Desenvolvimento formal utilizando Circus

David Paulo Pereira

25 de junho de 2004

1 Introdução

Ao longo das duas últimas décadas, observamos duas escolas produzindo técnicas formais para o desenvolvimento de software preciso, correto e conciso, aplicando duas abordagens distintas: uma focada na representação de estado e outra no comportamento do sistema. Linguagens como Z[4] e VDM[5] utilizam a primeira abordagem, baseada em modelos de estado, na qual a teoria matemática de conjuntos exerce fundamental importância na especificação dos objetos. Embora possível, as construções que descrevem aspectos comportamentais do sistema, como por exemplo, escolha, seqüência ou pararelismo, não são explicitamente suportadas por este tipo de linguagem.

Por outro lado, representantes da segunda abordagem, como CSP[1] e CCS[6], suportam construções utilizadas para descrever o comportamento do sistema. Infelizmente, não possuem meios concisos e elegantes para descrever aspectos referentes à representação de estado.

O trabalho de pesquisa desenvolvido teve como objetivo o estudo de uma linguagem de especificação e desenvolvimento formal de software que unisse as duas abordagens em uma só teoria, fazendo uso de construções para descrever tanto aspectos ligados à representação interna de estado como comportamentos observados externamente, decorrentes da comunicação, concorrência e mobilidade.

Muitas tentativas foram feitas para unir ambas as teorias, como por exemplo, combinações de Z com CCS[8], Z com CSP[11] e Object-Z com CSP[7]. Entretanto, nenhuma delas possuia uma teoria de cálculo de refinamentos associada. A falta de suporte ao desenvolvimento através de técnicas de refinamento motivou a criação de uma linguagem chamada *Circus*, que, em seu estágio atual de desenvolvimento, não fornece suporte à mobilidade.

Em *Circus*, os sistemas são caracterizados como processos, que agrupam construtores que descrevem as representações de dados (estado do processo), utilizando a notação Z; e o controle de comportamento, utilizando a álgebra de processos CSP. A semântica de *Circus*, apresentada em [9], baseia-se na teoria UTP (Unifying Theories of Programming)[2], que unifica diferentes paradigmas da ciência da computação para a programação de sistemas. Na seção 5, apresentaremos a estrutura sintática de *Circus* através de um programa exemplo. Porém antes, para facilitar o entendimento, serão apresentadas noções fundamentais de Z na seção 2, e de CSP na seção 3. Na seção 4, introduziremos algumas noções sobre refinamento, que será discutido no contexto de *Circus* na seção 6. Para finalizar, na seção 7 serão apresentados os conceitos de mobilidade junto com a linguagem π-calculus. Na seção 8 é feita uma conclusão sobre o trabalho desenvolvido e as perspectivas futuras.

2 Linguagem Z

Z é uma linguagem formal destinada à especificação de software. Como uma linguagem formal, possui um alfabeto definido(conjunto de símbolos), uma sintaxe precisa (conjunto de regras que validam uma expressão) e uma semântica também precisa (modo único para interpretar as expressões). Teoria de conjuntos e lógica de predicados dão o suporte matemático necessário à especificação concisa, correta e sem ambigüidades dos requisitos do sistema.

2.1 Schemas

A unidade de construção básica, denominada *schema*, define a representação matemática característica do Z, utilizada para especificar um tipo abstrato de dados, constituído por variáveis de estado e uma lista de operações que as modificam. O schema permite a organização da especificação de forma modular, hierárquica e reutilizável. Geralmente, é dividido em duas partes: *declaração* de variáveis e *predicado*, que restringe os valores que estas podem assumir. O texto de um schema pode ser escrito de duas formas distintas: horizontal ou vertical, conforme exibido abaixo, respectivamente.

$$Name \mathrel{\widehat=} [declaracao \mid predicado]$$

$$\begin{array}{|l} \hline \textit{Name} \\ declaracao \\ \hline predicado \\ \hline \end{array}$$

Suponha, por exemplo, uma casa de espetáculos que utiliza um software para a reserva de lugares na platéia em uma apresentação. Alguns dos lugares estarão reservados e outros ainda estarão vazios. Neste nível de abstração, não há a necessidade de considerarmos uma representação detalhada para lugar e cliente. Sendo assim, definiremos ambos como um tipo genérico de dados, que em Z. é feito da seguinte forma.

$$[Seat, Customer]$$

A bilheteria mantém o registro de quais lugares estão reservados (vendidos) para quais clientes. Isto pode ser representado como uma função parcial de *Seat* para *Customer* já que um mesmo lugar não pode ser reservado para dois clientes distintos. Em Z, descrevemos a função da seguinte forma.

$$sold \in Seat \nrightarrow Customer$$

Definimos como *seating* o conjunto de lugares da casa disponíveis em um dado espetáculo. Dessa forma, verificamos que o domínio de *sold* é um subconjunto de *seating*.

$$\operatorname{dom} sold \subseteq seating$$

A partir dos requisitos apresentados acima, uma bilheteria poderia ser representada por um schema na forma horizontal

$$BoxOffice \mathrel{\widehat=} [seating : \mathbb{P} Seat;\ sold : Seat \nrightarrow Customer \mid \operatorname{dom} sold \subseteq seating \land sold \in Seat \nrightarrow Customer]$$

ou por um schema na forma vertical.

$$\begin{array}{|l} \hline \textit{BoxOffice} \\ seating : \mathbb{P} Seat;\ sold : Seat \nrightarrow Customer \\ \hline \operatorname{dom} sold \subseteq seating \\ sold \in Seat \nrightarrow Customer \\ \hline \end{array}$$

2.2 Operações

Uma operação que modifica os valores de estado pode ser vista como uma relação sobre objetos do schema correspondente. A notação de schema fornece uma forma conveniente de descrevê-la:

o relacionamento entre estado anterior e estado posterior à operação é representado pelos objetos *State* e *State'*, respectivamente, como mostrado a seguir.

$$
\begin{array}{|l}
\hline
\mathit{Operation} \\
\mathit{State} \\
\mathit{State}' \\
\hline
\ldots \\
\hline
\end{array}
$$

A parte de predicado do schema caracteriza a operação, descrevendo o efeito sobre as variáveis de estado. Uma possível operação para o exemplo dado é a reserva (compra) de um lugar para um determinado espetáculo. Suponha que este lugar seja representado por $s?$, e o cliente interessado por $c?$. Para que a operação seja bem sucedida, s? precisa estar vago, ou seja

$s? \in \mathit{seating} \setminus \operatorname{dom} \mathit{sold}$

Após a operação, a relação *sold* precisa ser atualizada para indicar a venda.

$\mathit{sold}' = \mathit{sold} \cup \{s? \rightarrow c?\}$

Finalmente, a coleção dos lugares para o espetáculo não é modificada. Podemos encapsular esta operação em um schema chamado *Purchase*, que representa uma reserva bem sucedida de $s?$ pelo cliente $c?$.

$$
\begin{array}{|l}
\hline
\mathit{Purchase} \\
\mathit{BoxOffice} \\
\mathit{BoxOffice}' \\
\hline
s? \in \mathit{seating} \setminus \operatorname{dom} \mathit{sold} \\
\mathit{sold}' = \mathit{sold} \cup \{s? \rightarrow c?\} \\
\mathit{seating}' = \mathit{seating} \\
\hline
\end{array}
$$

Em Z, podemos substituir as duas cópias de um mesmo schema *Schema* dentro da operação por $\Delta\mathit{Schema}$, se a operação resultar em mudança de estado.

$$
\begin{array}{|l}
\hline
\Delta\mathit{Schema} \\
\mathit{Schema} \\
\mathit{Schema} \\
\hline
\end{array}
$$

De forma análoga, podemos substituir as duas cópias de um mesmo schema *Schema* dentro da operação por $\Xi\mathit{Schema}$, se a operação **não** resultar em mudança de estado.

$$
\begin{array}{|l}
\hline
\Xi\mathit{Schema} \\
\mathit{Schema} \\
\mathit{Schema}' \\
\hline
\end{array}
$$

Assim, poderíamos reescrever a operação acima:

$$
\begin{array}{|l}
\hline
\mathit{Purchase} \\
\Delta\mathit{BoxOffice} \\
\hline
s? \in \mathit{seating} \setminus \operatorname{dom} \mathit{sold} \\
\mathit{sold}' = \mathit{sold} \cup \{s? \rightarrow c?\} \\
\mathit{seating}' = \mathit{seating} \\
\hline
\end{array}
$$

3 Linguagem CSP

CSP (Communicating Sequencial Processes) é uma linguagem formal, baseada em teoria matemática, capaz de descrever o comportamento de um sistema concorrente através de uma coleção de leis algébricas. A idéia básica parte da decomposicão de um sistema em subsistemas menores operando concorrentemente e interagindo entre si, bem como com o meio ambiente.

O conceito de *evento* em CSP significa a ocorrência de uma açao de interesse para o sistema, de forma que o comportamento deste seja afetado. Seja, por exemplo, uma máquina automática de venda de chocolates, o sistema que queremos descrever. Os eventos que inicialmente poderiam ser considerados são os seguintes:

- *coin* - a inserção de uma moeda na máquina
- *choc* - a entrega do chocolate

Ao conjunto de nomes de evento que consideramos importante para para a descrição de um objeto damos o nome de *alfabeto*. Um processo é caracterizado por um padrão de comportamento, descrito em termos do conjunto de eventos contido em seu alfabeto. Seja *VMS* o processo que representa a máquina descrita acima. O alfabeto de *VMS* é representado por

$$\alpha VMS = \{coin, choc\}$$

Um processo de alfabeto A, que na realidade, é incapaz de participar de qualquer um dos eventos definidos, é representado por $STOP_A$. A seguir apresentaremos alguns dos operadores de CSP.

3.1 Prefixos

Seja x um evento e P um processo, então

$$(x \rightarrow P)$$

representa um objeto que primeiro participa de um evento x, comportando-se em seguida exatamente como especificado em P. Segue um exemplo da máquina descrita acima que serve dois clientes em seqüência antes de quebrar.

$$(coin \rightarrow (choc \rightarrow (coin \rightarrow (choc \rightarrow STOP_{\alpha VMS}))))$$

3.2 Recursão

CSP fornece uma notação para representar processos com comportamento de natureza recursiva. Veja o exemplo de um relógio que possui como único evento um *tick*. O alfabeto é descrito por $\alpha CLOCK = \{tick\}$ e a representação mais óbvia nos leva à seguinte equação

$$CLOCK = (tick \rightarrow CLOCK)$$

Substituindo-se o lado direito da equação pela própria equação, chega-se à $CLOCK = (tick \rightarrow (tick \rightarrow CLOCK))$, que após mais uma substituição, vira $CLOCK = (tick \rightarrow (tick \rightarrow (tick \rightarrow CLOCK)))$ e assim sucessivamente. Porém, como observado por [1] existem alguns problemas com esta notação, que substituiremos por uma mais adequada.

$$\mu X : A \bullet F(X)$$

onde X é uma variável local, A é um alfabeto e $F(X)$ é uma expressão denominada "guardada", significando um processo que deve iniciar-se com um prefixo. Um relógio perpétuo seria representado como

$CLOCK = \mu X : \{tick\} \bullet (tick \rightarrow X)$

3.3 Escolha

Muitos objetos comportam-se de acordo com a ação exercida pelo meio ambiente onde são colocados. Por exemplo, uma máquina automática de venda poderia oferecer chocolates e caramelos. É o cliente quem decide o que efetivamente comprar. Se x e y são eventos distintos

$(x \rightarrow P \mid y \rightarrow Q)$

descreve um objeto que participa do evento x ou do evento y. O exemplo da máquina de chocolates e caramelos $VMCT$ seria especificada da seguinte maneira

$VMCT = \mu X \bullet coin \rightarrow (choc \rightarrow X \mid toffee \rightarrow X)$

3.4 Paralelismo

Vimos no exemplo da máquina de venda que o cliente (meio ambiente) tem a opção de escolha sobre qual produto adquirir. Felizmente, o cliente também pode ser representado como um processo, com comportamento definido através das mesmas notações. Isto permite a análise de comportamento do sistema como um todo composto pelo processo e seu meio ambiente, interagindo um com o outro na medida que ambos vão sendo executados concorrentemente. De fato, é melhor esquecermos a distinção entre meio ambiente, processo e sistema.

Quando dois processos executam concorrentemente, a intenção usual é que eles interajam entre si. Estas interações podem ser vistas como eventos que exigem participação simultânea nos processos envolvidos, considerando que o alfabeto é idêntico para ambos. Dessa forma, se P e Q são processos com o mesmo alfabeto, introduzimos a notação

$P \parallel Q$

que denota um processo que se comporta como um sistema composto dos processos P e Q

3.5 Não determinismo

Algumas vezes, um processo disponibiliza uma variedade de comportamentos, mas o meio ambiente não tem como influenciar ou mesmo observar a seleção de alternativas. A escolha é feita de uma forma não determinística, ou seja, o fator de escolha é ignorado. Mas em alguns casos, dependendo da situação, o meio ambiente tem como controlar qual ação será executada. Para esse caso, introduziremos a operação

$(P \square Q)$

para denotar uma escolha não determinística se o primeiro evento aceito de P corresponder ao mesmo evento de Q. Caso os eventos sejam distintos, o meio ambiente tem como escolher a melhor opção.

3.6 Comunicação

Uma comunicação é um evento descrito pelo par $c.v$ onde c é o nome do canal no qual a comunicação acontece e v é o valor transmitido. Um processo que primeiro envia v para o canal c e depois comporta-se como P é definido por

$(c!v \rightarrow P)$

Um processo que inicialmente está preparado para receber um valor x, transmitido pelo canal c, comportando-se na seqüência como $P(x)$ é definido como

$(c?x \rightarrow P(x))$

Tomamos como exemplo um processo que repetidamente recebe um bit pelo canal *in*, repassando-o a diante pelo canal *out*

$COPYBIT = \mu X \bullet (in?x \rightarrow (out!x \rightarrow X))$

4 Refinamentos

Segundo [4], escrever uma especificação formal é uma atividade que realmente vale a pena devido aos benefícios proporcionados pelo entendimento correto de uma descrição concisa. Entretanto, seria interessante desenvolver a especificação de tal maneira que nos levasse a uma implementação possível. Esse processo de desenvolvimento da especificação passo a passo até chegarmos a um implementação executável do problema é chamado de *refinamento*.

Podemos refinar uma especificação formal adicionando mais informação. Por exemplo, podemos ser mais precisos na definição dos tipos de dados utilizados, ou nos detalhes de como um cálculo é realizado. Claramente, é importante que a nova e mais detalhada especificação seja consistente com a especificação anterior. Ou seja, o refinamento precisa estar correto.

O processo de refinamento da especificação envolve a remoção do não determinismo, ou incerteza. Uma especificação abstrata pode deixar em aberto algumas escolhas de design, que podem ser resolvidas com o refinamento, eliminando o não determinismo. Vários passos de refinamento podem ser necessários, sendo que cada passo precisa ser provado formalmente como forma de garantir a conformidade inicial. Dizemos que a especificação Q refina P através da seguinte notação

$P \sqsubseteq Q$

Aqui, uma especificação é encarada como um programa abstrato, onde sua *pré-condicao* define os estados no qual pode ser executado; e sua *pós-condição* descreve os estados finais w modificados, de acordo com a notação a seguir.

$w : [pre, post]$

A especificação pode ser refinada de duas formas diferentes: ou sua pós-condição é fortalecida ou sua pré-condição é enfraquecida, formalizada nas duas leis a seguir.

Lei 4.1 - Pós-condição fortalecida.
Se $post' \Rightarrow post$ então $w : [pre, post] \sqsubseteq w : [pre, post']$.

Lei 4.2 - Pré-condição enfraquecida.
Se $pre \Rightarrow pre'$ então $w : [pre, post] \sqsubseteq w : [pre', post]$.

5 Sintaxe do Circus

Nesta seção, apresentamos *Circus*[9], uma linguagem concorrente apropriada a refinamentos, baseada em CSP e que adiciona facilidades de especificação no estilo de Z. Aspectos de estado e comunicação são capturados na mesma especificação. *Circus* é uma linguagem apropriada não apenas para a especificação de sistemas concorrentes e reativos, mas também para o refinamento pois possui uma teoria associada. Da mesma maneira que nas especificações Z. os programas escritos em *Circus* são formados por uma seqüência de parágrafos.

$Program ::= CircusPar^*$

Aqui $CircusPar^*$ denota a possibilidade de uma lista, possivelmente vazia, de elementos da categoria sintática *CircusPar* de parágrafos *Circus*. Cada um desses paragráfos pode ser um parágrafo Z, aqui denotado pela categoria sintática *Par*, uma definição de canais, uma definição de conjunto de canais, ou uma definição de processo.

$CircusPar ::= Par \mid \mathbf{channel}\, CDecl \mid \mathbf{chanset}\, N == CSExp \mid ProcessDefinition$

A categoria sintática *Par* consiste de parágrafos Z, como definidos em [4]. A categoria sintática N é definida como identificadores válidos de Z.

5.1 Definição de Canais

Um processo possui a permissão de comunicar-se apenas com os canais previamente definidos na especificação. Uma definição de canal, representada pelo nome e pelo tipo do valor transmitido é mostrada a seguir.

$\mathbf{channel}\ in \colon \mathbb{N}$

Quando um canal não é usado para comunicar valores, apenas eventos de sincronização, a declaração é formada só pelo nome, elimininado-se o tipo. Podemos também utilizar a notação de schema para a declaração de canais. Tal notação agrupa as definições, não possuindo qualquer expressão na parte de predicado, de modo que a única restrição imposta é em relação ao tipo dos valores comunicados. Conjuntos de canais previamente definidos podem ser introduzidos em um parágrafo **chanset**.

5.2 Definição de Processos

Uma definição de processo declara um nome junto com uma especificação formada por uma seqüência de parágrafos de processo e uma ação sem identificação ao final, sendo delimitada por **begin** e **end**. Um parágrafo de processo pode ser um parágrafo Z ou uma definição de ação que juntos, definem o estado e o comportamento do sistema.

5.3 Exemplo de Processo Circus

As construções principais de *Circus* são ilustradas através do clássico exemplo de um *Buffer* reativo e limitado, apresentado em [9]. O tamanho máximo do buffer pode ser representado por uma constante estritamente positiva.

$\mid maxbuff : \mathbb{N}_1$

O processo utiliza dois canais para comunicação, um de entrada e o outro de saída.

$\mathbf{channel}\ input, output : \mathbb{N}$

Em *Circus*, utilizamos a notação Z para definir o estado do processo. As operações que modificam este estado são chamadas de ações, que são definidas através de schemas Z, operadores CSP ou linguagem de comandos "guardados"[12]. Em nosso exemplo, o processo *Buffer* encapsula duas componentes de estado: uma lista ordenada contendo o conteúdo e o tamanho dessa lista.

$\mathbf{process}\ Buffer \mathrel{\widehat=} \mathbf{begin}$

$\textbf{state}\ BufferState \mathrel{\widehat=} [buff : seq\mathbb{N};\ size : 0..maxbuff \mid size = \#buff \leq maxbuff]$

Inicialmente, o buffer está vazio, como apresentado pela ação de inicialização de estado.

$BufferInit \mathrel{\widehat=} [BufferState' \mid buff' = \langle\rangle \wedge size' = 0]$

O buffer recebe um valor na condição de existir espaço para seu armazenamento; neste caso, o elemento de entrada é adicionado à seqüência e o tamanho é incrementado. A especificação *InputCmd* descreve este comportamento de forma análoga ao Z, onde um schema é usado.

$InputCmd \mathrel{\widehat=} [\Delta BufferSate;\ x? : \mathbb{N} \mid size < maxbuff \wedge buff' = buff \frown \langle x? \rangle \wedge size' = size + 1]$

Input descreve a pré-condição para a comunicação através do canal *input*. Caso satisfeita, o schema *InputCmd* é utilizado para descrevr a mudança de estado.

$Input \mathrel{\widehat=} size < maxbuff\ \&\ input?x \rightarrow InputCmd$

Esta acao é "guardada" por $size < maxbuff$: caso a pré-condição não seja satisfeita, *Input* entra em deadlock. A ação $input?x \rightarrow InputCmd$ é um prefixo no estilo de CSP. Note que a variável de entrada x é introduzida, recebendo o valor transmitido pelo canal *input*. A ação *Output* é executada apenas quando há elementos no buffer; resultando na retirada do elemento presente na frente do buffer e na atualização de seu tamanho.

$OutputCmd \mathrel{\widehat=} [\Delta BufferState \mid size > 0 \wedge buff' = tail\ buff \wedge size' = size - 1]$
$Output \mathrel{\widehat=} size > 0\ \&\ output!(headbuff) \rightarrow OutputCmd$

A ação principal sem identificação ao final da descrição define o comportamento do processo em termos das ações aplicadas sobre o estado. No exemplo, esta ação inicializa o estado do buffer e, repetidamente, oferece uma escolha ao meio-ambiente para a entrada ou saída de um elemento.

$\bullet\ BufferInit;\ (\mu X \bullet (Input \mathrel{\Box} Output);\ X)$
end

Segue abaixo a descrição completa do processo *Buffer*.

$\mid maxbuff : \mathbb{N}_1$
$\textbf{channel}\ input, output : \mathbb{N}$
$\textbf{process} Buffer \mathrel{\widehat=} \textbf{begin}$
$\textbf{state} BufferState \mathrel{\widehat=} [buff : seq\mathbb{N};\ size : 0..maxbuff \mid size = \#buff \leq maxbuff]$
$BufferInit \mathrel{\widehat=} [BufferState' \mid buff' = \langle\rangle \wedge size' = 0]$
$InputCmd \mathrel{\widehat=} [\Delta BufferSate;\ x? : \mathbb{N} \mid size < maxbuff \wedge buff' = buff \frown \langle x? \rangle \wedge size' = size + 1]$
$Input \mathrel{\widehat=} size < maxbuff\ \&\ input?x \rightarrow InputCmd$
$OutputCmd \mathrel{\widehat=} [\Delta BufferState \mid size > 0 \wedge buff' = tail\ buff \wedge size' = size - 1]$
$Output \mathrel{\widehat=} size > 0\ \&\ output!(headbuff) \rightarrow OutputCmd$
$\bullet\ BufferInit;\ (\mu X \bullet (Input \mathrel{\Box} Output);\ X)$
end

O exemplo do *Buffer* mostrou como processos são construídos a partir de ações, mas estes podem também serem combinados através de operadores CSP, como por exemplo, a composição paralela. O processo construído dessa forma é obtido da conjunção dos estados dos processos constituintes e da combinação paralela de suas ações.

Um operador encontrado exclusivamente no *Circus* é o indexador: um processo $i : T \odot P$ comporta-se como P, utilizando, porém, canais diferentes. Para cada canal c de P, existe um canal

c_i correspondente que comunica um par de valores: o primeiro elemento corresponde a um índice, valor de T; e o segundo elemento ao valor originalmente comunicado através de c. A instanciação $(i : T \odot P)\lfloor e \rfloor$ comporta-se como P, porém, com o primeiro elemento do par possuindo o valor da expressao e. Existe também um operador de renomeação em *Circus*. Por exemplo, na expressão $P[c_1 := c_2]$, as comunicações de P pelo canal c_1 são realizadas através do canal c_2. Processos parametrizados também estão disponíveis. Em um processo $x : T \bullet P$, x é um parâmetro que pode ser usado na especificação de P como um valor do tipo T. Para usarmnos esse processo, precisamos instanciá-lo, fornecendo um valor para x, como em $(x : T \bullet P)(v)$, onde v é um valor do tipo T.

6 Estratégia de Refinamento

Em *Circus*, a unidade básica de refinamento é definida sobre uma ação, ou seja, o processo é refinado com base no refinamento de suas ações principais. A semântica de uma ação pode ser definida através de um schema que especifica a mudança no estado formado pelas componentes de estado do processo onde a ação se aplica e pelas variáves de observação adicionais *okay*, *wait*, *tr* e *ref*, cujo significado semântico encontra-se em [2]. Juntas, estas variáveis compôem o espaço de estados de uma ação.

Definição 6.1 (Refinamento de ação).

Para as ações A_1 e A_2 aplicadas sobre um mesmo espaço de estados, definimos $A_1 \sqsubseteq_{\mathcal{A}} A_2$, se, e somente se, $[A_2 \Rightarrow A_1]$.

A etapa de refinamento deve considerar tanto o estado como o comportamento do processo em questão. Como a ação principal define seu comportamento, um processo P_1 é refinado pelo processo P_2 se a ação principal de P_1 é refinada pela principal de P_2. Essas ações, entretanto, podem ser aplicadas em espaços de estados diferentes, não podendo ser comparadas entre si. Como o estado de um processo é um elemento interno deste, não sendo visível para outro processo, as ações serão comparadas abstraindo-se as componentes de estado de P_1 e P_2, como se elas tivessem sido declaradas em um bloco de variáveis local. O refinamento de processos ($A_1 \sqsubseteq_P A_2$) é, por isso, definido em termos de refinamento de ação aplicada a blocos locais de estado, com a semântica representada através do operador de quantificação existencial. Na definição a seguir, usamos $P.st$ e $P.act$ para denotar o estado local e a ação principal de um processo P.

Definição 6.2 (Refinamento de processo).

Definimos $P_1 \sqsubseteq_{\mathcal{P}} P_2$, se , e somente, se, $(\exists\, P_1.st;\ P_1.st' \bullet P_1.act) \sqsubseteq_{\mathcal{A}} (\exists\, P_2.st;\ P_2.st' \bullet P_2.act)$

Quando escondemos os estados locais dos processos, ambas as ações principais passam a ficar restritas sobre um mesmo espaço de estados, contendo apenas as variáveis *okay*, *wait*, *tr* e *ref*.

Em *Circus*, uma *simulação* pode ser definida como uma relação entre estados de dois processos distintos, satisfazendo algumas propriedades. Na realidade, além dos estados dos processos, esta relação também envolve um estado local que inclui variáveis locais no escopo, de entrada, e qualquer outra informação adicional inferida dos "guardas" e das condicionais no contexto.

Definição 6.3 (Simulação para frente).

Uma simulação para frente entre as ações A_1 e A_2 dos processos P_1 e P_2, respectivamente, com estado local L, é uma relação R entre $P_1.st$, $P_2.st$ e L, satisfazendo as seguintes propriedades:

1. (viabilidade) $\forall\, P_2.st;\ L \bullet (\forall\, P_1.st \bullet R)$
2. (corretude) $\forall\, P_1.st;\ P_2.st;\ P_2.st';\ L \bullet R \wedge A_2 \Rightarrow (\exists\, P_1.st';\ L' \bullet R' \wedge A_1)$

Neste caso, escrevemos $A_1 \preceq_{P_1,P_2,R,L} A_2$ e dizemos que a ação A_2 simula a ação A_1, de acordo com a relação R, em um estado extendido por L. Uma simulação para frente entre P_1 e P_2 é uma simulação para frente entre suas ações principais. O teorema reproduzido abaixo assegura que, dada uma simulação entre os processos P_1 e P_2, podemos substituir P_2 por P_1 em um programa.

Teorema 6.4 (Simulação para frente é consistente).

Quando uma simulação para frente existe entre dois processos P_1 e P_2, afirmamos que $P_1 \sqsubseteq_{\mathcal{P}} P_2$. Para a simulação para trás, as definições e os resultados são semelhantes.

Definição 6.4 (Simulação para trás).

Uma simulação para trás entre as ações A_1 e A_2 dos processos P_1 e P_2, respectivamente, com estado local L, é uma relação R entre $P_1.st$, $P_2.st$ e L, satisfazendo as seguintes propriedades:

1. (viabilidade) $\forall P_2.st;\ L \bullet (\forall P_1.st \bullet R)$
2. (corretude) $\forall P_1.st';\ P_2.st;\ P_2.st';\ L' \bullet R' \wedge A_2 \Rightarrow (\exists P_1.st;\ L \bullet R \wedge A_1)$

As noções de refinamento suportam a transformação de processos *Circus* para diversos propósitos, como por exemplo, mostrar a equivalência ou refinamento entre processos, realizar otimizações, ou justificar uma mudança na representação dos dados. Aqui, o interesse é no desenvolvimento de uma estratégia que será discutida a seguir. Ela se baseia nas leis de simulação e refinamentos de ação e processo, apresentados anteriormente e é aplicada nos seguintes passos:

- uso de simulação para introduzir os elementos de estado do sistema concreto.
- refinamento algorítmico das ações para particionamento do espaço de estado e ações do processo.
- refinamento de processo (decomposição).

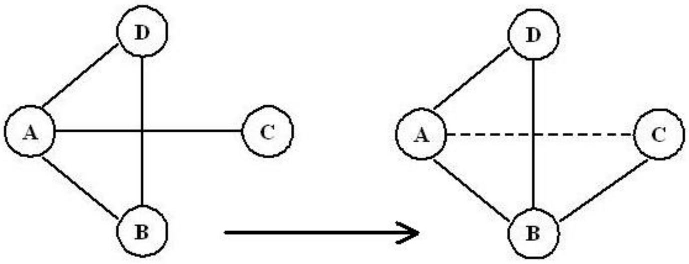
Embora simples, a aplicação iterativa destes passos serve como uma ferramenta para guiar e transformar uma especificação abstrata e centralizada em uma solução concreta e distribuída do sistema. Assumindo uma especificação centralizada como ponto de partida para o desenvolvimento, as duas primeiras etapas são usadas para aprimorar a estrutura interna do processo. Vale a pena enfatizar que muitas iterações possam ser necessárias, mesmo em um desenvolvimento de médio porte, desde que o processo resultante de uma iteração possa ser decomposto em outros processos, e assim por diante.

7 Mobilidade e Linguagem π-Calculus

Como normalmente acontece na área de ciência da computação, o termo *mobilidade* é usado com diferentes significados, geralmente, inconsistentes, o que torna apropriado a formulação de uma teoria matemática que encontre uma definição precisa e consistente. Os vários significados diferem pelo menos em dois aspectos: *que tipo de entidade se move, e em qual espaço isso acontece?* Existem algumas escolhas naturais, como por exemplo:

(A) processos movem-se, no espaço físico das unidades computacionais.
(B) processos movem-se, no espaço virtual das ligaçõs entre os processos.
(C) ligações movem-se, no espaço virtual das ligaçõs entre os processos.

π-Calculus[13] é um modelo que extende a teoria de CCS[6] no sentido de permitir a mobilidade dos processos, tendo a opção (C) escolhida como forma de representar esta característica. Dessa forma, o movimento de um processo pode ser representado inteiramente pelo movimento de suas ligações (canais). A opção (C) é adotada pela economia, flexibilidade e facilidade da representação da abstração de mobilidade. Um exemplo bastante simples é apresentado no *diagrama de fluxo* a seguir, onde A, B, C e D são processos, representados por círculos, que interagem entre si através dos canais de comunicação, representados pelas linhas. Veja que o processo C move-se de A para B, representado abaixo pela alteração nas ligações correspondentes.



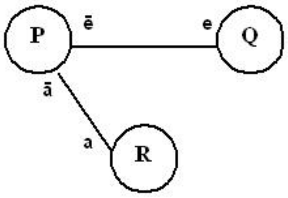
Suponha que um processo P queira enviar o valor 5 para o processo Q através do canal de comunicação a. Em π-Calculus, os processos são definidos como

$P \equiv \overline{a}5.P'$ e $R \equiv a(x).R'$

e a composição paralela é representada por

$(\overline{a}5.P' \mid a(x).R) \setminus a$

onde $\setminus a$ corresponde ao operador de *restrição* que torna o canal a privado a P e Q. Agora, suponha que o processo P queira delegar a Q a tarefa de enviar o valor 5 para R, sendo que inicialmente P comunica-se com Q através do canal e.

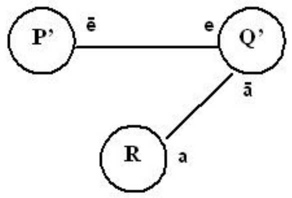


Vamos agora redefinir os processos de forma que $P \equiv \overline{e}a.\overline{e}5.P'$ e $Q \equiv e(y).e(z).\overline{y}z.0$. De acordo com a nova definição, P transmite pelo canal e o canal a e o valor 5; Q recebe um canal e um valor, transmitindo este valor pelo canal recebido, terminando em seguida. O sistema completo é apresentado abaixo.

$(\overline{e}a.\overline{e}5.P' \mid e(y).e(z).\overline{y}z.0 \mid a(x).R') \setminus a \setminus e$

Após duas comunicações, ambas pelo canal e, o sistema torna-se
$(P' \mid \overline{a}5.0 \mid a(x).R') \setminus a \setminus e$

O novo diagrama de fluxo é apresentado a seguir, indicando que o canal $\overline{a}$ de P mudou para Q, e Q tornou-se $Q' \equiv \overline{a}5.0$.



8 Conclusão

Ao longo do trabalho desenvolvido, algumas teorias formais foram estudadas com o intuito de conhecer os diferentes paradigmas de cada uma. Vimos que Z é uma teoria para especificação de software que procura descrever o comportamento interno em termos de relações sobre estruturas de dados complexas que representam o estado do sistema. Já em CSP, o importante é a descrição do comportamento externo através das interações concorrentes com outros sistemas. *Circus*, em contrapartida, é uma teoria completa que unifica estes paradigmas, possuindo uma teoria de refinamentos associada, permitindo o desenvolvimento de software da especificação até uma implementação executável. O conceito de mobilidade ainda precisa ser desenvolvido em *Circus* para torná-la ainda mais completa. Estudando-se π-Calculus, que é a linguagem que melhor caracteriza mobilidade pode nos ajudar a entender de que forma *Circus* poderia ser extendido. Um trabalho neste sentido já está sendo desenvolvido por [15] e entender a nova abordagem será o objetivo dos próximos estudos.

References

[1] C.A.R. Hoare. *Communicating Sequencial Processes*. Prentice-Hall International, 1985.

[2] C.A.R. Hoare e He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998

[3] C.C. Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[4] J.C.P. Woodcock e J. Davies. *Using Z - Specification, Refinement and Proof*. Prentice-Hall, 1995.

[5] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[6] R. Milner. *Communication and Concurrency*, Prentice-Hall, 1989.

[7] C. Fischer. *CSP-OZ: A combination of Object-Z and CSP*. FMOODS97, volume 2, páginas 423 - 438. 1997.

[8] K. Taguchi e K. Araki. *The State-based CCS Semantics for Concurrent Z Specification*. International Conference on Formal Engineering Methods, páginas 283-292, 1997.

[9] J.C.P. Woodcock e A.L.C. Cavalcanti. *Circus: a concurrent refinement language*. Technical Report, Oxford University.

[10] J.C.P. Woodcock e A.L.C. Cavalcanti. *A Refinement Strategy for Circus*. Formal Aspects of Computing.

[11] A.W Roscoe, J.C.P. Woodcock e L. Wulf. *Non interference through Determinism*. ESORICS 94, volume 1214 do Lecture Notes in Computer Science, páginas 33-54, Springer-Verlag, 1994.

[12] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[13] R. Milner, J. Parrow e D. Walker. *A Calculus of Mobile Processes, Parts I and II.* Information and Computation, 1992.

[14] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[15] X.Tang. *Mobile Processes in Unifying Theories.* Technical Report. Kent University, 2004.