Vale4 - Uma Linguagem Didática para Ensino de Programação Concorrente

Simão Sirineo Toscani

Faculdade de Informática – PUCRS
Departamento de Computação – UNILASALLE
stoscani@inf.pucrs.br

Resumo. *O artigo apresenta a linguagem Vale4 (V4), destinada ao ensino de programação concorrente. V4 permite praticar com os principais mecanismos de sincronização e comunicação, pode ser instalada em praticamente qualquer máquina e oferece boas facilidades de depuração. Os recursos oferecidos vão desde mecanismos básicos como lock-unlock e block-wakeup(p), até construções de alto nível como tasks com rendezvous. O compilador V4 gera código para uma máquina virtual, a qual implementa por hardware (virtual) um conjunto apropriado de instruções e um kernel de sistema operacional.*

Abstract. *This paper presents the Vale4 (V4) language, conceived to be a concurrent programming educational tool, which allows the practise with the main synchronizing and communicating tools. The language can be ported to practically any microcomputer and offers good debugging facilities. The programming resources go from basic ones, like lock/unlock and block/wakeup(p), to high level constructions like tasks with rendezvous. The V4 compiler generates code for a virtual machine, which implements a well suited set of instructions and an operating system kernel.*

1. Introdução

O estudo da programação concorrente costuma figurar na ementa das disciplinas de Sistemas Operacionais por dois motivos. Primeiro, porque a programação concorrente surgiu "dentro" dos Sistemas Operacionais, como uma técnica para construção desses sistemas. Segundo, porque os mecanismos de programação concorrente são normalmente implementados pelo próprio Sistema Operacional, no seu *kernel* [Toscani, Oliveira e Carissimi 2003]. Como o estudo deste tema é um dos mais difíceis de um curso de Computação, seria interessante (para professores e alunos) que houvesse uma linguagem adequada para seu ensino. O ideal seria que a linguagem permitisse praticar com todas as ferramentas de sincronização e comunicação existentes, as quais vão desde operações básicas do tipo *lock*/*unlock* e *block*/*wakeup*, para sincronização em baixo nível, até monitores e *tasks*, para sincronização em alto nível. Infelizmente, nenhuma linguagem de programação disponível oferece este leque de ferramentas ao seu usuário. Por exemplo, a linguagem SR [Andrews, 1991], que tem muito a ver com ensino de concorrência, não implementa o conceito de monitor. A linguagem Ada [Taft 1997], embora muito poderosa, se limita aos conceitos de *task* e *rendezvous.* Na linguagem Java fica "tudo escondido" e, segundo Brinch Hansen, parece que seus projetistas ignoraram os resultados das pesquisas em programação concorrente [Brinch Hansen 1999]. As extensões da linguagem C tampouco são satisfatórias; a biblioteca *Pthreads*, por exemplo, deixa muito a desejar quanto ao nível e à segurança das construções oferecidas [Toscani 2003].

Este artigo apresenta o ambiente de programação Vale4 (V4), que visa preencher a lacuna referida no parágrafo anterior. A linguagem V4 é *Pascal-like* e implementa um leque completo de ferramentas, com boas facilidades de depuração. Basicamente, o ambiente é implementado por um compilador, que gera código para uma máquina virtual, e por um simulador dessa máquina. Todo o sistema foi programado em SWI-Prolog [Wielemaker 1997] e, como tal, pode ser portado facilmente para diversas plataformas. [1] Não há preocupação com aspectos de eficiência, visto que a linguagem é destinada ao uso acadêmico (não é uma linguagem para produção de software).

O objetivo deste artigo é apresentar a linguagem V4 para a comunidade acadêmica. No seu início, o artigo apresenta as principais versões da linguagem. A seguir apresenta, nesta ordem, a sintaxe e os comandos da linguagem, a arquitetura da máquina virtual, o código gerado pelo compilador (arquivo de saída) e dois exemplos ilustrativos. No final, são apresentadas as conclusões, sendo citadas algumas possíveis extensões para o ambiente de programação.

2. As versões da linguagem

A linguagem V4 oferece uma grande variedade de mecanismos, os quais podem ser utilizados individualmente ou em conjunto, em um mesmo programa. Didaticamente, é conveniente vê-la como se oferecesse 6 versões de uso, conforme é resumido a seguir.

Versão básica, V4$_0$

Um programa V4 básico é um conjunto de *variáveis globais*, um conjunto de *procedimentos globais* e um conjunto de *processos*. Não há mecanismo de sincronização. Como tal, os problemas de sincronização e comunicação têm que ser resolvidos manualmente. Esta versão é útil para alertar o aluno sobre o problema das condições de corrida (*race conditions*) e ensinar os algoritmos clássicos de exclusão mútua.

Versão básica++, V4$_1$

É a versão anterior aumentada com operações *mutexbegin*/*mutexend* e *block*/*wakeup(p)* [Holt 1978], mais o tipo *queue* e suas operações *insert(p,Q)*, *insert(p,Q,prior)*, *age(Q)*, *empty(Q)* e *first(Q)*, onde *p* e *prior* são inteiros e *Q* é uma variável tipo *queue*. Os mecanismos desta versão permitem caracterizar muito bem as sincronizações elementares, voltadas para compartilhamento de recursos e para comunicação.

Versão com semáforos, V4$_2$

É a versão básica aumentada com *semáforos*. Nessa versão, um programa é formado por um conjunto de *variáveis globais*, incluindo variáveis *semáforas* [Dijkstra 1968], um conjunto de procedimentos e um conjunto de *processos*. Os processos se sincronizam, comunicam e formam filas através das operações P e V.

Versão com operações *send* e *receive*, V4$_3$

É a versão básica desprovida de variáveis globais e de procedimentos globais, mas aumentada com um mecanismo para troca de mensagens. A comunicação pode ser síncrona ou assíncrona. Nesta versão deixa de haver memória compartilhada entre os processos e cada processo passa a ter uma caixa postal para comunicação com os demais.

---

[1] O sistema SWI-Prolog foi desenvolvido na Universidade de Amsterdam, é de distribuição gratuita e está disponível para diversas plataformas (Linux, Windows, Solaris, etc.).

Versão com monitores, V4[4]

Nesta versão, um programa é formado por procedimentos globais, um conjunto de *monitores* e um conjunto de processos. Os monitores são especificados usando a sintaxe definida por Hoare [Hoare 1974] e permitem o uso de variáveis *condition* e *priority condition* [Holt 1978], sobre as quais podem ser executadas operações *wait* e *signal*.

Versão *multithread* com *tasks* e *rendezvous*, V4[5]

Nesta versão, um programa é formado por um conjunto de *tasks* que são especificadas através de uma sintaxe semelhante à usada na linguagem Ada [Taft 1997]. A comunicação é realizada por *rendezvous* em procedimentos do tipo *entry*. Toda *task* tem uma *thread*, denominada *main*, que pode criar novas *threads* através de comandos *fork*. Qualquer *thread* pode "aceitar" (através de comandos *accept*) chamadas de *entries* realizadas por outras *tasks*, originando esperas condicionais de múltiplas *threads*.

3. A sintaxe da linguagem V4

A sintaxe é especificada de maneira informal nas subseções que seguem.

3.1. Formato de um programa

Um programa V4 tem o seguinte formato:

V4prog D ; B endprog

onde:

- *D* é uma *lista de declarações* separadas por ";". Os componentes desta lista podem ser declarações de *variáveis*, declarações de *procedimentos* e/ou declarações de *monitores*.
- *B* é uma *lista de blocos* separados por ";". Os componentes da lista *B* podem ser *processos* e/ou *tasks*.

3.2. Unidades léxicas e comentários

As unidades léxicas (*tokens*) são as "palavras" que o usuário utiliza para construir programas A lista destas palavras permite antever as declarações e comandos oferecidos pela linguagem. Os tokens V4 são os seguintes.

Delimitadores: ( ) [ ] { } : . , ;

Operadores: *Aritméticos*: := + - * / mod
Lógicos: and or not (que também podem ser escritos & | ¬)
Relacionais: > = < >= <= <>

Identificadores: São formados por caracteres alfanuméricos, sendo o primeiro alfabético.

Palavras reservadas:

V4prog, integer, boolean, semaphore, init, initial, queue, array, procedure, returns, process, inline, read, write, nl, tab, while, do, forever, loop, endloop, exit, when, if, then, else, nothing, and, or, not, mod, mutexbegin, mutexend, lock, unlock, block, wakeup, P, V, yield, hold, monitor, endmonitor, condition, priority, initially, wait, signal, empty, count, first, insert, age, send, receive, task, endtask, entry, thread,

main, is, accept, fork, join, quit, new, myself, myId, getId, random, clockTime, debug1, debug2, nodebug, exec, pause, end, endprog.

Comentários: Podem ser introduzidos em qualquer lugar do programa, da maneira usual, isto é, delimitados por /* e */ ou iniciando por %, caso em que todo o resto da linha é considerado comentário.

3.3. Declarações de variáveis

Sendo *Id* um identificador, *k*, *m* e *n* constantes inteiras e *T* um elemento do conjunto {*integer*, *boolean*, *queue*, *semaphore*, *condition*}, as declarações válidas são:

- Id : T init *k* ;
- Id : array [*n*] of T init *k* ;
- Id : array [*m*, *n*] of T init *k* ;

A cláusula *init*, que também pode ser escrita *initial*, é opcional. No caso de ser usada na declaração de uma matriz (ou vetor), todos os elementos recebem o mesmo (único) valor, especificado por *k*. Para o tipo *boolean*, o valor inicial deve ser *true* ou *false*. Os tipos *queue* e *condition* não admitem inicialização. Ao invés de um único identificador *Id*, pode-se especificar uma lista de identificadores separados por ",". Neste caso, se for especificado um valor inicial, esse valor se aplicará a todos os identificadores da lista.

3.4. Procedimentos, processos, monitores, *tasks* e blocos *inline*

A figura 1 mostra os formatos dos "grandes componentes" da linguagem. Nessa figura, *Id* representa um identificador, *D* é uma lista de declarações de variáveis separadas por ";" (tal como especificado na seção anterior), *Dp* é uma lista de procedimentos separados por ";", *T* é o tipo *integer* ou *boolean* e *C* é um comando. Algumas observações são feitas a seguir.

Procedimento:	procedure Id (D) returns T ; D ; C
Processo simples:	process Id ; D ; C
Array de processos:	process Id (Var := k_1 to k_2) ; D ; C
Modelo de processo:	process type Id (D) ; D ; C
Monitor:	monitor Id ; D ; Dp ; initially C endmonitor
Task:	task Id is D ; Dp ; thread main is C endtask
Bloco inline:	inline Id ; Li .

Figura 1 - Componentes de um programa V4

A cláusula *returns* é utilizada apenas por *procedures* do tipo função. Os processos podem criar *threads* dinamicamente através de comandos *fork*. Em um *array* de processos, *Var* é uma variável inteira, k_1 e k_2 são constantes inteiras: são criadas k_2-k_1+1 exemplares (instâncias, clones) desse processo; para cada exemplar, a variável local *Var* é inicializada com um valor distinto entre k_1 e k_2 (todas as instâncias são criadas estaticamente, antes do início da execução do programa). Em um modelo de processo, as instâncias desse modelo são criadas dinamicamente, através de comandos *new* (vale observar que a sintaxe é a mesma de um procedimento tipo subrotina, porém trocando a especificação *procedure* por *process type*).

Em um monitor, os procedimentos de *Dp* são executados de forma mutuamente exclusiva, quando chamados pelos processos. A lista de variáveis *D* pode incluir os tipos *condition* e *priority condition*, e os procedimentos de *Dp* podem incluir as operações *wait* e *signal*. A cláusula *initially C* é opcional.

Em uma *task*, *Dp* pode incluir procedimentos do tipo *entry*. Um procedimento tipo *entry* tem apenas o seu cabeçalho declarado (usando a mesma sintaxe de um cabeçalho de procedimento, mas trocando a palavra "procedure" por "entry"). O corpo d*æntry* vai ser especificado no comando *accept* que referir essa *entry*. Isto significa que este tipo de procedimento pode executar diferentes códigos em função do ponto em que o *accept* é executado. A semântica do *accept* receberá atenção especial neste artigo.

Em um bloco *inline*, *Li* é uma lista de instruções de máquina, no formato do código que é gerado pelo compilador. Este bloco permite a introdução de código de máquina em qualquer lugar do programa fonte.

4. Os comandos da linguagem V4

A lista de comandos V4 é mostrada na figura 2. Nessa figura, *C* é qualquer um dos comandos lá contidos, *Var* é uma variável (pode ser um elemento de *array*), *E* é uma expressão (aritmética ou lógica), E_A é uma expressão aritmética, E_L é uma expressão lógica, *K* é uma constante ou variável inteira, *id* é um identificador de processo ou de *thread*, *Q* é uma variável do tipo *queue*, *Sem* é uma variável do tipo semáforo, *Time* é uma constante ou variável inteira, *Cond* é uma variável do tipo *condition* e *Prior* é uma constante ou variável inteira. Nos comandos *send/receive*, *ProcessId* é uma identificação de processo (pode ser um nome ou um número único de processo) e *Msg* é uma mensagem. No comando *new*, *Pname* é um nome de *template* de processo. No comando *accept*, *Et* é uma *entry* declarada na *task* em que o comando *accept* é executado. Além dos comandos da figura 2, a linguagem oferece as seguintes funções pré-definidas: *empty(Q)*, *count(Q)*, *myself*, *getId(ProcessName)*, *random(K)* e *clockTime*. Esta seção se restringe a comentar o comando *accept* e as facilidades de depuração.

Para explicar o funcionamento do comando *accept*, é necessário explicar primeiramente o que se entende por *rendezvous*. Um *rendezvous* (ou encontro) envolve duas *threads*. Uma, denominada *caller*, que executa um *call* para uma *entry X*. Outra, denominada *acceptor*, que executa um *accept* para essa mesma *entry X*. A primeira das *threads* que chega ao seu comando (*call* ou *accept*) se bloqueia até que a segunda *thread* chegue ao comando complementar (*accept* ou *call*) correspondente. Quando a segunda *thread* chega ao comando complementar, tudo está pronto para o *rendezvous*. A partir daí o par *caller-acceptor* está compromissado ("amarrado") para o *rendezvous*. A condição E_L da cláusula *when* é então avaliada. Se é verdadeira (ou se está ausente), o comando *C* especificado no *accept* é executado, usando os argumentos passados na chamada. Essa execução é feita de forma mutuamente exclusiva em relação aos demais *rendezvous* da *task*. Após a consumação do *rendezvous* (isto é, após a execução do comando *C*), os argumentos de retorno são passados para o *caller* e as duas *threads* prosseguem suas execuções de forma independente. Se a condição da cláusula *when* não é verdadeira, o *rendezvous* fica para mais tarde e a UCP passa para outra *thread* da *ready list*. O par que teve a condição de *rendezvous* falsa é colocado em uma fila de espera onde fica até que a condição tenha chance de ser verdadeira. Toda *task* tem duas filas de espera, formadas por pares *caller-acceptor* compromissados. Uma (fila 1) é a fila dos pares que esperam por

exclusão mútua (usada para escalonar os *rendezvous*), a outra (fila 2) encadeia os pares que estão à espera de reavaliação das condições para seus *rendezvous*. Sempre que acaba um *rendezvous* na *task*, os pares da fila 2 recebem uma nova chance, pois é nos *rendezvous* que as "variáveis críticas" da *task* (variáveis testadas nas condições) são alteradas. Estas filas voltarão a ser referidas adiante, na explicação das instruções *give_up_rdv* e *end accept*.

- Var := E
- if E then C_1 else C_2
- while E do C
- do forever C
- loop C_1; C_2; ... ; C_n endloop
- exit when E
- { C_1; C_2; ... ; C_n }
- read (Var)
- write (Var)
- nl
- tab (K)
- nothing
- mutexbegin
- mutexend
- lock
- unlock
- insert (id , Q)
- insert (id , Q, Prior)
- id := first (Q)
- age (Q)
- block
- wakeup (id)
- P (Sem)
- V (Sem)
- yield
- hold (Time)
- wait (Cond)
- wait (Cond , Prior)
- signal (Cond)
- send (ProcessId, Msg)
- nb_send (ProcessId, Msg)
- receive (ProcessId, Msg)
- id := fork
- id := new Pname (arg_1, arg_2, ..., arg_N)
- join (id)
- quit
- debug1
- debug2
- nodebug
- pause
- exec (Nome)
- accept Et (arg_1: T_1; arg_2: T_2; ... ; arg_N: T_N) returns T when E_L do C

Figura 2 - Comandos da linguagem V4

As facilidades de depuração são implementadas através dos comandos *debug1*, *debug2*, *nodebug*, *pause* e *exec(Nome)*. Aqui será apresentado o comando *debug2*, apenas. Este comando liga um indicador de rastreamento e, a partir daí, para cada instrução de máquina que vai ser executada (antes de executar essa instrução, portanto), o sistema escreve no terminal uma linha com o nome do processo (ou *thread*), o endereço e a instrução que vai ser executada. Feito isso, a execução pára até que seja pressionada a tecla ENTER. Dessa maneira, o usuário visualiza a execução passo a passo. Ao invés da tecla ENTER, o usuário pode digitar "comandos de inspeção" pré-definidos, os quais vão ser executados antes da execução da instrução.

5. A máquina virtual V4

A arquitetura da máquina virtual é bastante avançada, pois o conjunto de instruções mostrado na figura 3 é implementado diretamente por seu "hardware". Em uma máquina convencional, muitas dessas instruções seriam implementadas no *kernel* do sistema operacional. Nas instruções da figura 3, *X* e *Y* são variáveis (podendo ser constantes nas instruções *if*, *write*, *push*, *hold*, *random* e *tab*), *opRel* é um operador relacional, *E* é um endereço de memória, *Proced* é um nome de procedimento, *Id* é um identificador de processo (nome ou número único), *P* é um nome de processo, *Q* é uma fila, *S* é um

semáforo, *M* é um monitor, *C* é uma variável tipo *condition*, *Prior* é uma variável ou constante inteira, *Msg* é uma lista de valores (componentes de uma mensagem) e *Et* é uma *entry* de *task*. Somente as instruções envolvidas na implementação de *rendezvous* serão comentadas a seguir.

if X opRel Y goto E		hold	X	send	Id , Msg
goto	E	fork		receive	Id , Msg
read	X	create	P	rndzvs	Et
write	X	join	Id	accept	Et
push	X	quit		give_up_rdv	
pop	X	mutexbegin		end proced	
add		mutexend		end process	
sub		lock		end procesT	
mult		unlock		end monitor	
div		block		end task	
mod		wakeup	Id	end accept	
jump	E	first	Q	myself	
jzer	E	insert	Id , Q	getId	P
jpos	E	is_empty	Q	clockTime	
jneg	E	count	Q	random	X
jsub	E	P	S	nl	
ret		V	S	tab	X
call	Proced	entermonitor	M	debug1	
[n, a_1, a_2,..., a_n]		leavemonitor	M	debug2	
return		wait	C	nodebug	
nothing		wait	C, Prior	pause	
yield		signal	C	exec	Proced

Figura 3 - Conjunto de instruções da máquina virtual V4

A instrução *call P* serve para criar um registro de ativação na pilha do processo (ou *thread*) e transferir a execução para o procedimento *P*. Esta instrução é sempre seguida, no código objeto, pela lista dos parâmetros de chamada. O primeiro elemento da lista é o número de parâmetros, os demais são os parâmetros propriamente ditos. A instrução *rndzvs Et* funciona como um *call* para o procedimento tipo *entry Et*.

A instrução *give_up_rdv* (abandona *rendezvous*) facilita a implementação da semântica do comando *accept*. Ela é executada quando a expressão booleana da cláusula *when* é avaliada como *false*, e trabalha com as duas filas de espera referidas na apresentação do comando *accept*. O que a instrução faz, basicamente, é colocar o par de *threads* envolvido no *rendezvous* no fim da fila 2 (da *task* do *acceptor*) e passar adiante a UCP, dando preferência para a realização de um outro *rendezvous* (no caso, o primeiro *rendezvous* da fila 1).

As instruções *end proced*, *end process*, *end procesT*, *end monitor*, *end task* e *end accept*, servem para marcar, no código gerado pelo compilador, respectivamente, o fim de uma *procedure*, de um processo, de um modelo de processo, de um monitor, de uma *task* e de um comando *accept*. Além de marcar o fim de um trecho de código estas instruções também ocasionam ações em tempo de execução. O *end accept* (fim de *rendezvous*), por exemplo, faz com que os pares de *threads* da fila 2 sejam transferidos para a fila 1, onde terão nova chance de realizarem seus *rendezvous*.

6. O compilador e o simulador da máquina virtual

O compilador e o simulador são implementados por dois programas independentes. Esses programas se comunicam exclusivamente através do arquivo de saída gerado pelo compilador, o qual é composto por uma *tabela de símbolos* e um *código objeto*. O compilador usa a técnica de tradução dirigida por sintaxe através de um esquema de tradução implementado sobre um analisador LR(1) [Price e Toscani 2000]. Quando um programa fonte é compilado com sucesso e o usuário ordena a sua execução, o simulador gera as estruturas de dados necessárias para a execução desse programa, utilizando as informações contidas na tabela de símbolos.

Cada entrada da tabela de símbolos possui 6 campos: *N*, *Id*, *T*, *Vi*, *A* e *B*, onde *N* é o índice da entrada, *Id* é a identificação (nome) de um símbolo, *T* é o tipo do símbolo, *Vi* é o valor inicial do símbolo, *A* contém informações adicionais (dimensões, se *Id* é nome de *array*, ou número de argumentos, se *Id* é nome de procedimento) e *B* é o bloco no qual o símbolo é declarado. O tipo de um símbolo pode ser: *integer*, *boolean*, *queue*, *semaphore*, *function*, *routine*, *process*, *monitor*, *condition*, *priority condition*, *task*, *entry_fun* e *entry_sub*. O bloco *B* pode ser: *global*, *process(X)*, *procedure(P)*, *monitor(X)*, *procedure(monitor(X), P)*, *task(X)*, *entry(task(X),P)*, *procedure(task(X),P)*.

As estruturas internas para representar variáveis, *arrays*, processos, monitores, *tasks*, *threads*, etc., são relativamente simples e não serão detalhadas neste artigo. Na verdade, todo monitor tem uma fila de entrada, mais uma fila para cada variável *condition* declarada. Todo processo (ou *thread*) tem um registro descritor e uma pilha. Toda *task* tem uma fila de espera para cada *procedure* tipo *entry* declarada, mais uma estrutura com duas filas (ambas já referidas), as quais servem para implementar os encontros nessa *task*.

7. Programas exemplos

Esta seção apresenta dois programas V4. Para o primeiro deles é mostrado o arquivo de saída gerado pelo compilador (tabela de símbolos e código gerado).

7.1. Exemplo de condição de corrida

A figura 4 mostra um programa V4$_0$ completo que define dois processos *p1* e *p2*, e um procedimento *incrS*(*n*) que incrementa *n* vezes a variável global *S*, inicializada com zero. O primeiro processo incrementa *S* 50 vezes e o segundo, uma. Portanto, o valor final de *S* deveria ser 51. Contudo, o resultado (último valor colocado em *S*) poderá ser qualquer valor entre 1 e 51, e esse valor irá variar de uma execução para outra.

A figura 4 também mostra a tabela de símbolos e o código gerado pelo compilador. O código é apresentado em duas colunas, a primeira indica um endereço de memória e a segunda indica a instrução lá contida. Nesse código, #(*i*) representa o *i*-ésimo argumento (ou variável local) e $*k* representa a constante *k*.

7.2. Operações P e V generalizadas com tasks

A figura 5 mostra uma *task* (parte de um programa V4$_5$) que implementa operações *PP* e *VV*, que são generalizações das operações P e V sobre semáforos. Considerando um semáforo de valor *N* como um recurso de *N* unidades e que as operações P e V são, respectivamente, operações de requisição e liberação de uma unidade do recurso, então, analogamente, as operações *PP*(*K*) e *VV*(*K*) representam a requisição e a liberação de *K* unidades do recurso. Vale observar que *PP* atende as requisições na ordem FIFO.

```
V4prog
   S : integer initial 0;
   procedure incrS (n: integer);
   k: integer init 0;
   while k < n do  { S:= S+1;  k:= k+1 };
   process p1;
    { incrS(50);  write(S)  };
   process p2;
    { incrS(1);  write(S)  }
endprog
```

TABELA DE SÍMBOLOS

IND	NOME	TIPO	V_INIC	AGRUP	INSIDE
0	S	integer	0	none	global
1	incrS	routine	0	n_arg(1)	global
2	n	integer	0	none	proced(incrS)
3	k	integer	0	none	proced(incrS)
4	p1	process	12	none	global
5	p2	process	16	none	global

END.	CÓD. GERADO	END.	CÓD. GERADO
0	[if, #(2), <, #(1), goto, 2]	10	[goto, 0]
1	[return]	11	[end, proc]
2	[push, S]	12	[call, incrS]
3	[push, $1]	13	[1, $50]
4	[add]	14	[write, S]
5	[pop, S]	15	[end, process]
6	[push, #(2)]	16	[call, incrS]
7	[push, $1]	17	[1, $1]
8	[add]	18	[write, S]
9	[pop, #(2)]	19	[end, process]

Figura 4. Exemplo 1 – Programa V4$_0$ com race condition

```
task semafN is
    entry PP(N: integer);
    entry VV(N: integer);
    id: integer;
    count: integer init 5;
thread main is
    {   id:= fork;
        if  id = myself
        then  loop
                  accept  PP(N: integer)  when N <= count  do  count:= count-N
              endloop
        else  loop
                  accept VV(N: integer)  do count:= count+N
              endloop
    }
end semafN;
```

Figura 5. Exemplo 2 – Generalização das operações P e V em V4$_5$

8. Conclusão

O objetivo do ambiente de programação V4 é prover uma ferramenta de ensino. O sistema está operacional e seu *kit* de instalação está disponível [Vale4 2004]. O ambiente tem sido usado com sucesso em disciplinas de Programação Concorrente e Sistemas Operacionais na PUCRS e na UNILASALLE. A linguagem foi testada na solução de todos os problemas clássicos (exclusão mútua com *n* processos, produtor-consumidor com *buffer* limitado, barbeiro dorminhoco, fumantes, jantar dos filósofos, *readers & writers*, etc.) e a programação resultante foi sempre simples e clara. Versões anteriores da linguagem foram apresentadas em minicursos no Forum Internacional de Software Livre 2002 e no Simpósio

Brasileiro de Linguagens de Programação 2003. Infelizmente, os minicursos não costumam ficar registrados nos anais dos eventos. A linguagem também é utilizada para ilustrar os algoritmos do livro *Sistemas Operacionais* e *Programação Concorrente* [Toscani, Oliveira e Carissimi 2003]. Aliás, foi nas notas de aula que deram origem ao livro que surgiu a linguagem.

Até o momento, nenhuma especificação formal foi realizada sobre o sistemaV4. Um trabalho interessante (e não trivial) seria a especificação formal da semântica do comando *accept* apresentado informalmente neste artigo. Vale observar que este comando, tal como definido, permite implementar comunicação síncrona não determinística, dispensando o uso de comandos tipo *select*.

Outro trabalho possível seria a introdução de melhorias na parte de tratamento de erros do compilador e na comunicação com o usuário. Uma extensão interessante seria a construção de uma interface gráfica que permitisse visualizar a execução de um programa (por exemplo, ver as filas que se formam nos semáforos, nos monitores, etc.), bem como alterar atributos e políticas do sistema, como a política de escalonamento, por exemplo. Outro trabalho relevante seria a distribuição da linguagem V4, visando permitir a simulação de um ambiente de processos em máquinas distintas, com características de concorrência em cada máquina. Ainda, poderia ser conveniente reprogramar todo o sistema em C, para ganhar em eficiência e universalidade.

Para estas extensões do sistema, espera-se a contribuição da comunidade acadêmica. [2] Pode-se dizer que uma base está pronta e que ela possibilita o desenvolvimento de vários trabalhos interessantes. Vale a pena o investimento, pois se trata de uma ferramenta didática, sem fins lucrativos, que facilita o aprendizado de um tema difícil da computação.

Referências

Andrews, G. *Concurrent Programming: Principles and Practice.* Benjamin Cummings, 1991.

Brinch Hansen, P. *Java' s Insecure Parallelism* ACM SIGPLAN Notices 34, 4 (April), 1999.

Dijkstra, E.W. Cooperating sequential processes. In *Programming Languages* (F. Genuys, editor), Academic Press, 1968.

Hoare, C.A.R. Monitors: an operating system structuring concept. *Comm. ACM* 17, 10, 1974.

Holt, R.C.; Graham, G.S.; Lazowska, E.D.; Scott, M.A. *Structured Concurrent Programming with Operating Systems Applications*. Addison-Wesley, 1978.

Holt, R.C. *Concurrent Euclid, The UNIX System, and TUNIS.* Addison-Wesley, 1983.

Price, A. M.; Toscani, S. S. *Implementação de Linguagens de Programação: Compiladores*. Sagra-Luzzato, Porto Alegre, 2000.

Toscani, S.S.; Oliveira R.S.; Carissimi, A.S. *Sistemas Operacionais e Programação Concorrente*. Sagra-Luzzato, Porto Alegre, 2003.

Taft, S.T., Duff, R.A. *Ada 95 Reference Manual: language and standard libraries.* Springer-Verlag, 1997. (Lecture Notes in Computer Science, Vol. 1246)

Vale4. *Kit de instalação.* Março de 2004. Disponível em www.inf.pucrs.br/~stoscani/V4.

Wielemaker, J. *SWI-Prolog 2.7 Reference Manual.* University of Amsterdam, Dept. of Social Science Informatics. Amsterdam, The Netherlands, 1997.

[2] Todo o sistema V4 foi desenvolvido como "hobby" pelo autor após sua aposentadoria, portanto, sem a força de trabalho de seus estudantes (e essa força é fundamental para a manutenção e melhoria do sistema).