxiliares da parte de CSP foi omitida porque nenhum outro processo foi declarado na especificação, ou seja, da parte de CSP, apenas um agente foi criado.

```

elaborate-z [
  \begin{schema}{State}
    s : \seq \num
  \end{schema}

  \begin{schema}{Init}
    State'
  \where
    s' : \emptyseq
  \end{schema}

  \begin{schema}{com\_out}
    \Delta State \
    x! : \nat
  \where
    s \neq \emptyseq \
    s' = tail(s) \
    x! = head(s)
  \end{schema}

  \begin{schema}{com\_in}
    \Delta State \
    x? : \nat
  \where
    s' = s \cat \lseq x? \rseq
  \end{schema}
] =
  | elaborate-par State
  and then
  | | create-body-of-z-agent
  | then
  | | create-process-agent [ identified by token "z_csp" ] [ containing as body the given abstraction ]

```

O processamento da parte de Z, consistem em elaborar todas as declarações de esquemas (estado, inicialização e operações). Cada um deles corresponde a uma abstração especial que vai ser executada quando solicitada. Ao final, um processo chamado *z\_csp* vai representar a parte de Z. Este processo reconhece todas as operações definidas pelos esquemas. A elaboração dos demais esquemas não foram detalhadas.

```

elaborate-par [
  \begin{schema}{State}
    s : \seq \num
  \end{schema}
] =
| abstraction-from-schema-text [ s : \seq \num ]
then
| create-schema-abstraction [identified by token State] [containing as body then given abstraction]

```

O trecho anterior mostra a criação de uma abstração de um esquema. Para cada esquema declarado na especificação existe uma abstração.

```

create-body-of-z-agent =
  give abstraction of
  | unfolding
  | | give current z-abstractions
  | | then
  | | | choose an z-abstraction [in the given list] [with pre-condition true] [neither State nor Init]
  | | | and then
  | | | | enact the given abstraction
  | | | then
  | | | | unfold
  | | or
  | | | unfold

```

A função acima cria uma abstração encapsulando o corpo do processo *z\_csp*. Dado que, no estado atual, current z-abstractions retorna uma lista contendo as abstrações dos esquemas *State*, *Init*, *com\_in* e *com\_out*, a ação acima, quando executada, apenas ativa abstrações de esquemas que não sejam nem *State* nem *Init*.

```

init L =
  | choose an abstraction [in L] [identified by "Init"]
  then
  | enact the given abstraction

```

A partir das abstrações dos esquemas, esta ação procura e executa a abstração que representa a inicialização do sistema.

```

execute-agents L=
  | execute-channel-agents in L and regive
  then
  | | choose an agent [in the given list] [identified by "main"]
  | | then
  | | | execute the given agent
  | and
  | | choose an agent [in the given list] [identified by "z_csp"]
  | | then
  | | | execute-z-agent the given agent

```

Depois da inicialização, os agentes são postos em execução. Primeiramente os agentes representando os canais de comunicação, depois os agentes representando as partes de CSP e Z.

```

execute-channel-agents L =
  | check (L is () ) then complete
or
  | | check (not L is () )
  | then
  | | give first L
  | | then
  | | | check (the given agent is a channel-agent)
  | | | then
  | | | | execute-channel-agent the given agent
  | | | then
  | | | | give rest L then execute-channel-agents the given list
  | | or
  | | | check (not the given agent is a channel-agent) then complete

```

Dada uma lista de agentes, apenas os agentes representando os canais são postos em execução. Neste caso, L possui os seguintes agentes: *in*, *out*, *main*, *z\_csp*. Nesta etapa, apenas *in* e *out* serão postos em execução. Naturalmente, para os processos se comunicarem, os canais de comunicação devem ser estabelecidos.

```

execute-channel-agent in =
  unfolding
  | | receive a sync-message [from any agent] [containing an event]
  | | and then
  | | | give contents of the given message
  | | and then
  | | | choose an agent [interested in the given event]
  | | | and then
  | | | | send a message [to the given agent] [containing the given event]
  | | | then
  | | | | unfold
  | | or
  | | | send a message [to the sender agent] [containing the given event]
  | | then
  | | | unfold

```

```

execute-channel-agent out =
  unfolding
  | | receive a sync-message [from any agent] [containing an event]
  | | and then
  | | | give contents of the given message
  | | and then
  | | | choose an agent [interested in the given event]
  | | | and then
  | | | | send a message [to the given agent] [containing the given event]
  | | | then
  | | | | unfold
  | | or
  | | | send a message [to the sender agent] [containing the given event]
  | | then
  | | | unfold

```

Um agente representando um canal de comunicação se comporta como um receptor e repassador de mensagens específicas. Automaticamente, o agente fica esperando por mensagens de sincronização em um evento particular. Quando isso ocorre, os dois agentes (processos) interessados são avisados.

```

execute the agent [identified by "main"] =
  unfolding
  | choose next event in alphabet of the given agent
  then
    | check (status of the given agent is ACTIVE)
    and then
      | send a sync-message [to channel of the given event] [containing the given event]
      then
        | put [agent "main"] [in WAIT status ]
      then
        | unfold
    or
    | check (status of the given agent is WAIT)
    and then
      | receive a message [from a channel-agent] [containing the given event]
      then
        | put [agent identified by "main"] [in ACTIVE status]
        and then
          | unfold

```

Põe em execução o agente representando a parte de CSP.

```

execute-z-agent the agent [identified by "z_csp"]=
  | give the abstraction [contained in the agent]
  then
  | enact it

```

Põe em execução o agente representando a parte de Z. Depois disso, os agentes (*main* e *z\_csp*) realizam suas tarefas sincronizando em eventos específicos utilizando os canais (*in* e *out*).

## 10 Conclusões

Este trabalho abordou a definição da semântica de  $CSP_Z$  em termos de Semântica de Ações. Na Seção 1, foi dada uma visão geral sobre a Notação de Ações e suas facetas. A Seção 2 mostrou a linguagem CSP de forma resumida. A Seção 3 deu um a visão geral sobre a linguagem Z. Na Seção 4, foi mostrada uma técnica importante em Métodos Formais: a integração de formalismos para especificação de sistemas. A notação  $CSP_Z$ , integração semântica entre CSP e Z, foi apresentada através de um exemplo simples. Finalmente, nas Seções 6, 7 e 8 foram mostradas a Sintaxe Abstrata, Entidades Semânticas e Funções Semânticas, respectivamente. Após isso, a linguagem  $CSP_Z$  passou a ter seu comportamento definido em termos de Ações Semânticas.

Alguns detalhes deste trabalho merecem destaque:

- Natureza assíncrona da faceta comunicativa da Notação de Ações. Todas as facetas da Notação de Ações, exceto a Comunicativa, dizem respeito a um único agente em execução. Todas as comunicações realizadas por um agente são assíncronas. Certos modelos de concorrência como CCS [7] e CSP, que utilizam comunicação síncrona, tornam-se difíceis de especificar utilizando um *framework* assíncrono.
- Natureza *single-agent* da faceta Comunicativa. A definição do comportamento, em termos de ações semânticas, para modelos multi-agentes, pode ser muito difícil de implementar [15]. Linguagens como CSP necessitam de muitos processos interagindo entre si por comunicação. Vários agentes precisam ser criados e postos em execução. Nas referências consultadas, não foi encontrada nenhuma forma explícita de instanciar e executar mais de um agente. Em [13], um agente interage através de primitivas de comunicação admitindo que o outro agente já está em execução.



- Complexidade da linguagem modelada. Pelo fato de integrar duas linguagens,  $CSP_Z$  passa também a incorporar a complexidade das duas notações. Existem inúmeras formas de construir termos na linguagem, atividade que requereria muito tempo para ser completamente descrita em termos de Ações Semânticas. Algumas ações e funções semânticas precisaram ter sua existência assumida, visando simplificar o trabalho e manter o objetivo principal: especificar a sincronização entre as partes de CSP e Z de uma especificação escrita em  $CSP_Z$ .
- Carência de trabalhos na área. Poucos são os trabalhos envolvendo definição da Semântica Operacional de CSP utilizando Ações Semânticas. Alguns trabalhos foram feitos considerando-se a versão antiga de CSP [16]. A versão da linguagem utilizada atualmente é a definida em [4], para a qual existem ferramentas de auxílio à utilização da linguagem. Dois outros trabalhos mostrados em [3, 10] procuram tratar o assunto, mas não abordam a notação por completo. Em [5], encontram-se definidas a Semântica Denotacional e Operacional de CSP, mas não em termos de Semântica de Ações.
- Forma mais geral de definição do protocolo de comunicação. Embora o presente trabalho se preocupe em definir apenas o comportamento da sincronização entre CSP e Z, a forma como CSP foi modelada permite o tratamento da comunicação por eventos de forma mais geral do que em [10], onde um processo comunica-se apenas com outro processo, eliminando a existência de um agente para representar o canal de comunicação.
- Ausência de suporte à faceta Comunicativa no ABACO [8]. Devido à necessidade de estender a Notação de Ações e a ausência de implementação das primitivas de comunicação no ABACO, as funções semânticas definidas neste trabalho não puderam ser testadas.
- Carência de pequenos ajustes no ABACO:
  1. O sistema não implementa a ação *choose*<sub>-</sub> e rejeita alguns símbolos especiais como ?, ' e \.
  2. Uso de restrições na Notação de Ações (termos escritos entre [ e ] ) não são aceitos.
  3. Alguns problemas com expansão de algumas regras da sintaxe abstrata.

Apesar de possuir suas limitações, foi possível representar os novos tipos de dados, produtores e ações de  $CSP_Z$ . Algumas ações não puderam ser representada sob forma ações mais simples. Deveriam existir dentro da própria Notação de ações. As funções semânticas foram escritas, porém não puderam ser testadas, já que o suporte de execução à faceta Comunicativa não foi implementado.

Definir uma semântica completa para  $CSP_Z$  requer um grande trabalho. Antes de especificar como as partes de CSP e Z sincronizam, é necessário ter a semântica de CSP e Z definidas anteriormente. De acordo as dificuldades expostas acima, a Notação de Ações deveria ser estendida para dar um maior suporte à descrição de sistemas multi-agentes e prover mais ações primitivas que permitissem modelar comunicação síncrona e agentes de forma mais natural. Apesar dessa carência, a Notação de Ações é um formalismo muito interessante para a descrição de Semântica Operacional.

Os trabalhos, embora se mostrem incompletos, possuem forte contribuição para a difusão de Métodos Formais na indústria de *software*. O ABACO auxilia bastante na definição da semântica de linguagens que não utilizam a faceta comunicativa. Em [15, 16], diferentes modelos de concorrência (CCS e CSP) tiveram sua semântica definida. Particularmente, CSP foi descrita mais detalhadamente de em [10], porém limitando a comunicação entre dois processos.

O presente trabalho é a primeira tentativa de definição de uma Semântica Operacional para  $CSP_Z$ , em termos de Semântica de Ações. Embora carente de melhorias e adaptações, o trabalho representa um passo importante na divulgação e consolidação de  $CSP_Z$  como uma linguagem formal e adequada para especificação de sistemas distribuídos.

## Referências

- [1] A. Mota. *Formalização e análise do SACI-1 em CSP-Z*. Dissertação de mestrado, Universidade Federal de Pernambuco, 1997.
- [2] 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.
- [3] A. Mota. *Modeling CSP as LTS's using Action Semantics*. Technical Report, Center of Informatics, Federal University of Pernambuco, Brazil, 1996.
- [4] A.W.Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
- [5] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, 1998.
- [6] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.
- [7] 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.
- [8] H. Moura and L. C Menezes. *The ABACO system: an Algebraic Based Action Compiler*. In Armando Martn Haeberer, editor, 7<sup>th</sup> International Conference on Algebraic Methodology and Software Technology, Springer-Verlag, 1999, pages 527–529.
- [9] H. Moura. *Action Semantics of Specimen*. Department of Informatics, Federal University of Pernambuco, Brazil, 1996.
- [10] L. Freitas and A. Cavalcanti and H. Moura. *Animating CSP<sub>M</sub> Using Action Semantics*. Center of Informatics, Federal University of Pernambuco, Brazil, 2001.
- [11] L. Lamport. *L<sup>A</sup>T<sub>E</sub>X. A Document Preparation System. User's Guide and Reference Manual*. Addison-Wesley, 1998.
- [12] M. Spivey. *The Z Notation: A Reference Manual, 2<sup>nd</sup> Edition*.
- [13] P. Mosses. *Action Semantics*. Department of Computer Science, Aarhus University, Denmark, 1992.
- [14] P. Mosses and M. Musicante. *An Action Semantics for ML Concurrency Primitives*. Technical Report RS-94-20, Computer Science Department, Aarhus University, Denmark, 1994.
- [15] P. Mosses. *ON THE ACTION SEMANTICS OF CONCURRENT PROGRAMMING LANGUAGES*. Computer Science Department, Aarhus University, Denmark, 1992.
- [16] S. Christensen and M. H. Olsen. *Action Semantics of Calculus of Communicating Systems and Communicating Sequential Process*. Technical Report, Aarhus University, Denmark, 1988.