Explorando o Uso de Sistemas de Reescrita como Ferramenta de Suporte no Contexto do Particionamento de Hardware e Software

Manoel Messias Menezes[1], André Luis Silva[1], Leila Silva[2]

Departamento de Ciência da Computação e Estatística
Universidade Federal de Sergipe (UFS)
CEP 49100-000 – São Cristóvão – SE – Brasil
{manoel, andre, leila}@ufs.br

Abstract. *The focus of this work is hardware/software partitioning verification. The adopted approach derives the partitioned system from the original description by applying transformation rules. The aim of this work is to show how the rewriting systems can be used to automatically prove the partitioning rules, as well as to implement the reduction strategy that guides the application of these rules. In this way, rewriting systems can be regarded as supporting tools for the construction of partitioning environments, whose emphasis is correctness.*

Resumo. *Este artigo insere-se no âmbito da verificação formal do particionamento de sistemas em componentes de hardware e software. A abordagem adotada deriva o sistema particionado a partir da descrição original, mediante o emprego de regras de transformação. O objetivo principal deste trabalho é explorar o uso de sistemas de reescrita na mecanização das provas das regras para o particionamento, bem como da estratégia de redução que guia a aplicação destas regras. Desta forma, sistemas de reescrita podem ser considerados ferramentas de suporte para a construção de um ambiente para o particionamento, cuja ênfase é o rigor formal.*

1. Introdução

Sistemas embarcados são largamente utilizados na indústria eletrônica e em geral são projetados usando-se componentes programáveis (componentes de software) e circuitos dedicados a uma aplicação específica (componentes de hardware). Projeto integrado de hardware e software, ou simplesmente *co-design*, é um paradigma de projeto de sistemas que envolvem componentes de hardware e de software.

Um ponto central em *co-design* é a maneira pela qual o sistema é particionado nos diversos componentes de hardware e de software. Neste contexto, diversas heurísticas para o particionamento foram sugeridas (dentre elas, [11, 13]). Estas abordagens validam o sistema particionado por simulação. Poucos trabalhos [4, 5] usam métodos formais para provar que algumas propriedades do sistema original são preservadas após o particionamento.

Este trabalho é parte do Projeto AbADeS, financiado pelo CNPq (proc. 520071/01-8).

[1] Aluno de Iniciação Científica; [2] Orientadora.

Dadas a crescente diversidade e complexidade das aplicações, a mera validação do sistema após o particionamento não é suficiente para se garantir segurança. Nesse sentido, a verificação formal, ou seja, a prova de que o sistema particionado preserva a semântica da descrição original, faz-se imprescindível.

Em [17, 18] Silva *et al* propõem uma metodologia para o particionamento cuja ênfase é o rigor formal. Esta abordagem aceita como entrada um programa descrito em occam [7] e, a partir da descrição original, deriva a descrição do sistema particionado mediante o emprego de regras de transformação, também escritas em occam. As provas manuais destas regras, apresentadas em [17], usam as leis algébricas que definem a semântica de occam [12]. Ainda em [17], a estratégia de redução da descrição original na descrição particionada é detalhada e provada correta, usando-se indução estrutural nos construtores da gramática adotada.

Este trabalho propõe-se a complementar o rigor formal de [17] pelo uso de sistemas de reescrita para provar automaticamente as regras propostas para o particionamento, bem como para implementar a estratégia de redução que guia a aplicação destas regras. Neste trabalho foram explorados os sistemas BOBJ [2] e CafeOBJ [10].

Na realidade, o presente trabalho está restrito a uma das fases da metodologia proposta em [17], a fase de quebra. Entretanto, o procedimento aqui ilustrado é genérico e pode ser aplicado em todo o processo do particionamento. Desta forma, sistemas de reescritas podem ser vistos como ferramentas de suporte para construção de um ambiente para o particionamento, cuja ênfase é o rigor formal. Resultados preliminares do trabalho desenvolvido encontram-se em [9], onde BOBJ é empregado na prova de regras do particionamento, sugerindo a adequabilidade do uso de sistemas de reescrita no âmbito de uma metodologia formal para *co-design*. Resultados mais consolidados encontram-se em [15], onde o sistema de reescrita CafeOBJ é explorado na prova das regras da fase de quebra, bem como na implementação da estratégia de redução da quebra. O objetivo deste artigo é condensar os principais resultados obtidos, visando ilustrar o trabalho desenvolvido durante a Iniciação Científica. Muito embora duas ferramentas tenham sido exploradas neste projeto, este artigo está restrito a CafeOBJ, por razões de concisão. Nas conclusões sumarizamos as principais diferenças e semelhanças entre as duas ferramentas.

Embora o trabalho aqui apresentado seja inovador no contexto de *co-design*, ressalva-se que a implementação de estratégias de redução algébrica usando sistemas de reescrita já foram propostas, por exemplo, por Sampaio [14], no contexto de verificação de compiladores (usando OBJ3 [3]) e por Lira [8], no contexto da implementação de uma estratégia de redução para linguagens orientadas a objeto (usando Maude [1]). Além disso, outros formalismos para verificação em hardware, não restritos ao particionamento, foram propostos e estão sumarizados em [6].

Este artigo está organizado como descrito a seguir. A Seção 2 descreve o formalismo utilizado e a Seção 3 apresenta a abordagem adotada para o particionamento, detalhando a fase de quebra. O sistema de reescrita CafeOBJ é descrito na Seção 4. Na Seção 5, o trabalho aqui desenvolvido é relatado. Finalmente, na Seção 6, algumas considerações finais e direções de trabalhos futuros são abordadas.

2. O Formalismo Utilizado: A Linguagem occam

A principal razão do uso de occam deve-se ao fato de que esta linguagem é regida por um conjunto de leis algébricas que definem a sua semântica [12]. Estas leis servem como axiomas para condução das provas das regras propostas para o particionamento.

Na realidade, a abordagem proposta em [17] adota um subconjunto de occam, definido a seguir, usando a sintaxe expressa no estilo BNF. Por conveniência, algumas vezes a sintaxe de occam é apresentada linearmente. Por exemplo, pode-se escrever SEQ(P_1, P_2, ..., P_n) ao invés de se utilizar o estilo vertical padrão.

```
P ::= SKIP | STOP | x := e | ch ? x | ch ! e | IF(b1 P1, b2 P2, …, bn Pn)
        | ALT(b1 & g1 P1, b2 & g2 P2, …, bn & gn Pn) | SEQ(P1, P2, …, Pn)
        | PAR(P1, P2, …, Pn) | VAR x: P | CHAN ch: P
```

A seguir descrevemos brevemente estes comandos; para mais detalhes veja, por exemplo, [7]. O comando SKIP não tem efeito semântico, comportando-se como um comando nulo e STOP é o processo canônico para expressar *deadlock*, não podendo realizar qualquer progresso. Os comandos ch?x, ch!e e x:=e são os comandos de entrada, de saída e de atribuição, respectivamente; a comunicação em occam é síncrona. Os comandos IF e ALT selecionam um processo a executar, baseados numa condição (IF) ou numa guarda (ALT). A escolha do IF é determinística, ou seja, a condição de índice mais baixo que se tornar verdadeira executa o processo a ela correspondente; mais de uma condição pode ser verdadeira num dado momento. Já o comportamento do ALT é não determinístico, ativando, aleatoriamente, o processo correspondente a uma das guardas que for satisfeita. Enquanto a condição de um IF é sempre uma expressão booleana, a guarda de um ALT envolve tipicamente comandos de entrada. Os comandos SEQ e PAR representam a composição de programas em seqüência e em paralelo, respectivamente. Os comandos VAR e CHAN denotam declarações de variáveis e canais, respectivamente. Neste trabalho é omitida a especificação de tipos de variáveis e canais, tornando implícita a validade de quaisquer tipos nestas declarações. Apesar da abordagem não lidar com laços arbitrários, replicadores são permitidos nos comandos IF, ALT, SEQ e PAR, possibilitando o tratamento de vetores de processos. Por razões de concisão, replicadores não são tratados neste artigo. Assim, omitimos esta construção na descrição da sintaxe, bem como na descrição da estratégia da quebra (Seção 3.1).

Na abordagem adotada para o particionamento, o subconjunto de occam foi estendido para incluir novos construtores, cuja finalidade é servir como anotações para guiar a estratégia de aplicação das regras. Estes construtores não têm nenhum efeito semântico. O construtor BOX, por exemplo, indica que um conjunto de processos pode ser considerado um *processo atômico* no fluxo do particionamento. Processos atômicos não são transformados na fase de quebra.

Como mencionado anteriormente, a semântica de occam é regida por um conjunto de leis algébricas. Apresentamos aqui apenas as leis necessárias para compreensão deste trabalho. Cada uma destas leis possui um nome, que indica seu uso, além de um número. A justificativa operacional de cada lei foi extraída de [12].

As duas primeiras leis expressam a identidade para a composição seqüencial e paralela. O operador SEQ executa um certo número de processos em seqüência. Se não tem argumentos, ele simplesmente termina. Caso contrário, executa o primeiro argumento até terminá-lo e então executa o restante em seqüência. Portanto, o operador

SEQ obedece a Lei 3 (*SEQ assoc*). O operador PAR também é associativo e este fato é capturado pela Lei 4 (*PAR assoc*).

Lei 1 *(SEQ-SKIP unit)* $SEQ(SKIP, P) = SEQ(P, SKIP) = P$.
Lei 2 (*PAR-SKIP unit)* $PAR(SKIP, P) = PAR(P , SKIP) = P$.
Lei 3 *(SEQ assoc)* $SEQ(P_1, P_2, \ldots, P_n) = SEQ(P_1, SEQ(P_2, \ldots, P_n))$.
Lei 4 (*PAR assoc*) $PAR(P_1, P_2, \ldots, P_n) = PAR(P_1, PAR(P_2, \ldots, P_n))$.

Uma condicional C é ou uma expressão booleana seguida de um processo ou um construtor IF. A Lei 6 (*IF assoc*) é aplicada para desaninhar construtores IFs, tal que todos os argumentos sejam expressões booleanas seguidas de processo. Similarmente, tem-se uma lei para o construtor ALT.

Lei 5 *(IF assoc)* $IF(C_1, IF(C_2), C_3) = IF(C_1, C_2, C_3)$.
Lei 6 (*ALT assoc*) $ALT(ALT(G_1), G_2) = ALT(G_1, G_2)$.

Quando P não termina imediatamente, o comportamento inicial de SEQ(P,Q) é aquele definido por P. Desta forma, SEQ distribui sobre o IF, quando este é o primeiro processo a executar.

Lei 7 *(SEQ-IF distrib)* $SEQ(IF(b_1 P_1, \ldots, b_k P_k), Q) = IF(b_1\ SEQ(P_1, Q), \ldots, b_k\ SEQ(P_k, Q))$.

O escopo de um canal local pode ser aumentado sem nenhum efeito, caso ele não interfira em nenhum outro canal de mesmo nome.

Lei 8 (*CHAN-PAR*) $PAR((CHAN\ ch:\ P)\ ,\ Q) = CHAN\ ch:\ PAR(P\ ,\ Q)$, se ch não é livre em Q.

Não importa se as variáveis são declaradas em uma lista ou separadamente. A Lei 9 (*VAR assoc*) captura este fato. Podemos modificar o nome de uma variável local, usando a Lei 10 (*VAR rename*), se este nome não aparece *livre* em P, ou seja, se a variável não está declarada no escopo de P.

Lei 9 (*VAR assoc*) $VAR\ x_1 : (VAR\ x_2 : (\ldots VAR\ x_n : P))\ldots) = VAR\ x_1\ ,\ x_2\ ,\ \ldots,\ x_n : P$.
Lei 10 (*VAR rename*) $VAR\ x:\ P = VAR\ y:\ P[y/x]$, se y não é livre em P.

Os construtores introduzidos em [17], para a condução do particionamento, não têm efeito semântico. Para o construtor BOX isto é capturado pela lei a seguir.

Lei 11 *(BOX unit)* $BOX(P) = P$.

3. A Abordagem para o Particionamento

A abordagem para o particionamento proposta em [17] compreende três fases: *quebra*, *definição de componentes* e *junção*. O objetivo da fase de quebra é a transformação do sistema numa descrição que é um conjunto de processos paralelos, mediante o emprego de regras algébricas. Na fase de definição de componentes, heurísticas são aplicadas para se determinar quais processos comporão os componentes de hardware e de software e de que forma estes processos serão combinados, se em série ou em paralelo. A descrição final do sistema particionado em componentes de hardware e software é de fato obtida na fase de junção, mediante o emprego de transformações algébricas.

Ao todo noventa regras para o particionamento foram propostas e provadas manualmente em [17]. Com o objetivo de mostrar como sistemas de reescritas (em particular, CafeOBJ) podem ser utilizados na automação das provas destas regras e na implementação da estratégia de derivação do sistema particionado, este artigo concentra-se na fase de quebra. Como mencionado anteriormente, o procedimento aqui

ilustrado é genérico e pode ser estendido a todo o fluxo do particionamento. A seguir, descrevemos brevemente a estratégia de redução à forma normal de quebra.

3.1 A Estratégia de Quebra

O objetivo da fase de quebra é transformar a descrição original numa descrição contendo um conjunto de processos paralelos, obedecendo à forma normal dada na Definição 1.

Definição 1 (*Forma Normal de Quebra*) Uma descrição está na forma normal de quebra se ele tem a estrutura abaixo:

$$\texttt{CHAN } ch_1, ch_2, \ldots, ch_m\texttt{: PAR}(P_1, P_2, \ldots, P_r)$$

onde cada P_i, para $1 \leq i \leq r$, é *simples*.

O detalhamento das possíveis formas para processos simples não é necessário no contexto deste trabalho e pode ser encontrado em [17, 18]. Para se atingir a forma normal, uma estratégia de redução é aplicada, mediante o emprego de regras de transformação. Esta estratégia compreende dois passos principais. No primeiro passo, os comandos `IF` e `ALT` são reduzidos para a forma simples. No segundo passo, a descrição intermediária é paralelizada. Ao todo são necessárias e suficientes oito regras e algumas leis de occam para se reduzir um programa arbitrário (segundo a gramática expressa na Seção 2) em uma descrição na forma normal da quebra. Caso replicadores sejam considerados, algumas regras precisam ser generalizadas e mais duas regras são introduzidas. A estratégia da quebra pode ser sumarizada pelo seguinte algoritmo.

Algoritmo 1 (*Estratégia de Quebra*)

(1) Aplique exaustivamente as leis 5 (*IF assoc*), 6 (*ALT assoc*) e 9 (*VAR assoc*).
(2) Aplique exaustivamente as regras do primeiro passo.
(3) Aplique exaustivamente as leis 1 (*SEQ-SKIP unit*), 2 (*PAR-SKIP unit*), 3 (*SEQ assoc*), e 4 (*PAR assoc*).
(4) Aplique a Lei 11 (*VAR rename*), se necessário.
(5) Aplique exaustivamente as regras do segundo passo.
(6) Aplique exaustivamente as leis 4 (*PAR assoc*) e 8 (*CHAN-PAR*).

Por razões de concisão, neste artigo detalhamos apenas uma das regras aplicadas no primeiro passo; as demais podem ser encontradas em [17]. A Regra 1 (*IF fragmentation*), transforma um condicional arbitrário em uma seqüência de condicionais binários, um para cada ramo do condicional inicial. Esta transformação isola processos de ramos distintos do condicional, possibilitando uma análise mais flexível destes durante a fase de definição de componentes.

Regra 1 (*IF Fragmentation*)

$$\texttt{IF}(b_1\ P_1, b_2\ P_2, \ldots, b_n\ P_n)$$

=

$$\texttt{VAR } c_1, c_2, \ldots, c_n\texttt{ :}$$
$$\texttt{SEQ(BOX(SEQ(}c_1\texttt{ :=FALSE, }c_2\texttt{ :=FALSE, ..., }c_n\texttt{ :=FALSE)),}$$
$$\texttt{IF}(b_1\ c_1\texttt{ := TRUE, }b_2\ c_2\texttt{ := TRUE, ..., }b_n\ c_n\texttt{ := TRUE)}$$
$$\texttt{IF}(c_1\ P_1\texttt{,TRUE SKIP),IF}(c_2\ P_2\texttt{,TRUE SKIP),...,IF}(c_n\ P_n\texttt{,TRUE SKIP))}$$

desde que cada c_k seja uma variável nova, ocorrendo somente onde mostrado.

A função do primeiro operador `IF`, no lado direito da Regra 1 (*IF fragmentation*), é fazer a escolha e permitir a disposição seqüencial dos comandos condicionais subseqüentes. É por isso que as variáveis novas $c_1, c_2, \ldots, c_n$ são introduzidas. Caso contrário, a execução de um dos P_i pode interferir em algum c_j,

$j > i$. Observe que o comportamento do lado esquerdo e direito da regra é o mesmo. No lado esquerdo, o primeiro b_i verdadeiro ativa o processo P_i correspondente. No lado direito, se b_i é verdadeiro, c_i é verdadeiro e os demais c_k, $k \neq i$, são falsos. Assim, somente P_i irá executar.

Cada uma das regras propostas é provada em [17] a partir das leis básicas de occam. Para a condução desta prova inicia-se, por exemplo, no lado direito da equação (RHS) e mediante aplicação das leis de occam, obtém-se o lado esquerdo (LHS). Por razões de concisão, abaixo ilustramos somente os dois primeiros passos da prova do caso binário da Regra 1 (*IF Fragmentation*).

Prova do caso binário da Regra 1 (*IF fragmentation*)

```
   RHS
= {Leis 11(BOX unit) e 3(SEQ assoc)}
   VAR c1, c2 :
   SEQ(SEQ(c1 := FALSE, SEQ(c2 := FALSE, SEQ(IF(b1 c1 := TRUE, b2 c2 := TRUE),
       SEQ(IF(c1 P1,TRUE SKIP), IF(c2 P2, TRUE SKIP))))))
= {Lei 7(SEQ-IF distrib)}
   VAR c1, c2 :
   SEQ(c1 := FALSE, SEQ(c2 := FALSE, SEQ(IF(b1 c1 := TRUE, b2 c2 := TRUE),
       IF(c1 SEQ(P1, IF(c2 P2, TRUE SKIP)), TRUE SEQ(SKIP, IF(c2 P2, TRUE
                  SKIP))))))
= ... = LHS □
```

Este mesmo estilo de prova é o utilizado para derivar o sistema particionado, como mostrado a seguir. Para esta derivação as regras propostas são utilizadas, em conjunto com as leis de occam.

```
     Sistema original
=    {regras para o particionamento e leis de occam}
     Sistema particionado
```

4. CafeOBJ

CafeOBJ [10] faz parte de uma nova geração de linguagens de especificação e verificação algébrica, e é sucessora da família OBJ (OBJ1, OBJ2, OBJ3) [3].

A principal entidade de CafeOBJ são os módulos (*module*), os quais são compostos por tipos, operadores e equações. A sintaxe do módulo é dada por `module <ModId> {}`, onde `<ModId>` é o símbolo metasintático para um identificador de módulo.

Um módulo pode ser composto por três elementos. O primeiro deles, `imports`, especifica quais módulos devem ser importados, ou seja, herdados. Existem três formas de importação: `protecting` (o módulo importado não pode sofrer alteração), `extending` (o módulo importado pode ser estendido, porém a descrição original permanece inalterada) e `using` (o módulo importado pode ser estendido ou ter a descrição original modificada). O segundo elemento, `signature`, declara operadores, tipos e subtipos utilizados pelo módulo. Finalmente, `axioms` compreende variáveis e equações, as quais refletem o comportamento do módulo. Como ilustração considere o exemplo dado a seguir. O módulo `QUADINT` herda operações e outros módulos definidos em `NAT` e `INT`. Na seção `signature`, são definidos dois tipos `Nat` e `Int`. O símbolo `<` indica que `Nat` é um subtipo de `Int`. O operador `quad`, introduzido por `op`, recebe um argumento do tipo inteiro e retorna o quadrado deste argumento, que é do tipo natural. O comportamento de `quad` é expresso por uma equação, introduzida por `eq`.

```
module QUADINT{
imports {
   protecting (NAT)
   protecting (INT)}
signature {
   [Nat < Int]
   op quad : Int -> Nat }
axioms {
   var I : Int
   eq quad(I) = (I * I) . }}
```

5. Mecanização da Fase de Quebra Usando CafeOBJ

A automação da estratégia da quebra usando CafeOBJ compreende três etapas principais: a implementação das leis de occam e das regras propostas para a fase de quebra (Seção 5.1), a prova das regras propostas (Seção 5.2) e a implementação da estratégia de redução à forma normal da quebra (Seção 5.3). A seguir descrevemos brevemente cada uma destas etapas, as quais constituem o cerne deste trabalho.

5.1 Implementação das Leis de occam e das Regras do Particionamento

Para implementação das leis de occam e das regras do particionamento faz-se necessário definir um módulo `BASE`, no qual são declarados tipos e operadores de uso geral. Por exemplo, definem-se os tipos `List_Process` e `List_Conditional` que denotam lista de processos e lista de processos condicionais, respectivamente. Como exemplo de operadores deste módulo temos, o operador "`,`", que serve para concatenar processos, canais e variáveis, formando listas. Ao todo foram definidos 52 operadores de uso geral.

As leis de occam estão implementadas no módulo `OCCAM-LAWS`, perfazendo um montante de 28 leis. Por razões de concisão não apresentamos neste artigo um exemplo da implementação destas leis (veja [15, 16] para detalhes). Entretanto, esta implementação segue raciocínio análogo à implementação das regras mostrada a seguir.

As regras da fase de quebra estão implementadas no módulo `RULES`, o qual compreende a importação do módulo `BASE`, a definição da assinatura dos operadores e a implementação das equações que capturam o comportamento das regras (seção `axioms`). No exemplo a seguir, apresentamos apenas a implementação do âmago da Regra 1 (*IF Fragmentation*). Muitos procedimentos auxiliares não são detalhados. A descrição completa do módulo `RULES`, compreendendo a implementação das oito regras da fase de quebra, encontra-se disponível em [16].

```
1  module RULES{
2  imports{protecting(BASE)}
3  signature{…}
4  axioms{
5  vars LC   : List_Conditional
6  vars L L1 : List_Process
7  …
8  ceq (L , IF(LC) , L1) = (L , ifFrag(getV(nVar(LC) (LC , L , L1)) LC) , L1)
       if isSimples(LC) == false .}}
```

Nas linhas 5 e 6 são declaradas as variáveis necessárias para a implementação da Regra 1 (*IF Fragmentation*). A equação que representa a implementação da regra em questão encontra-se na Linha 8, e é aplicada quando o operador `IF` estiver entre os processos capturados por `L` e `L1`, e não for simples. Equações similares para aplicação da regra quando o operador `IF` estiver à esquerda ou à direita de uma lista de processos,

ou quando aparecer isoladamente, também são necessárias, mas não são aqui detalhadas por razões de concisão. Para implementação da regra, alguns operadores auxiliares foram utilizados. O operador `ifFrag` recebe uma lista de variáveis, geradas por `getV`, e uma lista de processos condicionais, os quais correspondem aos ramos do construtor `IF`. Este operador é responsável por gerar um processo na forma do lado direito da Regra 1 (*IF Fragmentation*). O operador `getV` recebe o número de variáveis necessárias e uma lista de processos e tem a função de criar as variáveis novas que são utilizadas pelo operador `ifFrag`. Estas variáveis não aparecem em nenhum processo da lista de processos que é argumento de `getV`. A quantidade de variáveis novas necessárias é calculada pelo operador `nVar`, o qual recebe uma lista de processos condicionais e devolve o número de processos presentes nesta lista.

5.2 Prova das Regras do Particionamento

Para garantir a corretude da estratégia de redução, faz-se necessária a prova das regras propostas para o particionamento. A automação destas provas segue um estilo similar às provas manuais dadas em [17]. Assim, para a prova do caso binário da Regra 1 (*IF Fragmentation*), parte-se do lado direito e através de aplicações sucessivas das leis já implementadas, deriva-se o lado esquerdo da regra. A exemplo da Seção 3.1, ilustramos apenas algumas reduções realizadas nos primeiros passos da prova desta regra. Além disso, a explicação foca apenas nos passos correspondentes à aplicação das leis de occam. Procedimentos auxiliares para a aplicação destas leis não são detalhados.

```
1   start VAR c1 , c2 : SEQ(BOX(SEQ((c1 := false), (c2 := false))),
       IF((b1 c1 := true), (b2 c2 := true)),IF((c1 P1), (true skip)),
       IF((c2 P2), (true skip))) .
2   apply .box-unit at (2 1 1) .
3   result VAR (c1 , c2) : SEQ(SEQ((c1 := false) , (c2 := false)),
      IF((b1 (c1 := true)),  (b2 (c2 := true))),
     IF((c1 P1) , (true skip)), IF((c2 P2) , (true skip))) : Process
4   ...
5   result VAR (c1 , c2) : SEQ((c1 := false) , SEQ((c2:= false) ,
      SEQ(IF((b1 (c1 := true)), (b2 (c2:= true))) ,
      IF((c1 P1), (true skip)), IF((c2 P2), (true skip))))) :Process
6   apply .seq-assoc at (2 1 2 1 2) .
7   result VAR (c1 , c2) : SEQ((c1 := false) , SEQ((c2 := false),
     SEQ(IF((b1 (c1:= true)),(b2 (c2:= true))), SEQ(IF((c1 P1),
           (true skip)) , IF((c2 P2),(true skip)))))) : Process
8   apply .seq-if-distrib at (2 1 2 1 2 1 2) .
9   ...
10  result VAR (c1 , c2) : SEQ((c1 := false) , SEQ((c2:= false),
      SEQ(IF((b1 (c1:= true)), (b2 (c2:= true))),
      IF((c1 SEQ(P1, IF((c2 P2) , (true skip)))),
     (true SEQ(skip , IF((c2 P2), (true skip))))))))) : Process
```

Após carregar o arquivo que contém os módulos `OCCAM-LAWS` e `BASE` e de declarar as constantes necessárias para a condução da prova, a prova inicia-se com o fornecimento do lado direito da Regra 1 (*IF Fragmentation*), através do comando `start` (Linha 1). Seguindo os passos vistos na Seção 3.1, a Lei 11 (*BOX unit*) é aplicada (Linha 2), resultando na primeira transformação, a qual consiste na eliminação do comando `BOX` (observe o resultado designado por `result`). Os parâmetros `(2 1 1)` são utilizados para capturar o termo sobre o qual será aplicado a transformação. Observe que o construtor `BOX` pertence ao segundo argumento do operador `VAR`; a lista de processos que se segue é o primeiro (único) argumento do operador `SEQ` e nesta lista, o processo que inclui o `BOX` é o primeiro. A prova prossegue com a aplicação de operadores auxiliares e da Lei 3 (*SEQ-assoc*), mostrada na Linha 6. Em seguida, aplica-se a Lei 7 (*SEQ-IF distrib*) (Linha 8) e todas as equações auxiliares utilizadas por esta

lei. Finalmente, o resultado do segundo passo da Regra 1 (*IF Fragmentation*) é alcançado, como pode ser observado comparando-se com a Seção 3.1. O restante da prova prossegue analogamente.

5.3 Implementação da Estratégia de Redução da Fase de Quebra

Para implementar a estratégia usando CafeOBJ, os módulos `OCCAM-LAWS` e `RULES` foram divididos em seis módulos, cada um deles encapsulando o conjunto de regras e leis empregados nos passos do Algoritmo 1 (*Estratégia de Quebra*). Estes módulos foram nomeados por `PASSO-i`, $1 \leq i \leq 6$. Além destes módulos, a implementação da estratégia usa o módulo `BASE`, mencionado na Seção 5.1. A aplicação da estratégia dá-se da seguinte forma. Inicialmente carrega-se o módulo `PASSO-1` na memória e indica-se o termo sobre o qual a estratégia será aplicada. Em seguida, aplicam-se as leis e regras contidas neste módulo e copia-se o resultado obtido, o qual servirá de entrada para o módulo `PASSO-2`. A estratégia prossegue de maneira análoga até o `PASSO-6`, onde o resultado obtido é a descrição do sistema na forma normal da quebra.

6. Conclusões

Verificação formal em *co-design* encontra-se ainda num estágio incipiente e são raras as abordagens a tratar o tema. Uma das contribuições deste trabalho é complementar o rigor formal do trabalho de Silva *et al* [17, 18], no sentido de mecanizar as provas das regras e a estratégia de redução usadas na fase de quebra. Em particular, durante a automatização das provas, foram detectadas três omissões de aplicações de leis básicas nas provas manuais das regras da quebra, apresentadas em [17], atestando a importância do uso de sistemas de reescrita como aqui proposto. Para validar a implementação da estratégia, foi desenvolvido o estudo de caso do programa da convolução, descrito em [16]. O número de reescritas necessárias foram de 250.049. No sistema operacional *Windows* 2000 consumiu-se 2 *min* e 9 *s*, enquanto que no *Red Hat Linux* 9 o tempo gasto foi de 14 *min* e 18 *s*.

No contexto deste trabalho foram explorados os sistemas de reescrita CafeOBJ e BOBJ. As duas ferramentas são acadêmicas e portanto, de reduzida amigabilidade. Apresentam funcionalidades similares, variando apenas na sintaxe. Entretanto, CafeOBJ se diferencia pela presença do atributo `memo`, o qual habilita o armazenamento dos resultados das reescritas a serem reutilizados quando necessário, reduzindo assim o tempo consumido quando da aplicação da estratégia de redução.

O uso de sistemas de reescrita se mostrou de grande valor para este trabalho. A implementação da sintaxe de occam, das leis básicas e das regras do particionamento não incorreu em grandes problemas. A implementação da estratégia de redução também foi desenvolvida naturalmente. Neste sentido, este trabalho é também uma contribuição à comunidade de CafeOBJ e BOBJ, pois constitui-se num estudo de caso de transformação de programas, no contexto destes sistemas de reescrita. Por outro lado, para a comunidade de *co-design*, este trabalho sugere que sistemas de reescrita podem ser considerados como ferramentas úteis de suporte a metodologias com ênfase no rigor formal.

Apesar deste artigo se restringir apenas a fase da quebra, o procedimento utilizado na mecanização das provas e da estratégia de redução é genérico e pode ser estendido a todo o fluxo do particionamento. O próximo passo é a implementação das

provas das regras da etapa de junção, bem como da estratégia de redução que guia a aplicação de tais regras, complementando assim, a formalização da abordagem proposta por Silva *et al* [17]. Assim, espera-se construir um ambiente para o particionamento que deverá ser utilizado no desenvolvimento de estudos de caso de maior porte.

Referências

[1] Clavel, M., Durán, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J. e Quesada, J. F. (1999) *Maude: Specification and Programing in Rewriting Logic*. Computer Science Laboratory-SRI International.

[2] Goguen, Joseph. *Fun with BOBJ*. Disponível on-line: http://www.cse.ucsd.edu/groups/tatami/bobj/. Último acesso 11/03/2003.

[3] Goguen, Joseph A., Winkler, T., Meseguer, J., Kokichi, F. e Jean P, J (1992). *Introducing OBJ**, Relatório Técnico. SRI-CSL-92-03, SRI International, Computer Science Laboratory.

[4] Hsieh, H., A. Sangiovanni-Vicentelli, F. Ballarin e L. Lavagno. (1999), Synchronous Equivalence for Embedded Systems: A Tool for Design Exploration, em *Proceedings of the International Conference on Computer-Aided Design*, pp. 505-509.

[5] Hughes, R. B. e Musgrave, G. (1996) The Lambda Approach to System Verification. Em *Hardware/Software Co-design*, Kluwer Academic Publisher, 427-451.

[6] Kern, C e M. Greenstreet. Formal Verification in Hardware Design: A Survey (1999), *ACM Transactions in Design Automation of Eletronic Systems*, 4(2):123-193.

[7] Jones, G. e Goldsmith, M. (1998) *Programing in occam 2*. Prentice-Hall.

[8] Lira, B, Cavalcanti, A. e Sampaio, A. (2002) Automation of a Normal Form Reduction Strategy for Object-Oriented Programming. *Proc. of 5th Workshop on Formal Methods*, 193-208.

[9] Menezes, M., Silva, A. e Silva, L. (2003) Verificação Formal da Etapa do Particionamento em Co-design Usando o Sistema de Reescrita BOBJ. *Anais do XXX Seminário Integrado de Hardware e Software*, pp. 195-210.

[10] Nakagawa.A., Sawada T. e Futatsugi, K (1999) *CafeOBJ user's manual ver 1.4.2*. Disponível on-line: http://www.ldl.jaist.ac.jp/cafeobj/doc/. Último acesso 01/07/2003.

[11] Niemann, R. e Marwedel, P. (1997) An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation of Embedded Systems*, 2(2):165-193.

[12] Roscoe, A. W. e Hoare, C.A.R. (1988) The Laws of occam programming. *Theoretical Computer Science*, 60:177-229.

[13] Saha, D., Mitra, R. S. e Basu, A. (1997) Hardware/Software Partitioning Using Genetic Algorithm. *Proc. of 10th International Conference on VLSI Design*, India, 155-160.

[14] Sampaio, A. (1997) An Algebraic Approach to Computer Design. *Volume 4 of Algebraic Methodology and Software Technology (AMAST) Series in Computing*, World Scientifc.

[15] Silva, A., Menezes, M. e Silva, L. (2003) Using CafeOBJ to implement a Reduction Strategy in the Context of Hardware/Software Partitioning. *Proc. of 6th Workshop on Formal Methods*, pp 43-58. Também em *Electronic Notes in Theoretical Computer Science* 95C pp. 63-82.

[16] Silva, A e Menezes, M. (2004) *Explorando Sistemas de Reescrita no Contexto de Co-design*, Monografia de final de curso, Universidade Federal Sergipe.

[17] Silva, L. (2000) *An Algebraic Approach to Hardware/Software Partitioning*. Tese de Doutorado. Universidade Federal de Pernambuco, Recife, Pernambuco.

[18] Silva L., Sampaio A. e Barros E. (2004) A Constructive Approach to Hardware/Software Partitioning. *Journal of Formal Methods in Systems Design*, 24:45-90.