

Um conversor da notação CSP_Z para CSP_M

ADALBERTO FARIAS¹
ALEXANDRE MOTA¹
AUGUSTO SAMPAIO¹

¹UFPE – Universidade Federal de Pernambuco
CIn – Centro de Informática
Cx. Postal 7851 – CEP 50.740-540 Recife (PE)
{acf,acm,acas}@cin.ufpe.br

Resumo: A Engenharia de Software faz uso de técnicas, métodos e ferramentas para descrever sistemas em uma linguagem de alto nível, permitindo provar propriedades e até definir a qualidade do sistema antes mesmo de sua implementação. Métodos Formais dão suporte ao desenvolvimento de software através de suas técnicas matemáticas, garantindo corretude, completude e eliminando a ambigüidade comumente encontrada no desenvolvimento *ad hoc*. Uma técnica de Métodos Formais muito usada na indústria é o *model checking*. Ferramentas de apoio automatizam a verificação de especificações formais baseando-se nessa técnica. A integração de formalismos também é uma outra técnica importante, que resulta em uma linguagem formal mais expressiva, caracterizada pelo reuso quase que total das sintaxes e semânticas das notações integrantes. O presente trabalho procura descrever uma ferramenta, implementada em Java, cujo propósito é transformar especificações escritas em CSP_Z, integração formal entre CSP e Z, para CSP_M, versão de CSP usada para realizar *model checking*.

Palavras Chave: CSP_Z, CSP_M, FDR, especificação formal, refinamento, *model checking*, Java.

1 Introdução

Os esforços em produzir ferramentas de suporte às diferentes fases do processo de desenvolvimento de software têm aumentado bastante. A abordagem formal, através de técnicas matemáticas consolidadas, apresenta-se como uma forte alternativa para a diminuição de erros e aumento da abstração em todas as fases do ciclo de vida do software. Diversas notações procuram dar suporte à descrição de sistemas em seus diferentes aspectos: operações, estruturas de dados, espaço de estados, interações entre processos etc.

Com o uso de ferramentas de suporte¹, sistemas podem ser analisados, verificados, depurados e até mesmo simulados antes de serem desenvolvidos, a partir de suas especificações, escritas numa linguagem mais abstrata e fortemente baseada em leis matemáticas.

Este trabalho está organizado como segue. A Seção 2 dá uma visão geral da técnica de *model checking* e da ferramenta FDR [FDR, 1997] (*model checker*). A Seção 3 descreve a sintaxe de CSP_Z [Mota, 1997] através de um exemplo, comparando-o com suas versões em CSP [Hoare, 1985] [Roscoe, 1998] e CSP_M [FDR, 1997]. Na Seção 4 encontra-se a estratégia de conversão implementada na ferramenta. Na Seção 5 o conversor é mostrado em detalhes e a Seção 6 contém nossas conclusões, bem como nossos futuros interesses na presente área.

Para um melhor entendimento do artigo, é desejável que o leitor conheça os principais termos de CSP e Z [Spivey, 1992].

¹ Por exemplo: com o z-eyes pode-se provar refinamento entre especificações escritas em Z. O FDR serve para verificar ausência de deadlock em especificações escritas em CSP_M. O Zans funciona como uma ferramenta de animação/simulação de operações escritas em Z.

2 Visão geral de *model checking* e FDR

Model checking é um método para verificar formalmente sistemas concorrentes de estado finito e provar certas propriedades desejáveis. A técnica consiste em especificar sistemas utilizando autômatos ou fórmulas da lógica temporal, onde os mesmos possam ser vistos como sistemas de transição de estados [Biere, 1999]. Representados por um complexo diagrama de estados, os sistemas podem ser analisados por algoritmos que percorrem o grafo no sentido de encontrar certas propriedades. Desta forma, *model checking* tem sido muito útil para encontrar erros sutis em especificações complexas e não triviais.

Sistemas modelados em CSP_M podem ser analisados pelo FDR, um *model checker* capaz de provar propriedades como determinismo, ausência de *deadlock/livelock* e refinamento de processos.

3 Sintaxe de CSP_Z

A linguagem CSP_Z é uma integração semântica de CSP e Z, de modo que CSP manipula os aspectos de concorrência do sistema, e Z trata das estruturas de dados [Mota, 2001]. A sintaxe desta notação é mostrada através de um exemplo.

3.1 Um exemplo simples

O exemplo a seguir é um relógio com dois eventos: *ticktack* e *bird*. O primeiro deles acontece a cada minuto e quando o valor de um determinado contador atingir 60, é restaurado novamente para 0 e o evento *bird!msg* acontece, onde *msg* é um dado (*Cookoo*) que trafega pelo canal *bird*. A especificação em CSP é mostrada abaixo:

$$\begin{aligned} \text{Counter}(c) &= P(c) \star c < 60 \star Q \\ P(c) &= \text{ticktack} \rightarrow \text{Counter}(c + 1) \\ Q &= \text{bird!msg} \rightarrow \text{Counter}(0) \end{aligned}$$

No trecho acima, *Counter* é um processo parametrizado que avalia um contador ($c < 60$). Caso o resultado seja verdadeiro, então *Counter* passa a se comportar como *P*, senão passa a se comportar como *Q*. Por sua vez, o processo *P* oferece o evento *ticktack*, incrementa o valor do contador e passa a se comportar como *Counter*. De forma similar, o processo *Q* oferece o evento *bird!msg*, colocando um dado no canal *bird*, restaura o valor do contador e passa a se comportar como *Counter*.

Poderíamos observar a especificação em CSP como se fosse uma máquina de estados.

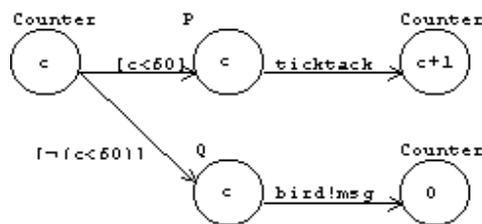


Figura 1: Visão de máquina de estados

Os círculos representam o estado do sistema com apenas um componente, o contador *c*. O rótulo acima de cada estado indica o processo executado. As transições rotuladas com uma condição entre '[' e ']' indicam o caminho seguido quando da validade da condição. As transições rotuladas com eventos indicam o acontecimento dos mesmos e o próximo comportamento do sistema (ver Figura 1).

O código correspondente em CSP_M é mostrado abaixo:

```
datatype SOUND = Cookoo
channel ticktack
channel bird:SOUND

Counter(c) = if c < 60 then P(c) else Q
P(c) = ticktack -> Counter(c + 1)
Q = bird?msg : {x | x <- SOUND} -> bird!msg -> Counter(0)
```

Nas versões em CSP e CSP_M mostradas acima, o processo *Counter* lida diretamente com as estruturas de dados, analisando e modificando valores. Para sistemas de grande porte, cujo estado geralmente é composto por estruturas complexas, estas notações não se apresentam como uma boa opção para a escrita de especificações. Por serem álgebras de processos, possuem maior expressividade para representar interação entre os mesmos. Ao contrário disso, a linguagem Z apresenta-se mais adequada para o tratamento de dados, propriedades e operações do sistema, embora não possua termos que permitam descrever aspectos de concorrência ou ordem entre operações.

3.2 Usando CSP_Z para especificar sistemas

Especificar sistemas concorrentes não significa apenas descrever os aspectos de sincronização. É preciso tratar também as estruturas de dados. Valendo-se da natureza complementar entre CSP e Z, CSP_Z procura integrá-las preservando as características semânticas de cada uma. As palavras reservadas **spec/end_spec** limitam o escopo da especificação que é dividida em duas partes distintas: uma de CSP e outra de Z.

3.2.1 Tipos globais

Em CSP_Z, tipos utilizados na parte de CSP e Z são declarados globais, ou seja, fora do bloco **spec/end_spec**. Nos trechos apresentados em 3.1, *ticktack* é um canal pelo qual não trafegam dados, ao contrário de *bird*, pelo qual trafega um tipo de dado ao qual *msg* pertence. A definição do tipo de dado que trafega pelo canal *bird* fica:

```
SOUND == Cookoo
```

3.2.2 Parte de CSP

Esta parte tem por objetivo descrever os aspectos de concorrência do sistema: canais, eventos, paralelismo, processos etc. O seguinte trecho mostra a declaração dos canais em CSP_Z:

```
channel ticktack:[]
channel bird: [msg:SOUND]
```

Os canais de CSP_Z têm a mesma funcionalidade dos canais em CSP_M². Surge apenas uma pequena mudança na declaração do tipo: em CSP_Z o tipo é denotado por um registro, ao contrário de CSP_M onde o tipo é declarado explicitamente. Declarações de canais locais³ também são permitidas com a seguinte sintaxe:

```
local_channel my_channel:[]
```

Em seguida, define-se um processo chamado *main* cujo propósito é reunir os demais processos (ou eventos) da parte de CSP. No exemplo proposto, o relógio oferece os dois eventos *e*, em cada vez que cada um deles acontece, *ticktack* e *bird* voltam a ser oferecidos novamente. O processo principal de CSP pode ser escrito da forma:

² A Tabela 1 mostra a correspondência entre os principais termos de CSP e CSP_M.

³ Canais locais são canais cujos eventos não acontecem fora do escopo **spec/end_spec**, sendo escondidos ao final da especificação.

```
main = (ticktack -> main) [] (bird!msg -> main)
```

O comando condicional foi eliminado e o não-determinismo foi inserido no processo. Isto será resolvido pela parte de Z.

A construção do processo `main` pode ser feita também a partir de processos auxiliares. Por exemplo:

```
main = d -> e -> main [] Q
Q = A [] {b} [] B
A = a -> b -> A
B = c -> b -> B
```

O processo `Q` participa diretamente da definição de `main`, enquanto `A` e `B` participam indiretamente.

CSP	CSP _M	Descrição
$stop$	STOP	Deadlock
$skip$	SKIP	Successful termination
$a \rightarrow P$	$a \rightarrow P$	Simple prefix
$a?x \rightarrow P$	$a?x \rightarrow P$	Input prefix
$a!v \rightarrow P$	$a!v \rightarrow P$	Output prefix
$a?x?y:A!v \rightarrow P$	$a?x?y:A!v \rightarrow P$	Complex (mix) prefix
$P \square Q$	$P [] Q$	External choice
$P \sqcap Q$	$P \sim Q$	Internal choice
$P \setminus A$	$P \setminus A$	Hiding
$P \stackrel{b}{\leftarrow} b \stackrel{b}{\rightarrow} Q$	if b then P else Q	Conditional choice
$P \stackrel{b}{\leftarrow} b \stackrel{b}{\rightarrow} stop$	$b \ \& \ P$	Boolean guard
$P [X] Q$	$P [[X]] Q$	Parallel composition
$P Q$	$P Q$	Interleaving
$P \gg Q$	$P [c \leftarrow c'] Q$	Piping
$P(s)$	$P(s)$	Parameterisation
$P(f(s))$	let $s' = f(s)$ within $P(s')$	Local declaration
$P \stackrel{\circ}{\circ} Q$	$P ; Q$	Sequential composition
$\square_{i \in T} P_i$	$[[i:T @ P(i)]$	Process indexing ⁴
P_i	$P(i)$	Parameterisation

Tabela 1: Mapeamento entre CSP e CSP_M

3.2.3 Parte de Z

A parte de Z tem por objetivo definir as seguintes estruturas:

- Tipos e valores manipulados apenas pela parte de Z;

⁴ Também aplicável aos operadores \square , $[X]$, $||$, \gg , $\stackrel{\circ}{\circ}$ e $|||$.

- O espaço de estados do sistema – definição do estado do sistema, envolvendo as propriedades e o invariante⁵;
- O esquema de inicialização – ponto inicial para o funcionamento do sistema. Nesse esquema, todos os componentes são inicializados;
- Esquemas de operações – englobam todas as operações, não interessando o fato de elas mudarem ou não o estado do sistema. Quando uma operação em Z corresponder a um evento em CSP, o nome do esquema deve ser composto de “com_” + “nomeDoCanal”. Isso indica que, se um evento no canal x corresponde a uma operação do sistema, haverá um esquema correspondente em Z cujo nome será com_x . Se o canal for tipado, variáveis de entrada/saída nos esquemas representarão os tipos. Os nomes destas variáveis devem coincidir com a variável utilizada no registro denotando o tipo do canal ($msg!$ em com_bird coincide com msg em $[msg:SOUND]$). Esta coincidência é essencial para sincronizar as partes de CSP e Z nos mesmos eventos.

A parte de Z fica então:

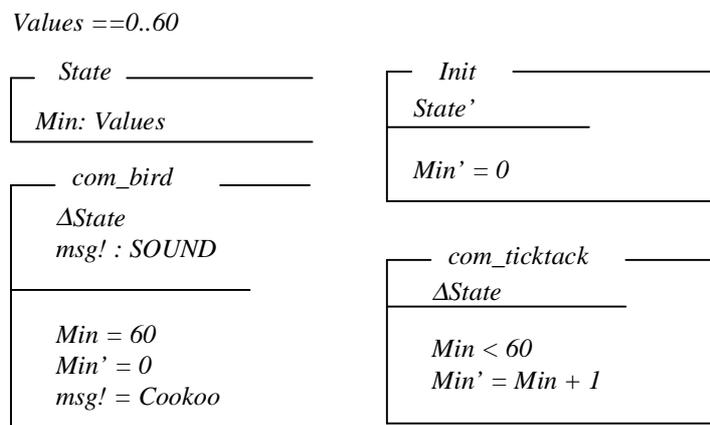


Figura 2: Parte de Z

Cada esquema em Z possui pré e pós-condições. Se num determinado instante houver mais de uma pré-condição verdadeira, os respectivos esquemas tornam-se disponíveis para execução. A Figura 3 mostra uma visão de máquina de estados para um esquema em CSP_Z.

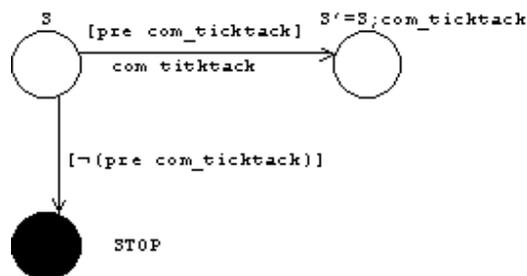


Figura 3: Execução de um esquema de Z

⁵ O invariante de um sistema especificado em Z é um predicado que deve ser obedecido a cada operação realizada pelo sistema, preservando as restrições e requisitos do mesmo.

O sistema encontra-se num estado S , oferecendo a operação $com_ticktack$. Caso sua pré-condição seja verdadeira, então o esquema é executado e o novo estado S' é definido a partir do estado anterior, modificado (possivelmente) pela execução de $com_ticktack$. Quando da falsidade da pré-condição, o sistema vai para um estado de *deadlock*.

É importante que a ausência de *deadlock* e o determinismo sejam garantidos na parte de Z . Para isso, os seguintes pontos devem ser considerados:

- Em determinado estado, o sistema oferece um certo número de operações. A condição necessária e suficiente para garantir o determinismo do sistema nesse estado pode ser escrita da seguinte forma:

$$\forall i, j \mid i \neq j. pre\ com_op_i \wedge pre\ com_op_j = false, \quad (1)$$

onde com_op_k é qualquer operação oferecida pelo sistema. Assim, em qualquer estado, apenas uma operação poderá ser realizada.

- Para garantir a ausência de *deadlock*, as pré-condições das operações, em cada estado do sistema, devem satisfazer à condição:

$$\forall_i pre\ com_op_i = true, \quad (2)$$

onde \vee representa a operação “ou” *booleano*. Com isso, é garantido que, em cada estado, pelo menos uma operação será realizada.

Analisando a Figura 4 e procurando satisfazer as condições 1 e 2, temos os seguintes resultados.

$$(1) pre\ com_ticktack \wedge pre\ com_bird$$

$$= Min < 60 \wedge Min = 60$$

$$= false$$

$$(2) pre\ com_ticktack \vee pre\ com_bird = true$$

$$= Min < 60 \vee Min = 60$$

$$= true, \text{ considerando que } Min \text{ assume valores definidos em } Values.$$

Assim, nosso exemplo é determinístico e livre de *deadlock*.

Uma representação mais completa do sistema oferecendo as duas operações é mostrada na Figura 4. A possibilidade de *deadlock* foi retirada devido à satisfatibilidade das condições (1) e (2).

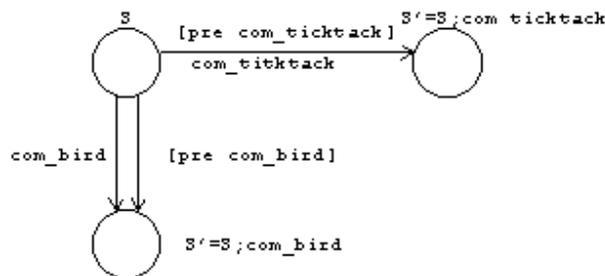


Figura 4: Representação com todos os esquemas de operações

Unindo agora as partes de CSP e Z, temos a especificação completa:

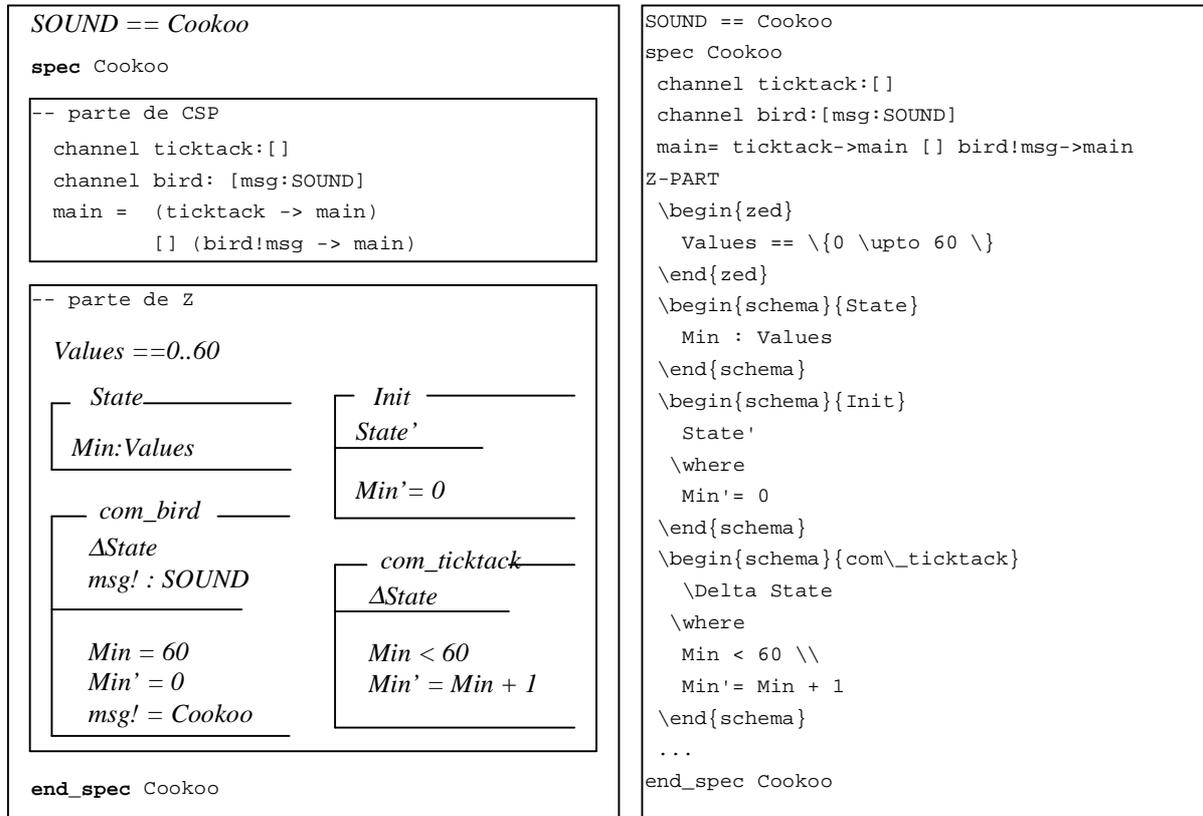


Figura 5: Especificação em CSPz: representação gráfica e textual⁶

4 Estratégia de conversão para CSP_M

Esta seção apresenta a estratégia de conversão implementada na ferramenta, cujo objetivo é gerar uma especificação em CSP_M para posterior análise em FDR. Um estudo de caso interessante do uso de CSPz foi a especificação formal de um subconjunto do computador de bordo do primeiro Satélite de Aplicações Científicas do Instituto Nacional de Pesquisas Espaciais (INPE) [Mota, 1997]. A conversão detalhada para CSP_M encontra-se em [Mota, 2001].

A idéia principal da estratégia de conversão é transformar declarações de Z em estruturas de dados e processos de CSP, depois sincronizá-los de forma que os processos escritos em Z passem a ativar eventos de CSP. Resumidamente, os detalhes da conversão são os seguintes:

- Processamento das definições dos tipos de dados globais da especificação;
- Todas as operações sobre estruturas de dados ficam restritas a esquemas de Z. Nenhuma alteração nos dados é feita na parte de CSP da especificação (ver Figura 5);
- Os processos de CSP são disponibilizados para sincronização através do processo `main` (ver Figura 5);
- Todos os esquemas de Z são transformados em funções de CSP, incorporando pré e pós-condições;
- Um outro processo, nomeado `Z_CSP`, reúne os processos originados dos esquemas de Z e leva em consideração detalhes de inicialização do sistema;

⁶ A parte de Z é editada em LaTeX [Lampart, 1998].

- Finalmente, todos os processos são declarados locais, fazendo parte apenas do escopo da sincronização entre os processos `main` e `Z_CSP`. Canais locais são escondidos ao final da sincronização.

Aplicando os passos da estratégia ao exemplo dado na Figura 5, o conversor produz uma especificação em CSP_M , semanticamente equivalente à especificação original. Cada passo é mostrado a seguir.

Primeiramente ocorre o processamento dos tipos de dados. Neste caso, $SOUND == Cookoo$ é convertido da seguinte forma:

```
datatype SOUND = Cookoo
```

A conversão dos canais *ticktack* e *bird* origina o seguinte código:

```
channel ticktack
channel bird : SOUND
```

O processo `main` sofre uma pequena alteração: eventos colocando valores de saída nos canais são substituídos por eventos de leitura de valores nos mesmos canais. Uma vez que a parte de CSP agora é livre da manipulação de dados, a parte de Z se encarrega de colocar dados nos canais. Além do mais, a construção anterior (`bird!msg`) levaria a um erro de sintaxe na especificação.

```
main = (ticktack -> main) [] (bird?msg -> main)
```

Na conversão dos tipos de dados e valores utilizados na parte de Z temos:

```
Values = {0..60}
```

Para sincronizar a parte de CSP com a de Z, a seguinte interface de canais é gerada:

```
channels = {|ticktack,bird|}
lChannels = {}
Interface = union(channels,lChannels)
```

A união dos canais declarados (locais e não-locais) serve de interface de sincronização. Canais locais são escondidos ao final da especificação.

Os esquemas representando o estado, inicialização e operações são transformados nas seguintes estruturas:

```
State = {Min | Min <- Values}
Init = {Min' | Min' <- Values, Min'==0}
com(Min,ticktack) = {Min' | Min' <- State, Min < 60, Min' == Min + 1}
com(Min,bird.msg)={Min' | Min' <- State, Min == 60, Min' == 0, msg == Cookoo}
```

O estado do sistema transformou-se num conjunto de valores que representam os componentes do sistema, ou melhor, o estado do sistema em um determinado tempo pode ser representado por uma tupla. No caso do exemplo, apenas um componente foi representado (`Min`). A inicialização simplesmente atribui um valor definido aos componentes do estado. As operações (esquemas) foram implementadas como funções partindo de pares (*estado, comunicação*) para conjunto de tuplas, cujos valores são definidos pelas pós-condições dos esquemas. As pré-condições incorporadas servem para habilitar a operação.

Ao reunir as funções geradas pelos esquemas de Z , considerando os aspectos de inicialização, constrói-se o processo Z_CSP da seguinte forma:

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

No trecho acima, $Z(State)$ é uma escolha externa de tuplas do tipo $(States, Comm)$, onde $States$ representa o espaço de estados e $Comm$, um evento de sincronização que pode acontecer. Em resumo, $Z(State)$ executa algum dos padrões " $com(\dots)$ ", que sincroniza com a parte de CSP e depois volta a oferecer todos os padrões novamente. Em particular, a função $com(Min, bird.msg)$ gera o evento $bird.msg$ que sincroniza com $bird!msg$ da parte de CSP.

Uma vez combinando as funções de Z num único processo, a definição de $Z(State)$ não precisa existir fora do escopo de Z_CSP . Assim, uma declaração local de CSP_M reúne a funcionalidade dos esquemas considerando aspectos de inicialização.

Finalmente, a especificação é transformada numa declaração local e as partes de CSP e Z são sincronizadas nos eventos ocorrendo em $Interface$:

```
(Tipos de dados e canais)
Cookoo =
  let
    ... (Definições de CSP e Z)
  within (main [|Interface|] Z_CSP)\lChannels
```

Agrupando os trechos apresentados num só bloco, pode-se visualizar a especificação em CSP_M , gerada pela ferramenta. Linhas iniciadas com "--" representam comentários.

```
-- Datatypes and channels
datatype SOUND = Cookoo
channel ticktack
channel bird : SOUND
Cookoo =
  let
    -- The Interface
    channels = {|ticktack,bird|}
    lChannels = {}
    Interface = union(channels,lChannels)

    -- The CSP part
    main = (ticktack -> main) [] (bird?msg -> main)

    -- The Z Part
    Values = {0..60}
    State = {Min | Min <- Values}
    Init = {Min' | Min' <- Values, Min'==0}

    com(Min,ticktack) = { Min' | Min' <- State, Min < 60, Min' == Min + 1}
    com(Min,bird.msg)={ Min' | Min' <- State, Min == 60, Min' == 0, msg == Cookoo}
```

```

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

```

5 O Conversor

Esta seção apresenta a ferramenta, procurando justificar a escolha de Java [Horstman, 2000] como linguagem de implementação e aborda outras técnicas de suporte ao *model checking* de sistemas concorrentes, sob o ponto de vista de integração de notações para especificação. Ao final encontram-se as principais funcionalidades e telas do conversor, bem como suas limitações.

5.1 Utilização de Java

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

- Independência de plataforma e portabilidade da linguagem.
- Simplicidade nos requisitos para utilização da ferramenta. Apenas uma Máquina Virtual Java versão 1.2.x (ou superior) é suficiente;
- A linguagem possui diversas APIs⁷ provendo suporte à natureza da aplicação: leitura de arquivos, interfaces gráficas etc;
- Auxílio do *JLex* e *JavaCUP*⁸, APIs para construção de analisadores léxicos e sintáticos, respectivamente;
- Reuso do analisador sintático de *Z*, implementado anteriormente no Centro de Informática da UFPE, utilizando também *JLex* e *JavaCUP*.

5.2 Trabalhos relacionados

Esta seção apenas cita trabalhos que possuem o mesmo propósito do nosso conversor, dar suporte à verificação de sistemas concorrentes.

A linguagem *RAISE Specification Language (RSL)* [Smith, 1997], desenvolvida para uso em sistemas complexos de escala industrial envolvendo concorrência, reúne feições de VDM⁹ e CSP, entretanto não foi encontrada uma estratégia de conversão para uma notação que torne possível sua verificação, nem ferramenta que realize o *model checking* em especificações escritas nessa linguagem.

A iniciativa de padronização da ODP (*Open Distributed Processing*) adotou uma abordagem para especificação de sistemas distribuídos utilizando *Z* e LOTOS¹⁰, onde cada uma delas trata de seu domínio de aplicação: *Z* para estruturas de dados complexas e LOTOS para interações entre processos.

Uma outra alternativa é integrar uma linguagem variante de *Z*, *Object-Z (OZ)*, com CSP [Fischer, 1997]. Utilizando *OZ*, é possível especificar sistemas como se fossem objetos: propriedades são atributos, inicializador representa o construtor e operações são vistas como métodos. Nessa integração, CSP é utilizada para descrever interações entre processos. CSP-OZ serve então como um *framework* para especificações de sistemas distribuídos baseados em objetos. Existe uma proposta de ferramenta para *model checking* de CSP-OZ, porém não implementada.

⁷ *Application Programming Interfaces* – pacotes provendo algum tipo de suporte extra para desenvolvedores.

⁸ Descritos detalhadamente em: <http://www.ualberta.ca/~maldrigg/tutor/JavaTut.html> e <http://www.cs.princeton.edu/~appel/modern/java/CUP>, respectivamente.

⁹ *Vienna Development Method* – método formal orientado a modelo, baseado em uma semântica denotacional e suporta refinamento passo-a-passo de especificações abstratas em especificações concretas. VDM-SL é a linguagem de especificação formal desse método e suporta várias formas de abstração.

¹⁰ Inclui uma parte de descrição de processos baseada em CCS (*Calculus of Communicating Systems*), um método de álgebra de processos com o qual é possível descrever aspectos de concorrência entre os mesmos.

Para CSP_Z, o conversor apresentado neste trabalho é uma tentativa pioneira de se produzir uma ferramenta transformacional¹¹, capaz de gerar uma especificação alvo em CSP_M, baseando-se numa técnica bem fundamentada descrita detalhadamente em [Mota, 2001].

5.3 Interface gráfica

A seguir, as principais telas e funcionalidades do conversor são mostradas. As telas foram implementadas utilizando-se componentes *swing* [Geary, 1999] ao invés de componentes AWT (*Advanced Window Toolkit*), devido a uma maior estabilidade de comportamento entre diferentes plataformas.

5.3.1 Tela principal

Principal tela de interação com o usuário (ver Figura 6). Possui os seguintes componentes:

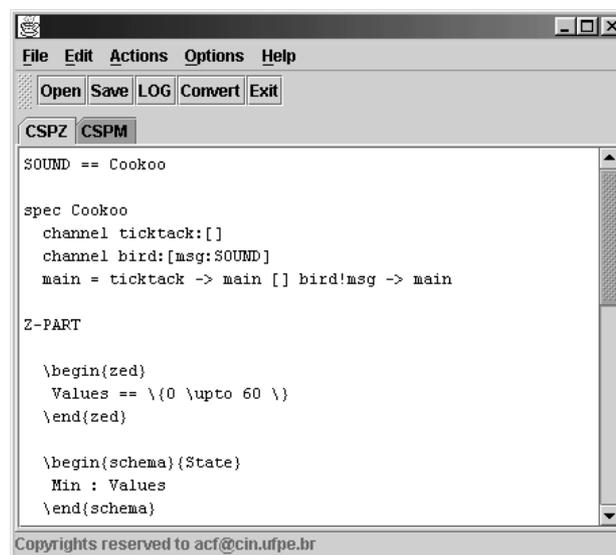


Figura 6: Tela principal

- Barra de Menu – apresenta as diversas opções de interação com o usuário por meio de menus;
- Botões de Ação – ativam as principais funcionalidades do conversor sem precisar acessar menus;
- Painel de Edição – fornece suporte para edição de especificações em CSP_Z.

5.3.2 Barra de menus

Contém as seguintes opções:

- File – composta pelos itens: *Open* (abre uma especificação em CSP_Z)¹², *Save* (salva a especificação mostrada no Painel de Edição), *Save as...* (permite mudar o nome da especificação a ser salva) e *Exit* (sai do programa);
- Edit – possui itens que atuam apenas na edição de texto: *Cut* (recorta o texto selecionado), *Copy* (copia o texto selecionado) e *Paste* (cola o conteúdo da área de transferência para a posição atual do cursor no Painel de Edição);

¹¹ Ferramentas capazes de gerar uma especificação alvo, a partir de uma especificação fonte, preservando o conteúdo semântico.

¹² A ferramenta não faz restrições quanto à extensão do arquivo, no entanto é recomendável que a extensão seja “.cspz”.

- Options – contém os itens: *Tab Placement* (modifica a posição das abas do Painel de Edição), *Convert* (converte a especificação de CSP_Z para CSP_M)¹³ e *Functions Mappings...* (permite criação/modificação do mapeamento de operações Z em funções CSP_M)¹⁴;
- Help – possui os itens: *Local Help* (exibe um arquivo de ajuda instalado juntamente com o conversor), *On Line Help* (acessa um arquivo de ajuda remoto), *About* (mostra informações sobre o produto).

5.3.3 Botões de ação

Permitem acesso às principais funções do conversor sem usar menus. Os seguintes botões podem ser utilizados:

- *Open* – funcionalidade semelhante ao item *Open* do menu *File* (5.3.2);
- *Save* – salva a especificação atualmente mostrada, tal como a opção *Save* do menu *File* (5.3.2);
- *LOG* - abre uma caixa de diálogo mostrando os erros ocorridos durante a análise sintática da especificação;
- *Convert* - gera a especificação em CSP_M, semelhante à opção *Convert* do menu *Options* (5.3.2);
- *Exit* – Sai do programa.

5.3.4 Painel de Edição

Fornece suporte à edição de especificações. Possui duas abas através das quais pode-se editar as especificações em CSP_Z e CSP_M. A Figura 7 mostra a tela principal com a aba “CSPM” do Painel de Edição ativada.

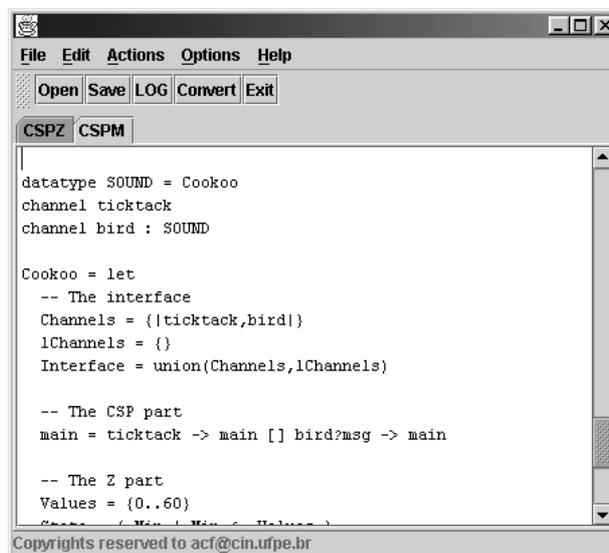


Figura 7: Especificação em CSP_M

¹³ Ao ser gerada, a especificação em CSP_M pode ser vista através da aba “CSPM” do Painel de Edição.

¹⁴ Maiores detalhes sobre o mapeamento das funções podem ser encontrados no item 5.3.6 deste trabalho.

5.3.5 Tela de erros

Mostra os erros ocorridos durante a análise sintática da especificação escrita em CSP_Z (ver Figura 8).

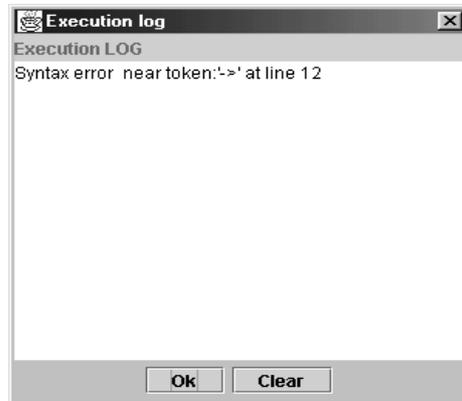


Figura 8: Tela de erros sintáticos

5.3.6 Tela de mapeamento de funções

Nem sempre é possível mapear todos os termos de Z em termos de CSP_M. Por exemplo, *Bag* é uma estrutura de dados disponível em Z que não possui correspondente em CSP_M. Operadores como \subset são implícitos em Z , enquanto em CSP_M existe apenas uma função, $member(x,A)$, que retorna verdadeiro caso x pertença ao conjunto A .

Nos casos em que o mapeamento direto não for possível (questões não-decidíveis¹⁵), a ferramenta gera um trecho limitado entre '\$', devendo o usuário modificá-lo manualmente.

Com o objetivo de fornecer um maior suporte ao usuário, o conversor possui funções de CSP_M que representam algumas operações e predicados de Z . A Figura 9 mostra esse mapeamento. Do lado esquerdo, encontram-se as principais operações/predicados de Z . Ao selecionar uma delas, o código correspondente em CSP_M é mostrado na caixa de texto à direita. O usuário tem a liberdade de modificar o corpo da função. Na conversão, caso alguma operação de Z tenha sido utilizada, por exemplo $C_1 \setminus \text{subseteq} C_2$, sua ocorrência será substituída pela função presente no mapeamento. Neste caso, $\text{subsetq}(C_1, C_2)$.

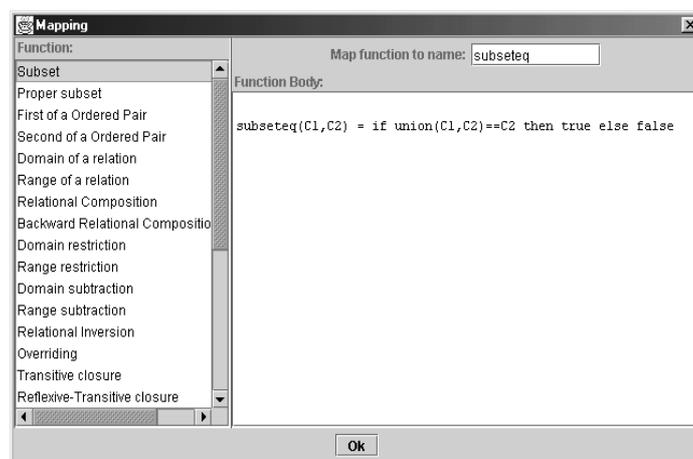


Figura 9: Tela de mapeamento de funções

¹⁵ Maiores detalhes sobre o que pode ser convertido são encontrados em [Borba, 1991].

5.3.7 Tela de ajuda

Exibe uma pequena ajuda sob forma de hipertexto, semelhante à visualização de documentos da Internet.

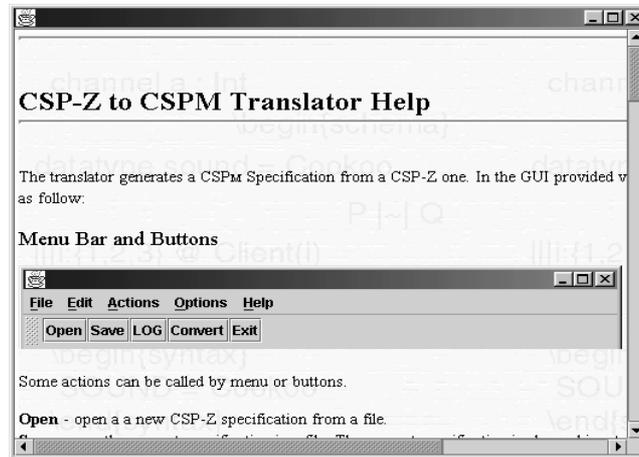


Figura 10: Tela de ajuda

6 Conclusão

Com o grande aumento da utilização de *model checking* no desenvolvimento de sistemas concorrentes surge a necessidade de notações e ferramentas mais poderosas para especificação e verificação.

A integração de notações existentes apresenta-se como uma alternativa promissora, ao invés de se criar novas linguagens suportando especificar tanto a parte de controle de processos quanto a parte de estruturas de dados. Algumas opções de integração entre diferentes linguagens foram mostradas no item 5.2 deste trabalho.

A ferramenta¹⁶ apresentada tem o propósito de implementar uma estratégia de conversão de uma notação integrada, CSP_Z , para uma notação verificável em FDR, CSP_M , sendo inclusive usada na disciplina Especificação de Sistemas Distribuídos, do curso de Mestrado do Centro de Informática da UFPE. O conversor está inserido no contexto do projeto ForMW do LMF (Laboratório de Métodos Formais), cujas informações podem ser obtidas em <http://www.cin.ufpe.br/~lmf>.

Esperamos que este trabalho contribua para a difusão de Métodos Formais na Engenharia de Software e estimule o uso de CSP_Z em especificações, tanto em nível acadêmico quanto em nível comercial, como também encoraje o desenvolvimento de ferramentas de apoio à técnica descrita. Nas próximas versões, planejamos prover suporte à detecção de *deadlock/livelock* local¹⁷, bem como tornar finitos processos CSP_Z infinitos¹⁸.

¹⁶ Uma versão encontra-se disponível na Internet em <http://www.cin.ufpe.br/~acf>.

¹⁷ Este suporte consiste em particionar especificações em CSP_Z em especificações mais simples preservando a semântica. A prova de propriedades sobre a especificação inicial pode ser feita em função das verificações realizadas sobre as especificações menores (composicionalidade). Os detalhes sobre verificações locais encontram-se em [Mota, 1997].

¹⁸ Processos infinitos podem ser transformados em finitos, verificáveis em FDR, sem alteração da semântica original. A fundamentação para este suporte encontra-se em [Mota, Tese em Andamento] [Lazić, 1999].

7 Referências

- [Biere, 1999] Biere, A. & Cimatti, A. & Clarke, E. & Zhu, Y. *Symbolic Model Checking without BDDs*, CMU-CS-99-101, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, 1999, pp. 01– 06.
- [Borba, 1991] Borba, P. & Meira, S. *From model based specifications to funcional prototypes*. IEEE TENCON'91. Session on Rapid Prototyping with Functional Programming Languages, 1991.
- [FDR, 1997] *Failures Divergences Refinement*. FDR2 User Manual, 1997.
- [Fischer, 1997] Fischer, C. *Combining object-Z and CSP*, 2nd International Conference on Formal Methods for Open Object-based Distributed Systems, Chapman & Hall, London, 1997.
- [Geary, 1999] Geary, D. *Graphic Java 2. Mastering the JFC*, Sun Microsystems Press, 1999.
- [Hoare, 1985] Hoare, C. *Communicating Sequential Processes*, Prentice Hall, Englewood Cliffs, NJ, 1985.
- [Horstman, 2000] Horstman, C. & Cornell, G. *Core Java 2*, Sun Microsystems Press, 2000. Vols I and II.
- [Lamport, 1998] Lamport, L. *LaTeX. A Document Preparation System. User's Guide and Reference Manual*, Addison-Wesley, 1998.
- [Lazić, 1999] Lazić, R. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis. Oxford University Computing Laboratory, 1999.
- [Mota, 1997] Mota, A. *Formalização e Análise do SACI-1 em CSPZ*, Dissertação de Mestrado, Universidade Federal de Pernambuco, 1997, pp. 06-81.
- [Mota, 2001] Mota, A. & Sampaio, A. *Model-Checking CSP-Z: strategy, tool support and industrial application*. Science of Computer Programming, Elsevier, Netherlands. (40)1. 2001, pp. 59 – 96.
- [Mota, Tese em Andamento]. Mota, A. *Model Checking CSP_Z: Techniques to Overcome State Explosion*. Universidade Federal de Pernambuco. Tese em Andamento.
- [Roscoe, 1998] Roscoe, A. W. *The Theory and Practice of Concurrency*, Prentice Hall, 1998.
- [Smith, 1997] Smith, G. *A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems*, Technische Universität Berlin, Berlin, 1997.
- [Spivey, 1992] Spivey, M. *The Z Notation: A Reference Manual, 2^{dn} Edition*. Prentice Hall International, Englewood Cliffs, NJ, 1992.