III Simpósio Brasileiro de
Linguagens de Programação

SBLP 1999

TUTORIAL III

MÁQUINAS DE ESTADO ABSTRATAS

Fábio Tirelo, Marcelo A. Maia, Vladimir O. Di Iorio e Roberto S. Bigonha

Porto Alegre - RS

05 a -7 de maio de 1999

Maquinas de Estado Abstratas

Fabio Tirelo [*]
ftirelo@dcc.ufmg.br

Marcelo A. Maia [†]
marcmaia@dcc.ufmg.br

Vladimir O. Di Iorio [‡]
vladimir@dcc.ufmg.br

Roberto S. Bigonha [§]
bigonha@dcc.ufmg.br

Resumo

Maquinas de Estado Abstratas (ASM) sco a base de um mitodo formal, poderoso e elegante de especificagco e verificagco de sistemas, que ja foi aplicado com sucesso no projeto e analise de diversos tipos de sistemas dinbmicos discretos. Ao contrario de muitos mitodos formais de especificagco, como, por exemplo, a Sembntica Denotacional, a metodologia ASM utiliza conceitos simples e bem conhecidos, tornando a leitura e a escrita da especificagco bastante direta. A metodologia possui base matematica rigorosa, o que permite a demonstragco formal de propriedades do sistema especificado. Outra vantagem i a possibilidade de executar a especificagco, o que auxilia na verificagco de sua corregco em relagco ao mundo real. Neste tutorial, apresentaremos a metodologia de especificagco de sistemas de ASM e mostraremos sua aplicagco para especificagco de sembntica de linguagens de programagco. **Palavras chave:** Maquinas de Estado Abstratas, Especificagco Formal, Sembntica de Linguagens de Programagco

Abstract

Abstract State Machines (ASM) are the basis of a powerful and elegant formal method for specification and verification of systems, which has already been successfully applied on design and analysis of several discrete dynamic systems. Differently from several formal specification methods, e.g., Denotational Semantics, ASM methodology uses simple and well-known concepts, which eases reading and writing. The methodology has rigorous mathematical basis, which allows formal demonstration of properties of a system. ASM also allows specification execution in order to verify by testing its correctness with respect to the real specified system. In this tutorial, we present the ASM methodology for systems specification, and show its application on specification of Programming Languages Semantics. **Keywords:** Abstract State Machines, Formal Specification, Semantics of Programming Languages

[*]Departamento de Ciência da Computação, Universidade Federal de Minas Gerais.
[†]Departamento de Computagco, Universidade Federal de Ouro Preto.
[‡]Departamento de Informatica, Universidade Federal de Vigosa.
[§]Departamento de Ciênciacia da Computação, Universidade Federal de Minas Gerais.

1 Introdugco

Maquinas de Estado Abstratas (ASM, do ingljs *Abstract State Machines*), introduzidas por Yuri Gurevich em [20, 22], sco um conceito poderoso e elegante para modelagem matematica de sistemas dinbmicos discretos. A idiia original era prover sembntica operacional para algoritmos elaborando a tese implmcita de Turing, isto i, "todo algoritmo i simulado por uma maquina de Turing apropriada". Desta maneira, Turing deu sembntica operacional para algoritmos. Entretanto, esta sembntica i muito inconveniente, pois pode ser necessaria uma longa seq—jncia de passos na maquina de Turing para simular um znico passo do algoritmo. Assim, as ASM foram criadas com o objetivo de simular algoritmos de maneira mais prsxima e natural [20]. A Tese de ASM Seq—encial diz que "todo algoritmo seq—encial, em qualquer nmvel de abstragco, pode ser visto como uma ASM"[23]. As ASM possuem recursos para modelar interagco, possibilitando formalizar as agues do ambiente no qual o sistema esta inserido. A modelagem de tais agues por meio de maquinas de Turing pode gerar um nzmero exponencial de estados, sendo impossmvel de ser implementada na pratica. A metodologia que apresentaremos neste tutorial provj recursos expressivos para especificar a sembntica operacional de sistemas dinbmicos discretos, em um nmvel de abstragco natural e de uma maneira direta e essencialmente livre de codificagco [14]. Com isso, tem-se por objetivo diminuir a distbncia que ha entre modelos formais de computagco e mitodos praticos de especificagco [22]. Para tal, definem-se maquinas de estado que simulem passo a passo o algoritmo. A metodologia utiliza conceitos simples e bem conhecidos, o que facilita a leitura e a escrita de especificagues de sistemas. Alim disso, ela pode ser utilizada inclusive por novatos e por praticos em computagco que possuam pouca base matematica. Na literatura, ha varios exemplos de utilizagco de ASM na especificagco formal de sistemas, dentre os quais podemos citar: arquiteturas [6, 7, 10], linguagens de programagco [11, 12, 13, 24, 36], sistemas distribumdos [1, 2, 8, 25] e de tempo real [18, 26], entre outros [9]. Este tutorial esta organizado da seguinte maneira:

- na Segco 2, apresentaremos uma introdugco informal ao modelo ASM de especificagco, mostrando diversos exemplos para ilustrar o mitodo;
- na Segco 3, apresentaremos o modelo formal de ASM;
- na Segco 4, mostraremos um estudo de caso em definigco de sembntica operacional de linguagens de programagco, definindo formalmente em ASM uma linguagem imperativa de pequeno porte;
- na Segco 5, mostraremos a extensco de ASM para lidar com mzltiplos agentes, metodologia utilizada para especificar algoritmos distribumdos e assmncronos;
- na Segco 6, faremos uma avaliagco da metodologia, comparando com outras metodologias utilizadas para definigco formal de sistemas dinbmicos discretos;
- finalmente, na Segco 7, apresentaremos algumas consideragues finais.

ostos por diferentes fungues e relagues. Dizemos que a interpretagco dos nomes pertencentes ao vocabulario i modificada de estado para estado durante a execugco. Mais precisamente, um *vocabulario* Υ (pronuncia-se como "Y") i um conjunto de nomes de fungues e relagues, cada nome com uma aridade fixa associada. Por exemplo, suponha que a aridade da fungco de nome i seja 0 e da fungco de nome f seja 1. Temos que o conjunto $\{i, f\}$ i um vocabulario. Os nomes de relagues de zero argumento *true*, *false*, o nome de fungco de zero argumento *undef*, os operadores booleanas usuais e o sinal de igualdade estco presentes em todo vocabulario. Um *estado* S de vocabulario Υ i um conjunto X, denominado o *superuniverso* de S, junto com as interpretagues, em X, dos nomes de fungues e relagues pertencentes a Υ. Se f i um nome de fungco de aridade r, entco f i interpretado como uma fungco $\boldsymbol{f} : X^r \to X$. Se f i um nome de relagco de aridade r, entco f i interpretado como uma fungco $\boldsymbol{f} : X^r \to \{true, false\}$. Se U i um nome de relagco pertencente a Υ, entco o conjunto $U = \{\bar{x} : U(\bar{x}) = true\}$ i um *universo* contido em X. Neste caso, dizemos que $U(\bar{x}) = true$ e $\bar{x} \in U$ sco equivalentes. Um novo estado i criado a partir do estado atual, por uma regra de transigco, por meio da mudanga da interpretagco de cada nome de fungco. As regras mais simples (regras basicas) sco *atualizagco*, *bloco* e *condicional*. Uma regra de atualizagco i da forma $R \equiv f(\bar{x}) := y$, onde o comprimento de $\bar{x}$ i igual ' aridade de f. Esta regra cria, a partir de um estado S, um novo estado S', tal que a interpretagco do nome de fungco f i uma fungco $\boldsymbol{f} : X^r \to X$, que, no ponto $\bar{\boldsymbol{x}}$ (a avaliagco da tupla $\bar{x}$), o seu valor i $\boldsymbol{y}$ (a avaliagco de y). Por exemplo, a regra $f(1) := 2$ determina um novo estado no qual o valor da fungco $\boldsymbol{f}$, no ponto 1, i 2. Uma regra condicional da forma $R \equiv$ **if** g **then** R_1 **else** R_2 **endif** tem a seguinte sembntica: se a expressco g avaliar em *verdadeiro*, entco o estado resultante i o resultado da regra R_1; caso contrario, o estado resultante i o resultado da regra R_2. Uma regra bloco da forma $R \equiv R_1, \ldots, R_n$ tem a seguinte sembntica: o estado formado por R i o resultado da execugco de todas as regras R_i em paralelo. Por exemplo, a execugco da regra bloco

$$f(1) := 2, f(2) := 4$$

produz um novo estado, no qual o valor da fungco $\boldsymbol{f}$, no ponto 1, i 2 e, no ponto 2, i 4. Uma especificagco ASM contim a definigco de um estado inicial, S_0 e uma regra, R, que define as mudangas de estado. A execugco de uma especificagco i uma seq—jncia de estados $\langle S_n : n \geq 0 \rangle$, onde um estado S_i i obtido executando a regra R em S_{i-1}. Para exemplificar estes conceitos, apresentaremos uma especificagco ASM para a fungco fatorial.

Exemplo 2.1 (Fungco Fatorial) Suponha que o estado inicial S_0 seja dado com o vocabulario $\Upsilon = \{f, i\}$, onde i i um nome de fungco de zero argumento, interpretado como 0, e f i um nome de fungco unaria, interpretado como a fungco $\boldsymbol{f} = \lambda x.x = 0 \to 1, undef$[1]. A regra da especificagco i:

$$f(i+1) := (i+1) \times f(i), i := i + 1$$

[1]Se $f = \lambda x.E$, entco f i uma fungco dada por $f(x) = E$. Mais detalhes sobre a notagco lambda pode ser encontrada em [31]. A notagco $a \to b, c$, utilizada em [19], i equivalente a **if** a **then** b **else** c.

Estado	0	1	2	3	4	$\cdots$	n
S_0	1	*undef*	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_1	1	1	*undef*	*undef*	*undef*	$\cdots$	*undef*
S_2	1	1	2	*undef*	*undef*	$\cdots$	*undef*
S_3	1	1	2	6	*undef*	$\cdots$	*undef*
S_4	1	1	2	6	24	$\cdots$	*undef*
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
S_n	1	1	2	6	24	$\cdots$	$n!$

Figura 1: Interpretagco do nome de fungco *fat* nos estados que formam a execugco da ASM do Fatorial.

Ao executarmos a regra no estado S_0, obtemos um novo estado S_1, no qual i i interpretado como 1 e f i interpretado como a fungco

$$\boldsymbol{f} = \lambda x.x = 0 \rightarrow 1, x = 1 \rightarrow 1, \mathit{undef}$$

Como podemos ver na Figura 1, ao executarmos a regra no estado S_1, obtemos um novo estado S_2, no qual i i interpretado como 2 e a interpretagco de f, no ponto 2, passa a ser igual a 2. Da mesma forma, obtemos os estados $S_3, S_4, ...$ Pode-se ver facilmente que, no estado S_k, $k \geq i$, a interpretagco de f, $\boldsymbol{f}$, i uma fungco que, aplicada a $\boldsymbol{i}$, retorna $\boldsymbol{i}!$. A demonstragco formal sera feita na Segco 3.2. □

Um aspecto importante que deve ser modelado na especificagco de um sistema i que, em geral, sistemas sco afetados pelo ambiente. Suponha que o ambiente se manifeste por meio de algumas fungues basicas $e_1, \ldots, e_k$, chamadas *fungues externas*. Um exemplo tmpico de fungco externa i uma entrada fornecida pelo usuario. Pode-se pensar em fungues externas como *oraculos*, tais que, a especificagco fornece argumentos e o oraculo fornece o resultado [22]. Alim das regras basicas, mostradas acima, ha tambim as regras que utilizam variaveis[2]. Em ASM, variaveis sco utilizadas para modelar paralelismo, nco-determinismo e a "criagco"de novos elementos. As regras que utilizam variaveis sco as regras *import*, *choose* e *var*. Uma regra *import* i da forma

$$\textbf{import } v \ R_0 \textbf{ endimport},$$

onde v i uma variavel e R_0 i uma regra. O efeito dessa regra i executar R_0 em um estado em que a variavel v esta associada a um valor importado de um universo especial chamado *Reserve*. Este universo esta contido em X, o superuniverso dos estados da maquina, e contim todos os elementos que serco importados. Em geral, regras *import* sco utilizadas para estender universos, isto i, adicionar novos elementos aos universos. Assim, regras da forma

$$\textbf{import } v \ U(v) := \mathit{true}, R_0 \textbf{ endimport}$$

onde U i um nome de universo e R_0 i uma regra, podem ser escritas utilizando a seguinte regra *extend*:

$$\textbf{extend } U \textbf{ with } v \ R_0 \textbf{ endextend}.$$

[2]Variaveis sco smmbolos que podem denotar elementos do superuniverso.

Uma regra *choose* i da forma

choose v **in** U **satisfying** g R_0 **endchoose**

onde v i uma variavel, U i o nome de um universo finito, g i um termo booleano e R_0 i uma regra. O efeito desta regra i executar a regra R_0 em um estado no qual a variavel v esta associada a um valor pertencente ao universo U. Este valor i escolhido de maneira nco-determinista e satisfaz a guarda g. Uma regra *var* i da forma

var v **ranges over** U R_0 **endvar,**

onde v i uma variavel, U i o nome de um universo finito e R_0 i uma regra. O efeito desta regra i criar uma instbncia de R_0 para cada elemento pertencente ao universo U. Em cada instbncia de R_0, a variavel v esta associado ao elemento correspondente de U. Apss criadas as instbncias, todas sco executadas em paralelo. ASM possui ainda recursos para expressar paralelismo assmncrono, como veremos na Segco 5. Alim disso, para simplificar as especificagues, supomos que conjuntos como o dos nzmeros inteiros, dos racionais, das listas e das partes de conjuntos estejam contidos no superuniverso de um estado.

2.1 Exemplos

Nesta segco, apresentaremos alguns exemplos de definigco de sistemas em ASM.

Exemplo 2.2 (Pesquisa Binaria) Este exemplo tem por finalidade mostrar a utilizagco de fungues e regras basicas para especificagco de algoritmos simples. O problema consiste em encontrar um valor k em um arranjo ordenado de inteiros, a, de comprimento n, indexado de 1 a n. Um algoritmo bastante utilizado para resolugco deste problema i a *busca binaria*, que, a cada passo, restringe o espago de busca da chave. Na solugco proposta, o arranjo a sera sera representado pela fungco f, de modo que $f(k) = a[k]$, para k inteiro. No estado inicial, *inf* i interpretado como 1, *sup* i interpretado como n, k i interpretado como o valor da chave de pesquisa, o nome de relagco *encontrado* i interpretado como *false* e o nome de fungco f i interpretado como uma fungco que, aplicada a um nzmero inteiro i, $1 \leq i \leq n$, retorna $a[k]$. Todos os outros nomes de fungco sco interpretados como *undef*. A regra da especificagco i

if (**not** *encontrado* **and** $inf \leq sup$) **then**
  if $(k = f((inf + sup)/2))$ **then**
    $encontrado := true$,
    $pos := (inf + sup)/2$
  elseif $(k < f((inf + sup)/2))$ **then**
    $sup := (inf + sup)/2 - 1$
  else
    $inf := (inf + sup)/2 + 1$
  endif
endif

A maquina executa diversos passos ati que a chave seja encontrada ou o algoritmo possa determinar que a chave nco esta presente no arranjo. Se a chave for encontrada, entco $encontrado = true$ e o nome de fungco pos i interpretado como a posigco no arranjo onde esta a chave, isto i, $f(pos) = a[pos] = k$. □

Exemplo 2.3 (Ordenagco por selegco) Da mesma forma que no Exemplo 2.2, a fungco f representara os elementos que desejamos ordenar. Quando a execugco convergir, esperamos que se $1 \leq i \leq j \leq n$, entco $f(i) \leq f(j)$. No estado inicial, o nome de fungco *Modo* i interpretado como 1, i i interpretado como 1 e f i interpretado como no Exemplo 2.2. Todos os outros nomes de fungco sco interpretados como *undef*. A regra da especificagco i a seguinte:

if $Modo = 1$ **and** $i < n$ **then**
    $k := i$, $j := i + 1$, $Modo := 2$
elseif $Modo = 2$ **then**
    if $j > n$ **then**
        $Modo := 3$
    elseif $f(j) < f(k)$ **then**
        $k := j$
    endif
    $j := j + 1$
elseif $Modo = 3$ **then**
    if $k \neq i$ **then**
        $f(k) := f(i)$, $f(i) := f(k)$
    endif
    $i := i + 1$, $Modo := 1$
endif

A maquina executa ati que i atinja o valor n. Neste estado, o arranjo esta ordenado. □

Exemplo 2.4 (Nzmeros Primos) Neste exemplo, especificaremos um algoritmo bastante simples para encontrar todos os primos menores ou iguais a um nzmero n qualquer. Para isso, definiremos um universo *Numeros*, subconjunto dos nzmeros inteiros, tal que $Numeros = \{x \in Inteiros : 2 \leq x \leq n\}$. O vocabulario contim os nomes de fungco *primo* de aridade 1 e x e n de aridade zero. No estado inicial, faremos x ser interpretado como o nzmero 3 e a fungco *primo* aplicada a qualquer elemento pertencente ao universo *Numeros* retornar *true*. A regra marcara com false todos os nzmeros de 2 a n que nco forem primos. A regra da especificagco i a seguinte:

if $x \leq n$ **then**
    var y **ranges over** *Numeros*
        if $y < x$ **and** $x\%y = 0$ **then**
            $primo(x) := false$
        endif
    endvar
    $x := x + 1$
endif

A maquina executara varios passos ati que x atinja o valor $n + 1$. Neste estado, o nome de fungco *primos* sera interpretada como uma fungco que, aplicada a um nzmero inteiro k, $2 \leq k \leq n$, retorna *true*, se k for um nzmero primo, ou *false*, se k for um nzmero composto. □

Exemplo 2.5 (Sistema Operacional) Consideraremos neste exemplo o problema de especificagco do nzcleo de um sistema operacional multiprogramado simples. Especificaremos as seguinte operagues: criagco de um novo processo, escalonamento de processos, difusco de uma mensagem para todos os processos ativos e recebimento de um sinal de interrupgco. Suponha que o universo *Processes* seja o conjunto dos processos criados no sistema operacional. A cada processo esta associado um nzmero inteiro, a sua identificagco. Esta associagco i representada pela fungco *id*, de aridade 1. Cada processo possui tambim uma "caixa de mensagens", na qual o processo recebe as mensagens que lhe foram enviadas. Esta caixa de mensagens i representada por uma lista de elementos. Ha tambim a fungco *owner*, que associa cada recurso da maquina ao processo que o esta utilizando no momento, e a relagco *waiting*, de aridade 2, que relaciona os processos com os recursos que eles estco esperando. A operagco de criagco de um novo processo pode ser especificada atravis da regra *import* abaixo. Um nzmero inteiro i associado ao novo processo por meio da atualizagco da fungco *id*.

import v
  $Processes(v) := true$**,**
  $id(v) := num\_procs + 1$**,**
  $num\_procs := num\_procs + 1$
endimport

A regra abaixo especifica um protocolo de escalonamento, onde o prsximo processo a utilizar um recurso i escolhido aleatoriamente entre os processos que estco esperando para utiliza-lo. A regra *choose* permite a especificagco de uma escolha nco-determinista.

choose v **in** *Processes* **satisfying** $waiting(v, recurso)$
  $owner(recurso) := id(v)$
endchoose

A regra abaixo especifica a difusco da mensagem representada por *mesg* para todos os processos ativos. Esta operagco i modelada por meio de uma regra *var*. Para cada elemento pertencente ao universo *Processes*, uma nova instbncia da regra condicional associada ‘ regra *var* i criada. Em cada uma destas instbncias, a variavel p esta associada ao elemento correspondente do universo *Processes*.

var p **ranges over** *Processes*
  if $ativo(p)$ **then**
    $messages(id(p)) := append(messages(id(p)), mesg)$
  endif
endvar

O recebimento de um sinal de interrupgco i modelado utilizando a fungco externa *interrupted*, que contim o valor *true* se houve uma interrupgco ou contim o valor *false*, caso contrario. A cada iteragco da maquina, esta fungco i testada. Se seu valor for verdadeiro, as operagues necessarias sco executadas.

if *interrupted* **then**
    `...trata a interrupgco...`
else
    `...continua realizando as operagues normais...`
endif

□

3 O Modelo Formal de ASM

Nesta segco apresentaremos o modelo formal de ASM, mostrando um pequeno exemplo de definigco ASM e uma prova de corregco da especificagco apresentada.

3.1 Definigco da Maquina

ASM sco sistemas de transigco que especificam computagco cujos estados sco algebras [14]. Os universos de algebras que formam os estados da computagco constituem o superuniverso da ASM.

Definição 3.1 (Vocabulario) Um *vocabulario* Υ i uma colegco finita de nomes de fungco e relagco, cada nome com uma aridade fixa. Estco presentes em todo vocabulario: o sinal de igualdade, *true*, *false* e *undef* e os operadores booleanas usuais. □

A fungco *undef* i utilizada para modelar fungues parciais.

Definição 3.2 (Estado) Um *estado* S i uma algebra [17], dada por:

- um vocabulario Υ, o vocabulario do estado S;
- um conjunto X, denominado o superuniverso de S, no qual estco contidos os conjuntos dos nzmeros inteiros, dos valores lsgicos, dos nzmeros racionais, das cadeias de caracteres, das listas, das tuplas e das partes de conjuntos;
- uma fungco $Val : \Upsilon \rightarrow X^* \rightarrow X$, que fornece a interpretagco dos nomes de fungues pertencentes a Υ em fungues de $X^* \rightarrow X$.

Para os conjuntos do superuniverso, estco definidas as operagues usuais (como, no caso dos inteiros, adigco, multiplicagco, etc.). □

A nogco de estado i importante, pois cada estado constitui um passo na execugco da maquina.

Definição 3.3 (Fungues Estaticas e Fungues Dinbmicas) Uma fungco i *dinbmica* se puder sofrer atualizagues, isto i, se durante uma mudanga de estado, a sua interpretagco puder ser modificada em alguns pontos. Caso contrario, dizemos que a fungco i *estatica*. □

Os conceitos de fungues estaticas e dinbmicas sco importantes nas definigues de regras e atualizagco, dadas abaixo. Basicamente, o carater dinbmico do sistema i modelado pelas alteragues, de estado para estado, na interpretagco de fungues dinbmicas.

Definição 3.4 (Termo) *Termos* sco definidos recursivamente como:

- uma variavel i um termo;
- um nome de fungco ou relagco de zero argumento i um termo;
- se f i um nome de fungco ou relagco r-aria, $r > 0$, e $\bar{x} = (x_1, \ldots, x_r)$, entco $f(\bar{x})$ i um termo.

□

Termos sem variaveis sco interpretados, em um estado S, da seguinte maneira:

- se f i um nome de fungco ou relagco de zero argumento, sua interpretagco i $Val_S(f)$;
- se f i um nome de fungco r-aria e $(x_1, \ldots, x_r)$ i uma tupla de comprimento r, entco
$$Val_S(f(x_1, \ldots, x_r)) = Val_S(f)(Val_S(x_1), \ldots, Val_S(x_r))$$

A interpretagco de termos que contjm variaveis sera mostrada na Segco 3.3, onde definiremos as regras nco-basicas, isto i, as regras que utilizam variaveis.

Definição 3.5 (Enderego) Um *enderego* em um estado $S = (\Upsilon, X, Val)$ i um par $(f, \bar{x})$, onde $f \in \Upsilon$ i um nome de fungco dinbmica, e $\bar{x} \in X^*$ i uma tupla cujo comprimento i igual ‘ aridade de f. □

Definição 3.6 (Atualizagco) Uma *atualizagco* em um estado $S = (\Upsilon, X, Val)$ i um par (l, y), onde l i um enderego em S e $y \in X$. □

A nogco de atualizagco i importante para a definigco de regras e disparo de regras. Para cada regra definimos um conjunto de atualizagues, que contera os pontos em que havera mudanga na interpretagco de alguns nomes de fungues dinbmicas no estado seguinte.

Definição 3.7 (Consistjncia de um Conjunto de Atualizagues) O conjunto de atualizagues Δ i *consistente*, se

$$((f, \bar{x}), y_1) \in \Delta \wedge ((f, \bar{x}), y_2) \in \Delta \Rightarrow y_1 = y_2,$$

isto i, se nco houver, em Δ, duas atualizagues diferentes para o mesmo enderego. □

Definição 3.8 (Regra) *Regras* sco definidas recursivamente por:

- a regra de atualizagco $R \equiv f(\bar{t}) := t_0$ i uma regra; nesta expressco, f i um nome de fungco dinbmica, e t_0 e $\bar{t}$ sco termos; se f i um nome de relagco, entco t_0 deve ser booleano. Definiremos o conjunto de atualizagues de R em um estado S, $Updates(R, S)$, como o conjunto $\{(l, y)\}$, onde $l = (f, Val_S(\bar{t}))$ e $y = Val_S(t_0)$;

- um bloco de regras $R \equiv R_1, \ldots, R_k$ i uma regra; o seu conjunto de atualizagues i dado por

$$Updates\,(R, S) = \bigcup_{i=1}^{k} Updates\,(R_i, S);$$

  isto significa que, para disparar um bloco de regras, disparam-se todas as regras que o compuem *simultaneamente*. Note que a ordem de $R_1, \ldots, R_k$ nco i relevante na execugco do bloco;

- se k i natural, $g_0, \ldots, g_k$ sco termos booleanos e $R_0, \ldots, R_k$ sco regras, entco

$$R \equiv \textbf{ if } g_0 \textbf{ then } R_0 \textbf{ elseif } g_1 \textbf{ then } R_1 \ldots \textbf{ elseif } g_k \textbf{ then } R_k \textbf{ endif}$$

  i uma regra. O conjunto de atualizagues desta regra i dado por

$$Updates(R, S) = \begin{cases} Updates(R_i, S) & \text{se } g_i \wedge \forall j < i : \neg g_j \\ \emptyset & \text{se } \forall i : \neg g_i \end{cases}$$

Estas regras sco conhecidas como *regras basicas*. □

Estas regras sco as mais simples de ASM e permitem a especificagco de apenas alguns tipos de sistemas reais. Outros tipos de regras mais poderosas serco mostrados nas segues subseq—entes. Observe que **nco existe** composigco seq—encial de regras, isto i, uma regra da forma

$$R_1; R_2$$

onde primeiro executa-se R_1 e em seguida R_2. Isto porque um programa em ASM descreve somente um passo do algoritmo. A composigco seq—encial pode gerar estruturas de execugco muito complexas, o que pode tornar o raciocmnio sobre a especificagco muito mais difmcil [23].

Definição 3.9 (Disparo de uma Regra) O *disparo de uma regra* R em um estado S produz um novo estado S', tal que, se $Updates(R, S)$ for consistente, entco

$$Val_{S'}(f, \bar{x}) = \begin{cases} y & \text{se } ((f, \bar{x}), y) \in Updates(R, S) \\ Val_S(f, \bar{x}) & \text{caso contrario.} \end{cases}$$

Se o conjunto $Updates(R, S)$ nco for consistente, entco o efeito do disparo de R em S sera nulo, isto i, o estado S' resultante sera igual a S. Outra abordagem utilizada para tratar conjuntos de atualizagco inconsistentes i fazer uma escolha nco-determinista da atualizagco que sera disparada [28]. Definiremos $Fire(R, S)$ como o estado resultado do disparo de R em S. □

Definição 3.10 (Especificagco ASM) Uma *especificagco ASM* i uma tupla da forma $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$, onde

- $\Upsilon = \Upsilon_0 \cup \Upsilon_e$ i um vocabulario composto pela unico de um vocabulario pri-definido, Υ_0, e um vocabulario especificado pelo usuario, Υ_e. O vocabulario pri-definido Υ_0 contim os nomes de relagco *Boolean*, *Integer*, *String*, *List*, *Set*, que serco interpretados, respectivamente, como os universos dos valores lsgicos, dos nzmeros inteiros, das cadeias de caracteres, das listas e das partes de conjuntos. Υ_0 contim tambim as operagues usuais sobre estes conjuntos;

- $\mathcal{A}$ i um conjunto de Υ-algebras ou estados S, cada um consistindo no vocabulario Υ e um conjunto X (o superuniverso de S) de fungues, que sco as interpretagues em X dos nomes de fungues de zero argumento pertencentes a Υ. Um nome de fungco de aridade r, $r > 0$, i interpretado como uma fungco de $X^r \to X$. O conjunto X i comum a todos os estados pertencentes a $\mathcal{A}$;

- $S_0 \in \mathcal{A}$ i o estado inicial, onde a interpretagco de alguns nomes de fungues sco dados; nomes de fungues cujas interpretagues nco sco dadas em S_0 sco interpretados como *undef*;

- $\mathcal{P}$ i uma regra que descreve as modificagues das interpretagues de nomes de fungues de uma algebra (estado) para outra.

□

O vocabulario de uma ASM reflete apenas os recursos verdadeiramente invariantes de um algoritmo, em vez de detalhes de um estado em particular. O estado inicial S_0 pode ser definido atravis de qualquer formalismo existente, em particular, via regras de transigco do modelo ASM. Pode-se utilizar tambim mecanismos como lambda calculo, fungues recursivas, entre outros.

Definição 3.11 (Execugco de uma Especificagco ASM) A *execugco de uma especificagco ASM* $(\Upsilon, \mathcal{A}, S_0, \mathcal{P})$ i uma seq—jncia $\mathcal{S} = \langle S_n : n \geq 0 \rangle$ de estados pertencentes a $\mathcal{A}$, onde $S_{n+1} = \mathit{Fire}(\mathcal{P}, S_n)$, isto i, S_{n+1} i obtido a partir de S_n, disparando a regra $\mathcal{P}$ em S_n. □

Na maioria dos sistemas dinbmicos discretos, a execugco pode ser influenciada pelo ambiente. Intuitivamente, um ambiente ativo i um agente externo. Desta maneira, utilizamos *fungues externas* na modelagem do ambiente no qual o sistema esta inserido. Alim disso, utilizam-se fungues externas para:

- prover ocultamento de informagues – qualquer caractermstica do sistema, cujo funcionamento nco i relevante no entendimento da especificagco, pode ser modelada por meio de fungues externas;

- inserir nco-determinismo ao modelo – considerando que as agues do ambiente acontecem de maneira nco-determinista, utilizamos fungues externas para descrever implicitamente o nco-determinismo; na Segco 3.3 mostraremos uma construgco para especificarmos explicitamente o nco-determinismo.

3.2 Exemplo de Especificagco ASM

Nesta segco mostraremos um exemplo simples de especificagco formal em ASM. Apss apresentada a especificagco, mostraremos a sua corregco em relagco ao mundo real.

Exemplo 3.1 (Fungco Fatorial) Uma especificagco ASM para a fungco fatorial i $ASM_{fat} = (\Upsilon, \mathcal{A}, S_0, P_{fat})$, onde:

- $\Upsilon = \Upsilon_0 \cup \Upsilon_e$, onde Υ_0 i o conjunto dos smmbolos pri-definidos e $\Upsilon_e = \{fat, i\}$;
- $\mathcal{A}$ i um conjunto de estados;
- a interpretagco dos nomes de fungco pertencentes a Υ em $S_0 \in \mathcal{A}$ i dado pela fungco Val_{S_0}, tal que

$$\begin{array}{rcl} Val_{S_0}(i) & = & 0 \\ Val_{S_0}(fat) & = & \lambda n.n = 0 \to 1, undef \end{array}$$

- P_{fat} i dado pela regra:

  $fat(i+1) := (i+1) \times fat(i)$,
  $i := i + 1$

A execugco de ASM_{fat} i a seq—jncia $\mathcal{S}_{fat} = \langle S_0, S_1, \cdots \rangle$ de elementos de $\mathcal{A}$, onde cada $S_n = Fire(P_{fat}, S_{n-1})$, para $n > 0$. □

Proposição 3.1 (Corregco do Exemplo 3.1) Para todo $i \geq 0$, $fat(i) = i!$ no estado S_i. A demonstragco por indugco em i i a seguinte:

- para $i = 0$, $Val_{S_i}(i) = Val_{S_0}(i) = 0$ e

$$\begin{array}{rcl} Val_{S_i}(fat)(Val_{S_i}(i)) & = & Val_{S_0}(fat)(0) \\ & = & (\lambda n.n = 0 \to 1, undef)(0) = 1 = 0! = (Val_{S_0}(i))! \\ & = & (Val_{S_i}(i))! \end{array}$$

- supondo que, no estado S_i, $Val_{S_i}(fat)(Val_{S_i}(i)) = (Val_{S_i}(i))!$, temos que, no estado S_{i+1},

$$Val_{S_{i+1}}(i) = Val_{S_i}(i) + 1$$

  e

$$\begin{array}{rcl} Val_{S_{i+1}}(fat)(Val_{S_{i+1}}(i)) & = & (Val_{S_i}(i) + 1) \times Val_{S_i}(fat)(Val_{S_i}(i)) \\ & = & (Val_{S_i}(i) + 1) \times (Val_{S_i}(i))! \\ & = & (Val_{S_i}(i) + 1)! \\ & = & (Val_{S_{i+1}}(i))! \end{array}$$

C.Q.D.

3.3 Regras Nco-Basicas

Nesta segco apresentaremos as regras de transigco nco-basicas, que sco as regras que utilizam variaveis. A introdugco de variaveis ao modelo ASM da maior poder ‘s definigues, permitindo, por exemplo, importar elementos, o que permite estender universos, e especificar o nco-determinismo de sistemas. Para a regra *import* que definiremos a seguir, consideraremos que existe um universo de nome *Reserve*, contido no superuniverso do estado. Este universo contim os elementos que serco importados. Cada regra *import* retira um elemento de *Reserve*.

Definição 3.12 (Regra *Import*) A regra *import* i uma regra da forma

$$R \equiv \textbf{import } v \ R_0 \textbf{ endimport},$$

onde v i uma variavel e R_0 i uma regra. A sembntica desta regra i a seguinte: escolhe-se um elemento a do universo *Reserve* e associa-o ‘ variavel v. Desta forma, executa-se a regra R_0, no estado S_a, que i uma expansco do estado S, no qual interpretamos v como a. O conjunto de atualizagues para esta regra i dado por

$$\mathit{Updates}(R, S) = \{((\mathit{Reserve}, a), \mathit{false})\} \cup \mathit{Updates}(R_0, S_a).$$

□

O efeito desta regra i “criar”um novo elemento. Em geral i utilizada com uma regra da forma $U(v) := \mathit{true}$, onde U i um nome de universo, para inserir o novo elemento ao universo U, modelando uma expansco de U.

Definição 3.13 (Regra *Var*) A regra *var* i uma regra da forma

$$R \equiv \textbf{var } v \textbf{ ranges over } U \ R_0 \textbf{ endvar},$$

onde v i uma variavel, U i o nome de um universo finito e R_0 i uma regra. A sembntica desta regra i a seguinte: para cada elemento x de U, executamos a regra R_0 no estado S_x, que i uma expansco do estado S, tal que a variavel v i interpretada como x. Esta execugco cria o conjunto de atualizagues $\mathit{Updates}(R_0, S_x)$. O efeito da regra i a unico de todos os conjuntos $\mathit{Updates}(R_0, S_x)$, tal que $x \in U$, isto i,

$$\mathit{Updates}(R, S) = \bigcup_{x \in U} \mathit{Updates}(R_0, S_x).$$

□

O efeito desta regra i criar uma instbncia de R_0 para cada elemento pertencente ao universo U. Em cada instbncia de R_0, a variavel v esta associada ao elemento correspondente de U. Apss criadas as instbncias, todas sco executadas em paralelo. Esta regra i usada para modelar paralelismo smncrono, como foi mostrado no Exemplo 2.4.

Definição 3.14 (Regra *Choose*) A regra *choose* i uma regra da forma

$$\textbf{choose } v \textbf{ in } U \textbf{ satisfying } g \ R_0 \textbf{ endchoose},$$

onde v i uma variavel, U i o nome de um universo finito, g i um termo booleano e R_0 i uma regra. O efeito desta regra i executar a regra R_0 em um estado S_a, no qual a variavel v esta associada a um elemento $a \in U$. O elemento a i escolhido de maneira nco-determinista e torna a guarda g verdadeira. □

Algoritmos nco-deterministas sco zteis, por exemplo, como descrigues (especificagues) em alto nmvel de algoritmos "reais". Na Segco 3, modelamos o nco-determinismo implicitamente, deixando a escolha para o ambiente. Esta regra i usada para modelar explicitamente o nco-determinismo. Outra maneira de modelar o nco-determinismo i permitir que a inconsistjncia de conjuntos de atualizagco sejam resolvidas escolhendo-se nco-deterministicamente qual atualizagco disparar. Entretanto, esta abordagem pode dificultar a leitura de uma especificagco e a demonstragco de propriedades sobre ela.

4 Estudo de Caso: A Linguagem *Tiny*

Nesta segco apresentaremos um exemplo de definigco formal de sistemas em ASM. O problema que exploraremos aqui i a definigco formal de uma linguagem de programagco. Definiremos uma linguagem de programagco imperativa de pequeno porte, *Tiny*, contendo somente comandos e expressues. A descrigco informal e o estilo de definigco formal utilizado i baseado na definigco denotacional de *Tiny* encontrada em [19]. ASM ja foram utilizadas na definigco formal de varias linguagens de programagco de grande porte, entre elas, Java, C e Prolog. Uma listagem do que ja foi feito pode ser encontrada em [9].

4.1 Descrigco Informal da Linguagem *Tiny*

4.1.1 Sintaxe Informal de *Tiny*

A Linguagem *Tiny* possui dois tipos de construgues: expressues e comandos. Ambos podem conter identificadores (`x`, `y1`, `id`), que sco cadeias de caracteres contendo letras e dmgitos, comegando com uma letra, e nzmeros inteiros (1, 2, ...) e os valores lsgicos `true` e `false`. Utilizaremos meta-variaveis I, I_1, I_2, $I^{'}$, ... para identificadores, E, E_1, E_2, $E^{'}$, ... para expressues, e C, C_1, C_2, $C^{'}$, ... para comandos. A sintaxe abstrata de *Tiny* i:

$$
\begin{array}{lcl}
E & ::= & 0 \mid 1 \mid \mathit{true} \mid \mathit{false} \mid \mathit{read} \mid I \\
 & & \mid \mathit{not}\ E \mid E_1 = E_2 \mid E_1 + E_2 \mid (E) \\
C & ::= & I := E \mid \mathit{output}\ E \mid \mathit{if}\ E\ \mathit{then}\ C_1\ \mathit{else}\ C_2 \\
 & & \mid \mathit{while}\ E\ \mathit{do}\ C \mid C_1; C_2 \mid (C)
\end{array}
$$

Note que a gramatica dada i ambmgua, mas isto nco constitui problema, porque, nesta especificagco, questues relativas a analise sintatica nco sco tratadas. Desta forma, a sembntica i baseada em sintaxe abstrata. Utilizaremos parjnteses para evitar ambig—idades no texto.

4.1.2 Sembntica Informal de *Tiny*

Todo comando de *Tiny*, quando executado, modifica um estado. Este estado contim trjs elementos:

- a memsria – correspondjncia entre identificadores e valores. Na memsria, cada identificador esta associado a algum valor ou esta nco-associado (*unbound*);

- a entrada – fornecida pelo usuario antes do inmcio da execugco do programa; consiste em uma seq—jncia (possivelmente vazia) de valores que podem ser lidos utilizando a expressco *read*;
- a samda – i inicialmente uma seq—jncia vazia de valores que armazena os resultados do comando *output* E.

O valor de cada expressco i dado abaixo:

1. 0 ou 1 – o valor de 0 i 0 e o valor de 1 i 1;
2. *true* ou *false* – o valor de *true* i o valor lsgico *true* e o valor de *false* i o valor lsgico *false*;
3. *read* – o valor de *read* i o valor do prsximo elemento na entrada (ocorre um erro se a entrada for vazia); a expressco *read* possui o "efeito colateral"de remover o primeiro item da entrada, de modo que, apss a avaliagco da expressco, a entrada possui um elemento a menos;
4. I – o valor de uma expressco I i o valor associado a I na memsria (ocorre um erro se I esta nco-associado);
5. *not* E – se o valor de E for *true* entco o valor de *not* E sera *false*; se o valor de E for *false* entco o valor de *not* E sera *true*; em todos os outros casos, ocorre um erro;
6. $E_1 = E_2$ – o valor de $E_1 = E_2$ i *true* se os valores de E_1 e E_2 forem iguais; caso contrario, a expressco avalia em *false*; se E_1 ou E_2 nco forem nzmeros inteiros, ocorre um erro;
7. $E_1 + E_2$ – o valor de $E_1 + E_2$ i a soma numirica entre os valores de E_1 e E_2; se algum destes valores nco for um nzmero, ocorre um erro.

O efeito de cada comando i dado abaixo:

1. $I := E$ – o identificador I i associado, na memsria, ao valor da expressco E, sobrescrevendo o que estava anteriormente associado a I;
2. *output* E – o valor de E i colocado na samda;
3. *if* E *then* C_1 *else* C_2 – se o valor de E for *true*, C_1 i executado; se o valor de E for *false*, C_2 i executado; em todos os outros casos, ocorre um erro;
4. *while* E *do* C – se o valor de E for *true*, entco C i executado e, em seguida, *while* E *do* C i executado novamente no estado resultante da execugco de C; se o valor de E for *false*, entco nada i feito; se o valor de E nco for nem *true* nem *false*, ocorre um erro;
5. $C_1; C_2$ – os comandos C_1 e C_2 sco executados nesta ordem; C_2 i executado no estado resultante da execugco de C_1.

Expressues de *Tiny*	**Representagco**
0	[0]
1	[1]
false	[FALSE]
true	[TRUE]
read	[READ]
I	$[I]$
not E	[NOT, E]
$E_1 = E_2$	[EQUALS, E_1, E_2]
$E_1 + E_2$	[ADD, E_1, E_2]

Tabela 1: Tradugco de expresues de *Tiny* para a forma de lista

Comandos de *Tiny*	**Representagco**
$I := E$	[ASSIGN, I, E]
output E	[OUTPUT, E]
if E *then* C_1 *else* C_2	[IF, E, C_1, C_2]
while E *do* C	[WHILE, E, C]
$C_1; C_2$	$[C_1, C_2]$

Tabela 2: Tradugco de comandos de *Tiny* para a forma de lista

4.1.3 Um Programa em *Tiny*

O exemplo abaixo escreve na samda a soma dos nzmeros contidos na entrada. O fim da entrada i marcado com 0.

sum := 0;
x := *read*;
while not (*x* = 0) *do*
(
    sum := *sum* + *x*;
    x := *read*
);
output sum

4.2 Definigco Formal de *Tiny* em ASM

Nesta segco mostraremos uma definigco formal de *Tiny* utilizando a metodologia de definigco de ASM. A linguagem *Tiny* i descrita por $ASM_{tiny} = (\Upsilon, \mathcal{A}, S_0, \mathcal{P}_{tiny})$.

4.2.1 Convengco

A arvore sintatica abstrata de um programa em *Tiny* sera armazenada em uma lista. As regras serco definidas de forma a consultar sempre o primeiro elemento desta lista para decidir o que sera feito. As expressues de *Tiny* sco traduzidas

para a forma de lista da maneira mostrada na tabela 1. A tradugco de comandos i mostrada na tabela 2. Por simplicidade, consideraremos que uma seq—jncia de comandos da forma $C_1; \ldots; C_n$ sera armazenado como $[C_1, \ldots, C_n]$. Por exemplo, o programa da Segco 4.1.3 i armazenado na forma:

```
[
    [ASSIGN, "sum", [0]],
    [ASSIGN, "x", [READ]],
    [WHILE,
        [NOT, [EQUALS, ["x"], [0]]],
        [
            [ASSIGN, "sum", [ADD, ["sum"], ["x"]]],
            [ASSIGN, "x", [READ]]
        ]
    ],
    [OUTPUT, ["sum"]]
]
```

4.2.2 O Vocabulario Υ

O vocabulario de ASM_{tiny} contim os seguintes nomes de fungco:

- *program* – fungco de aridade 0 que contim um programa em *Tiny*;
- *memory* – fungco de aridade 1 que associa nomes de identificadores a valores;
- *input* – fungco externa de aridade 0 que contim a seq—jncia de valores de entrada;
- *output* – fungco de aridade 0 que contim a seq—jncia de valores de samda do programa;
- *opstack* – fungco de aridade 0 que simula uma pilha auxiliar de execugco.

Consideraremos tambim que sco pri-definidos em Υ os seguintes nomes de fungco ou relagco:

- os universos *Integer*, *Boolean*, *String* e *List*, simbolizando respectivamente os conjuntos dos nzmeros inteiros, dos valores lsgicos, das cadeias de caracteres e das listas;
- as operagues usuais sobre os nzmeros inteiros, os valores lsgicos e as cadeias de caracteres;
- as operagues *hd*, *tl*, *cons* e *concat* sobre as listas.

4.2.3 O Estado Inicial S_0

No estado inicial da especificagco ASM_{tiny}, definimos interpretagues iniciais de alguns nomes de fungco pertencentes a Υ. Na ASM_{tiny}, definiremos as seguintes interpretagues:

- o nome de fungco *program* i interpretado como um valor externo ‘ especificagco, de modo a tornar a especificagco independente de um programa em particular;
- *memory* – inicialmente, $Val_{S_0}(memory) = \lambda x.undef$, simbolizando que, inicialmente, nenhum identificador esta associado a valores;
- *input* i interpretada como um valor externo ‘ especificagco; o objetivo i tornar a especificagco independente da entrada de um programa;
- *output* e *opstack* sco interpretados inicialmente como a lista vazia ([]).

4.2.4 A Regra da Especificagco $\mathcal{P}_{tiny}$

Nesta segco mostraremos a regra que promove a mudanga de estados. A regra principal do programa i a seguinte:

if *program* $\neq$ [] **and not** *error* **then**
`...Passo da Interpretagco...`
endif

Esta regra da a seguinte sembntica operacional: executam-se “passos da interpretagco”, enquanto houver alguma coisa para ser interpretada ou nco ocorrer um erro. Um passo da interpretagco sera dado do seguinte modo:

- consulta-se o primeiro elemento da lista *program*;
- se este elemento for uma lista, promove-se a sua expansco, conforme definido abaixo;
- se for um identificador, ele i substitumdo por seu valor na memsria;
- se for alguma palavra reservada da interpretagco (NOT, EQUALS, READ, ADD, ASSIGN, OUTPUT, IF, WHILE), sco executadas as agues necessarias; tais agues podem alterar a pilha *opstack*;
- se for um nzmero inteiro ou um valor booleano, a pilha *opstack* i consultada para determinar a agco que sera executada.

O passo da interpretagco i formado por um bloco contendo as seguintes regras:

1. Regra para expansco de lista:

   if $List(hd(program))$ **then**
   $program := concat(hd(program), tl(program))$
   endif

2. Para a expressco de negagco lsgica:

if $hd(program) = \mathsf{NOT}$ **then**
    $program := tl(program)$,
    $opstack := cons(\mathsf{NOT}, opstack)$
endif

3. Para a expressco de igualdade, temos a seguinte regra:

if $hd(program) = \mathsf{EQUALS}$ **then**
    $program := tl(program)$,
    $opstack := cons(\mathsf{EXCHANGE}, cons(\mathsf{EQUALS}, opstack))$
endif

4. Para a expressco de adigco:

if $hd(program) = \mathsf{ADD}$ **then**
    $program := tl(program)$,
    $opstack := cons(\mathsf{EXCHANGE}, cons(\mathsf{ADD}, opstack))$
endif

5. Para a expressco identificador:

if $String(hd(program))$ **then**
    if $memory(hd(program)) = undef$ **then**
        $error := true$
    else
        $program := cons(memory(hd(program)), tl(program))$
    endif
endif

6. Para a expressco de leitura:

if $hd(program) = \mathsf{READ}$ **then**
    if $input = []$ **then**
        $error := true$
    else
        $program := cons(hd(input), tl(program))$,
        $input := tl(input)$
    endif
endif

7. Para o comando de atribuigco:

if $hd(program) = \mathsf{ASSIGN}$ **then**
    $program := tl(tl(program))$,
    $opstack := cons(\mathsf{ASSIGN}, cons(hd(tl(program)), opstack))$
endif

8. Para o comando de impressco:

   if $hd(program) = \mathsf{OUTPUT}$ **then**
   $program := tl(program)$,
   $opstack := cons(\mathsf{OUTPUT}, opstack)$
   endif

9. Para o comando de condicional:

   if $hd(program) = \mathsf{IF}$ **then**
   $program := tl(program)$,
   $opstack := cons(\mathsf{COND}, opstack)$
   endif

10. Para o comando de repetigco

    if $hd(program) = \mathsf{WHILE}$ **then**
    $opstack := cons(\mathsf{COND}, opstack)$,
    $program := cons(\, WhileExp(program), cons(\, WhileTrue(program),$
    $cons([], WhileCont(program))))$
    endif

    Para simplificar a escrita desta regra, utilizamos as seguintes fungues estaticas:

    $WhileExp(p) = hd(tl(p))$
    $WhileTrue(p) = cons(hd(tl(tl(p))), cons(\, WhileCom(program), []))$
    $WhileCom(p) = cons(\mathsf{WHILE}, cons(hd(tl(p)), cons(hd(tl(tl(p))), [])))$
    $WhileCont(p) = tl(tl(tl(p)))$

11. Para as expressues que sco valores literais (nzmeros inteiros e valores lsgicos), a pilha *opstack* i consultada e as agues necessarias sco executadas. A regra para isso i:

    if $Integer(hd(program))$ **or** $Boolean(hd(program))$ **then**
    if $hd(opstack) = \mathsf{OUTPUT}$ **then**
    $output := concat(output, [hd(program)])$,
    $program := tl(program)$,
    $opstack := tl(opstack)$
    if $hd(opstack) = \mathsf{ASSIGN}$ **then**
    $opstack := tl(tl(opstack))$,
    $memory(hd(tl(opstack))) := hd(program)$
    if $hd(opstack) = \mathsf{EXCHANGE}$ **then**
    $opstack := tl(opstack)$,
    $program := cons(hd(tl(program)), cons(hd(program), tl(tl(program))))$
    if $hd(opstack) = \mathsf{ADD}$ **then**
    $opstack := tl(tl(opstack))$,
    if not $Integer(hd(program))$ **or not** $Integer(hd(tl(program)))$ **then**
    $error := true$

```
            else
                opstack := tl(opstack)
                program := cons(hd(program) + hd(tl(program)), tl(tl(program)))
            endif
        if hd(opstack) = NOT then
            opstack := tl(tl(opstack)),
            if not Boolean(hd(program)) then
                error := true
            else
                program := cons(not hd(program), tl(program))
            endif
        if hd(opstack) = COND then
            opstack := tl(tl(opstack)),
            if not Boolean(hd(program)) then
                error := true
            elseif hd(program) then
                program := cons(hd(tl(program)), tl(tl(tl(program))))
            else
                program := tl(tl(program))
            endif
        if hd(opstack) = EQUALS then
            opstack := tl(tl(opstack)),
            if not Integer(hd(program)) or not Integer(hd(tl(program))) then
                error := true
            else
                opstack := tl(opstack)
                program := cons(hd(program) = hd(tl(program)), tl(tl(program)))
            endif
        endif
    endif
```

5 ASM Multiagentes

Uma *ASM Multiagente* $\mathcal{M}$, tambim conhecida como *ASM Distribumda*, contim um nzmero finito *agentes* computacionais que executam concorrentemente um nzmero finito de programas. A cada agente esta associado um programa. Formalmente, uma ASM Multiagente $\mathcal{M}$ i uma tupla $(\Upsilon_{\mathcal{M}}, \mathcal{A}, \mathcal{C}_0, \mathcal{P})$, tal que:

- $\Upsilon_{\mathcal{M}}$ i um vocabulario que contim os nomes de fungco que aparecem nas regras de $\mathcal{P}$, com excegco de *Self*. Pertencem a $\Upsilon_{\mathcal{M}}$ os nomes de fungues de zero argumento que representam os nomes de msdulos (elementos de $\mathcal{P}$) e uma fungco unaria, *Mod*, que representa a relagco entre agentes e msdulos. Um elemento a i um *agente* em um dado estado S se houver um nome de msdulo μ tal que $S \models Mod(a) = \mu$. Da mesma forma, o programa de a i $Prog(a) = \pi_\mu$.
- $\mathcal{A}$ i um conjunto de estados;

- $\mathcal{C}_0$ i uma colegco de estados denominados *estados iniciais* de $\mathcal{M}$; todos os elementos de $\mathcal{C}_0$ compartilham as mesmas interpretagues dos nomes de fungco de $\Upsilon_{\mathcal{M}}$, com excegco de *Self*, que i interpretado como o agente associado ao estado;

- $\mathcal{P}$ i um conjunto finito indexado de programas π_μ (os msdulos), identificados pelos nomes de msdulos $\mu \in \Upsilon$; cada programa pertencente a $\mathcal{P}$ i uma regra.

O nome de fungco especial de zero argumento *Self* permite a auto-identificagco de agentes: *Self* i interpretado como a por cada agente a e pode ser usada, por exemplo, para modelar algum estado local dos agentes. Dizemos que um agente a realiza uma *transigco* (*"move"*) em S se o conjunto de atualizagues

$$Updates(a, S) = Updates(Prog(a), View_a(S))$$

i disparado em S, isto i, se a regra que forma o programa do agente a for disparada no estado $View_a(S)$. Este estado i a expansco do estado S, no qual a variavel *Self* i interpretada como a. Sobre esta definigco de transigco, podemos definir diversos tipos de execugco. Abaixo, temos a definigco de *execugco parcialmente ordenada*, que i uma importante nogco de computagco distribumda. Esta definigco garante que nco havera inconsistjncias na execugco de 2 ou mais agentes, pois cada transigco i executada por um agente de maneira attmica.

Definição 5.1 (Execugco parcialmente ordenada) Uma execugco parcialmente ordenada ρ de uma ASM Multiagentes $\mathcal{M}$ i uma tripla (M, A, σ) que satisfaz as seguintes condigues:

1. M i um conjunto parcialmente ordenado, no qual todos os conjuntos de transigco $\{y : y \preceq x\}, x \in M$, sco finitos: os elementos de M representam transigues realizadas pelos varios agentes durante a execugco[3].

2. A i uma fungco sobre M tal que todo conjunto nco-vazio $\{x : A(x) = a\}$ i linearmente ordenado: $A(x)$ i o agente que realiza a transigco x (supomos que as transigues de um znico agente sejam linearmente ordenados).

3. σ i uma fungco que associa um estado de $\mathcal{M}$ a cada segmento inicial[4] de M, tal que $\sigma(\emptyset)$ i um estado inicial: para cada segmento inicial X de M, $\sigma(X)$ i o estado que resulta da realizagco de todos os movimentos em X.

4. A *condigco de coerjncia*: se x i um elemento maximal em um segmento inicial finito X de M e $Y = X - \{x\}$, entco $A(x)$ i um agente em $\sigma(Y)$ e $\sigma(X)$ i obtido a partir de $\sigma(Y)$ disparando $A(x)$ em $\sigma(Y)$.

□

Nos exemplos a seguir, utilizaremos esta nogco de execugco.

[3]Duas transigues x e y sco tais que $y \prec x$, se a transigco x iniciou apss o tirmino de y

[4]Um *segmento inicial* de um conjunto parcialmente ordenado P i uma sub-estrutura X de P tal que

$$x \in X \wedge y < x \Rightarrow y \in X.$$

5.1 Exemplos de ASM Multiagente

Nesta segco apresentaremos exemplos de utilizagco de ASM multiagentes para definigco de sistemas distribumdos. Os sistemas que especificaremos sco o problema do *Jantar dos Filssofos* e o protocolo *Alternating Bit*.

Exemplo 5.1 (Jantar dos Filssofos) Um conjunto de *filssofos* senta-se a uma mesa circular, e um garfo i posicionado entre cada dois filssofos. Cada um alterna seu estado entre pensando ("*thinking*"), com fome ("*hungry*") e comendo ("*eating*"). Para entrar no estado "eating", um filssofo deve estar com os dois garfos adjacentes a ele em seu poder. Cada garfo ss pode estar servindo a um filssofo de cada vez. O csdigo apresentado a seguir corresponde ao programa de um agente ASM. Cada agente representa um filssofo distinto e nenhuma suposigco i estabelecida sobre a velocidade relativa da execugco do csdigo de cada um. Suponha que p seja uma fungco equivalente a *Self* citada anteriormente, associada a instbncias do universo *Philosophers*. A fungco $Status(p)$ representa o estado atual do filssofo p, a fungco estatica $Next(p)$ representa o filssofo imediatamente ‘ esquerda de p e $Holder(p)$ indica qual i o filssofo que, num dado momento, possui o garfo imediatamente ‘ direita do filssofo p. Observe que $Holder(p) = undef$ indica que o garfo nco esta sendo usado.

if $Status(p) =$ "*thinking*" **then**
  $Status(p) :=$ "*hungry*"
endif
if $Status(p) =$ "*hungry*" **and** $Holder(LeftFork(p)) = undef$
  and $Holder(RightFork(p)) = undef$ **then**
  $Holder(LeftFork(p)) := p$,
  $Holder(RightFork(p)) := p$,
  $Status(p) :=$ "*eating*"
endif
if $Status(p) =$ "*eating*" **then**
  $Holder(LeftFork(p)) := undef$,
  $Holder(RightFork(p)) := undef$,
  $Status(p) :=$ "*thinking*"
endif

A solugco apresentada i isenta de *deadlock*, uma vez que os dois garfos sco requisitados atomicamente. A inicializagco dos valores das fungues i apresentada a seguir. Observe que todas as atualizagues sco executadas em paralelo, diferentemente do csdigo acima, onde cada instbncia representa um agente diferente.

var q **ranges over** *Philosophers*
  $Status(q) :=$ "*thinking*",
  $Holder(LeftFork(q)) := undef$
  $Holder(RightFork(q)) := undef$
endvar

□

Exemplo 5.2 (*Alternating Bit Protocol*) Um agente i designado por *Sender* e envia mensagens a outro agente, designado por *Receiver*, atravis de um meio onde as mensagens podem ser perdidas ou duplicadas, mas a ordem de envio i preservada. Para identificar as mensagens corretas, um bit adicional i enviado, alternando 0 e 1 a cada transmissco. Uma especificagco do csdigo dos agentes *Sender* e *Receiver* i apresentada a seguir. **Agente *Sender*:**

if $Clock > Slast + timeout$ **then**
$SRmessage := Smessage$,
$SRbit := Sbit$,
$Slast := Clock$
endif
if $RSbit = Sbit$ **then**
$Scont := Scont + 1$,
$Smessage := InputFile(Scont + 1)$,
$Sbit := \neg Sbit$
endif

Agente *Receiver*:

if $Clock > Rlast + timeout$ **then**
$RSbit := Rbit$,
$Rlast := Clock$
endif
if $SRmessage \neq undef$ **and** $SRbit \neq Rbit$ **then**
$Rcont := Rcont + 1$,
$OutputFile(Rcont + 1) := SRmessage$,
$Rbit := \neg Rbit$
endif

A fungco externa *Clock* i utilizada para representar tempo transcorrido. O envio de mensagens pode ser repetido apss um intervalo determinado pela fungco estatica *timeout*. I razoavel supor que as duas referjncias ‘ fungco *Clock*, em cada agente, retornem o mesmo valor em um mesmo passo de execugco. As mensagens sco geradas em *Sender* pela fungco *InputFile* e armazenadas em *Receiver*, na fungco *OutputFile*. As fungues *Smessage* e *Sbit* armazenam o valor da mensagem e do bit atuais em *Sender*, enquanto *Rbit* i o valor do bit em *Receiver*. As fungues *SRmessage* e *SRbit* representam os dados em trbnsito de *Sender* para *Receiver*, enquanto *RSbit* representa o bit enviado pelo canal de comunicagco na diregco inversa. A fungco $\neg$ i utilizada para gerar os valores alternados para os bits. Alim dos dois agentes apresentados acima, a especificagco conta ainda com os agentes *SRLose* e *RSLose*, que simulam a perda ocasional de mensagens nos canais de transmissco.

Agente *SRLose*:

$SRbit := undef$,
$SRmessage := undef$

Agente *RSLose*:

$RSbit := undef$

Esses agentes eliminam as informagues armazenadas nos “canais” de comunicagco. Como nenhuma suposigco i estabelecida sobre a velocidade relativa de execugco dos

agentes, a perda de mensagens acontece aleatoriamente. Para o correto funcionamento do protocolo, as fungues *Sbit*, *Rbit*, *Scont*, *Rcont*, *Slast* e *Rlast* devem ser todas inicializadas com o valor zero. □

6 Avaliagco da Metodologia

Nesta segco apresentaremos uma avaliagco pragmatica da metodologia, levando em consideragco os seguintes aspectos: precisco, demonstragco de corregco da especificagco, generalidade, facilidade de apredizado, facilidade de leitura e de escrita, escalabilidade e possibilidade de execugco.

6.1 Precisco

Toda especificagco deve descrever, de forma precisa, o sistema real correspondente. Pode-se utilizar a metodologia de especificagco para descrever um sistema via uma sintaxe particular e uma sembntica associada claras e bem definidas. Se a sembntica da metodologia de especificagco nco i clara, descrigues que utilizam a metodologia podem nco ser mais claras do que o sistema original que esta sendo descrito. ASM utiliza estruturas matematicas classicas (conjuntos, sistemas de transigco), que sco modelos precisos e bem conhecidos, para descrever estados da computagco [21, 22].

6.2 Demonstragco de Corregco da Especificagco

Diversos aspectos da metodologia tornam facil demonstrar propriedades da especificagco ou a sua corregco em relagco ao sistema real especificado em ASM. Uma ticnica utilizada para especificar sistemas i a ticnica de refinamento passo a passo, sugerido em [22] e utilizado em [24, 27]. Esta ticnica permite que o projetista especifique um modelo do sistema em alto nmvel e detalhe passo a passo os diversos objetos e operagues do sistema, como na metodologia *top-down*. Em [8], esta ticnica foi aliada ‘ verificagco formal. A verificagco consistiu em provar que o modelo em mais alto nmvel (o “*ground model*”) estava correto em relagco ao mundo real. Em seguida, para cada passo do refinamento, provou-se que a introdugco de novos detalhes preservava a corregco com relagco ao modelo imediatamente anterior. A demonstragco de propriedades do sistema de transigco tambim i simples, pelos seguintes motivos:

- a transigco i sbvia, visto que nco ha nogco de fluxo de controle como nas linguagens de programagco, onde as possmveis seq—jncias de execugco sco dinbmica e imprevismveis;
- nco ha efeitos colaterais na execugco de regras, o que significa que, com uma simples inspegco das regras, determina-se o conjunto de atualizagues.

Alim disso, podemos verificar informalmente a corregco da especificagco, por inspegco das regras, como sugerido em [22], ou executando-se a especificagco, como veremos na Segco 6.7.

6.3 Generalidade

A metodologia ASM i ztil em uma grande variedade de dommnios: sistemas seq—enciais [10, 35]; sistemas paralelos e distribumdos [1, 8]; sistemas de tempo real [18, 26]. Em [9], ha muitos outros exemplos de aplicagues de ASM em diversos tipos de sistemas dinbmicos discretos. Muitas metodologias de especificagco sco, na pratica, zteis apenas em dommnios particulares. Por exemplo, a Sembntica Denotacional i aplicavel a sistemas seq—enciais, como algumas linguagens de programagco. Entretanto, a sua adaptagco para computagco distribumda i problematica. A extensco da metodologia ASM para computagco distribumda i natural, como pode ser visto em [22].

6.4 Facilidade de Aprendizado

ASM utiliza somente notagco matematica simples e bem conhecida e uma sintaxe semelhante ‘ de linguagens de programagco imperativa populares. Pode-se trabalhar com a metodologia, utilizando qualquer conhecimento ou ticnica existente, evitando a camisa de forga dos mitodos formais. Pode-se construir um sistema real seguindo um caminho natural de explicagco, utilizando notagco matematica padrco, de maneira que o modelo ASM resultante torna-se simples, transparente e facil de ler, entender e manipular. Isto refuta, para a metodologia ASM, uma objegco que freq—entemente i feita contra a utilizagco de mitodos formais para sistemas grandes e complexos. Diz-se que o programador midio nco tem base matematica suficiente para ser capaz de aplicar mitodos formais. No caso de ASM, apenas experijncia com algoritmos i suficiente para desenvolver ou compreender as especificagues e as provas.

6.5 Facilidade de Leitura e de Escrita

A leitura e a escrita de toda especificagco devem ser faceis. Se a notagco i difmcil de ler e escrever, poucas pessoas irco utiliza-la. Programas em ASM utilizam uma sintaxe bastante simples, na forma de pseudocsdigo, que pode ser lida inclusive por novatos. O znico conhecimento necessario para compreender uma especificagco ASM i o entendimento de programagco. Outros mitodos de especificagco, em especial a Sembntica Denotacional [19], utilizam conceitos menos usuais, que podem tornar a leitura e a escrita tarefas muito complicadas. Qualquer programador i capaz de ler e escrever uma especificagco ASM, pois os conceitos utilizados sco simples. Ao contrario de Sembntica Denotacional, que utiliza uma meta-linguagem baseado no lambda-calculo, ou da Sembntica Operacional Estruturada [32], que utiliza conceitos de programagco em lsgica, especificagues em ASM utilizam uma linguagem na forma de pseudo-csdigo semelhante ‘s linguagens de programagco do paradigma imperativo, que i mais comum ao programador midio. ASM permite o uso de termos e conceitos do dommnio do problema, com um mmnimo de codificagco notacional. Muitos mitodos populares de especificagco formal requerem uma grande quantidade de codificagco notacional, o que pode tornar a tarefa de especificagco muito difmcil.

6.6 Escalabilidade

Muitos mitodos formais tradicionais nco sco escalaveis. Eles funcionam muito bem para exemplos pequenos, freq—entemente inventados para ilustrar o mitodo. Entretanto, quando utilizados em grandes sistemas reais, muitos deles caem em uma explosco combinatorial ou simplesmente falham. A comunidade ASM trabalha na definigco de linguagens do mundo real: em [3, 4, 5], Egon Bvrger explica toda a linguagem Prolog e, em [24], i apresentada a definigco da linguagem C. O uso de ASM permite lidar com a complexidade de sistemas reais, atravis da construgco de hierarquias de nmveis de sistemas. Com mzltiplas camadas, pode-se facilmente examinar caractermsticas particulares do sistema ignorando outras. Como ja discutimos na Segco 6.2, a demonstragco de propriedades do sistema tambim i mais simples se utilizarmos esta ticnica. Em [6], Bvrger argumenta que a metodologia i escalavel para especificagco de sistemas integrados de *hardware/software*.

6.7 Possibilidade de Execugco

Alim da inspegco direta, outra maneira de testar a corregco de uma especificagco i executa-la diretamente. Mitodos como VDM [30] ou Z [29, 34] nco sco diretamente executaveis. As principais vantagens em executar uma especificagco sco:

1. permite "experimentar"a especificagco antes que o algoritmo seja implementado ou quando nco ha implementagues do algoritmo disponmveis;

2. varias condigues do ambiente externo podem ser representadas por parbmetros na especificagco, o que significa que ha mais oportunidades de insergco de situagues de falha;

3. permite combinar teste com verificagco formal.

7 Consideragues Finais

Neste tutorial mostramos como as ASM permitem especificar, de maneira simples, a sembntica operacional de linguagens de programagco. Esperamos que o leitor seja capaz de experimentar o mitodo com qualquer linguagem de seu interesse. O leitor tambim i convidado a experimentar o mitodo para especificar formalmente qualquer sistema computacional que desejar. Qualquer que seja o dommnio do problema, certamente as ASM constituem um vemculo interessante para especificagco e raciocmnio a respeito da parte dinbmica das solugues para este problema. Certamente, para os leitores que desejarem escrever especificagues maiores, o artigo [16], de Bvrger, i uma leitura obrigatsria. Neste artigo, i descrita uma metodologia para escrever especificagues ASM complexas, alim de serem feitas algumas comparagues com outros mitodos. Por se tratarem de um mitodo relativamente recente, as ASM oferecem uma area de pesquisa extremamente rica. Podemos citar algumas das principais frentes de pesquisa:

- Desenvolvimento de ferramentas – alguns interpretadores para ASM ja foram desenvolvidos. Entre eles, destacamos o interpretador da Universidade de

Michigan e o ambiente da Universidade de Paderborn. Ambos executam no sistema Unix, e o segundo possui uma interface grafica baseada em Tcl/Tk. Aqui no Brasil, na UFMG, estamos em fase de desenvolvimento de um ambiente baseado em Java; a primeira versco estara disponmvel ati o final de 1999.

- Padronizagco da linguagem – em fungco da necessidade de produgco de ferramentas para auxiliar o projetista na especificagco de sistemas, a comunidade ASM esta empenhada em produzir uma padronizagco da linguagem de especificagco ASM, denominada ASM-SL, com o objetivo de facilitar a integragco de diferentes ferramentas.

- Integragco com provadores de teoremas – o objetivo i produzir provas mecanizadas, evitando erros na demonstragco de propriedades do sistema. Existem trabalhos envolvendo *model checking* [37], PVS [15] e KIV [33].

- Introdugco de mecanismos de modularizagco e tipagem – este aspecto tambim esta relacionado com a necessidade de ferramentas para auxiliar na especificagco de grandes sistemas. As diversas extensues propostas ao modelo original podem ser obtidas a partir da pagina de Michigan.

Finalmente, gostarmamos de ressaltar que todo material existente sobre ASM esta disponmvel a partir das paginas *http://www.eecs.umich.edu/gasm* (da Universidade de Michigan) e *www.uni-paderborn.de/cs/asm.html* (da Universidade de Paderborn).

Referências

[1] D. Bèauquier and A. Slissenko. On Semantics of Algorithms with Continuous Time. Technical Report 97-15, Dept. of Informatics, Université Paris-12, October 1997.

[2] D. Bèauquier and A. Slissenko. The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE*, volume 1214 of *LNCS*, pages 201–212. Springer, 1997.

[3] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36–64. Springer, 1990.

[4] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer, 1990.

[5] E. Börger. A Logical Operational Semantics for Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output. In Y. Moschovakis, editor, *Logic From Computer Science*, volume 21 of *Berkeley Mathematical Sciences Research Institute Publications*, pages 17–50. Springer, 1992.

[6] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.

[7] E. Börger and U. Glässer. Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger, editors, *Evolving Algebras – Mini-Course, BRICS Technical Report (BRICS-NS-95-4)*, pages 128–153. University of Aarhus, Denmark, July 1995.

[8] E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[9] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998.

[10] E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1997.

[11] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[12] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim and J. Gruska and J. Zlatuska, editor, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic*, number 1450 in LNCS. Springer, August 1998.

[13] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[14] G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra Abstract Machine. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 191–214. Springer, 1996.

[15] A. Dold, T. Gaul, V. Vialard, and W. Zimmerman. ASM-Based Mechanized Verification of Compiler Back-Ends. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.

[16] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and Stephan and Traverso and Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS. Springer-Verlag, 1999.

[17] A. Gill. *Applied Algebra for the Computer Sciences.* Prentice Hall, Englewood Cliffs, 1976.

[18] U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[19] M J C Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer-Verlag, 1979.

[20] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[21] Y. Gurevich. Evolving Algebras. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 423–427, Elsevier, Amsterdam, the Netherlands, 1994.

[22] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[23] Y. Gurevich. The Sequential ASM Thesis. Technical Report MSR-TR-99-09, Microsoft Research, February 1999.

[24] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[25] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.

[26] Y. Gurevich and R. Mani. Group Membership Protocol: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 295–328. Oxford University Press, 1995.

[27] J. Huggins. Kermit: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 247–293. Oxford University Press, 1995.

[28] J. Huggins and R. Mani. Evolving Algebras Interpreter v. 2.0. Technical report, 1992.

[29] D. C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Oxford University Press, 1988. ISBN 0-19-859667-7.

[30] C. B. Jones. *Systematic Software Development using VDM (second edition).* Prentice Hall, 1990.

[31] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.

[32] F. Nielson and H. R. Neilson. *Semantics with Applications - A Formal Introduction*. John Wiley and Sons, New York, N.Y., 1992.

[33] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[34] J. M. Spivey. *The Z Notation*. Prentice Hall, New York, 1989.

[35] K. Stroetmann. The Constrained Shortest Path Problem: A Case Study In Using ASMs. *Journal of Universal Computer Science*, 3(4):304–319, 1997.

[36] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[37] K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.