CACIC 2000

VI Congresso Argentino de Ciencias de la Computación

2 al 7, Octubre 2000

Título do Artigo : **Aspectos da Implementação de um Ambiente Multilinguagem de Programação**

Autores: **Aparecido Valdemir de FREITAS e João José NETO**

Escola Politécnica da Universidade de São Paulo
Depto. de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, No. 158. Cidade Universitária
São Paulo – Brasil
e-mail: afreitas@imes.com.br e jjneto@pcs.usp.br

Área de Aplicação: **Computational Languages**

João José NETO

Formado em 1971 na Escola Politécnica da USP em Engenharia de Eletricidade, modalidade Eletrônica. Mestre em Eng. Elétrica pela EPUSP em 1975. Doutor em Eng. Elétrica pela EPUSP em 1980. Livre-Docente pela EPUSP em 1993. Ocupa atualmente o cargo de Professor Associado junto ao Depto. de Eng. de Computação e Sistemas Digitais da EPUSP. Ministra disciplinas de graduação e pós-graduação na área de Sistemas Operacionais, Eng. de Software, Linguagens de Programação, Compiladores e Teoria da Computação. Desenvolve junto à EPUSP a linha de pesquisa em Tecnologias Adaptativas, através da orientação de programas de Mestrado e Doutorado na área.

Aparecido Valdemir de FREITAS

Mestrando em Eng. de Computação – Sistemas Digitais pela EPUSP. Especialização em Engenharia de Computação – Ênfase Programação pela EPUSP-FDTE em 1986. Bacharel em Matemática Plena pela F.S.A. em 1974. Formado em Engenharia Civil pela E.E.Mauá em 1979. Ocupa atualmente o cargo de Professor-I no curso de Ciência da Computação do IMES – Instituto Municipal de Ensino Superior de São Caetano do Sul, onde ministra as disciplinas Sistemas Operacionais-I e Técnicas de Programação.

Aspectos da Implementação de um Ambiente Multilinguagem de Programação

Aparecido Valdemir de Freitas e João José Neto

Escola Politécnica da Universidade de São Paulo
Depto. de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, No. 158. Cid. Universitária – S. Paulo – Brasil
e-mail: afreitas@imes.com.br e jjneto@pcs.usp.br

Abstract: The development of large-scale complex software is often made difficult by the lack of tools allowing the programmer to express his or her ideas in a neat way for all technical aspects involved. This is particularly the case when we deal with the expressive power of programming languages. Since large programs are usually composed by segments of different natures, it seems natural to provide specifical tools for the programmers in order to achieve good expressiveness in the corresponding code. In other that interface between these different segments be accomplished, become viable the use of schemes that facilite the interaction them. The paper presents an implementation propose mechanism of a data change between language modules that compose a multilanguage application. The mechanism would be as well, be applied to languages derived from different paradigms programming. The paper presents a complete simple sample that shows the operation of the proposed environment.
Keywords: paradigm, multilanguage, multiparadigm, environment, composition.

Resumo: O desenvolvimento de software complexo de grande porte é muitas vezes dificultado pela carência de ferramentas adequadas para a clara expressão das idéias dos programadores em todos os aspectos técnicos do projeto. Isto é particularmente verdadeiro quando se lida com o poder de expressão de linguagens de programação. Como os grandes programas se compõem usualmente de segmentos com características técnicas diversificadas, parece natural disponibilizar ferramentas específicas para os programadores, de forma que uma boa expressividade seja obtida no código correspondente. Para que a interface entre estes diferentes segmentos seja efetivada, torna-se viável o emprego de esquemas que facilitem a interação entre os mesmos. O artigo apresenta uma proposta de implementação de um mecanismo de troca de dados entre módulos de linguagens que compõem uma aplicação multilinguagem. O mecanismo pode também ser aplicado a linguagens oriundas de diferentes paradigmas de programação. O artigo também apresenta um pequeno exemplo completo de implementação que exercita parcialmente o ambiente proposto.
Palavras-chave: paradigma, multilinguagem, multiparadigma, ambiente, composição.

1. Motivação

No desenvolvimento de alguma tarefa, muitas vezes encontramos dificuldades em finalizá-la devido a ausência de ferramentas que facilitem a implementação da mesma. Este cenário também pode ser aplicado à área de desenvolvimento de software, uma vez que a linguagem ideal pode não estar disponível, pode não ser de domínio da equipe de desenvolvimento, ou ainda, pode não existir. Assim, ao implementarmos projetos de software, muitas vezes somos obrigados a codificar com a linguagem de programação disponível, mesmo sabendo que esta não seja a linguagem ideal para as necessidades do projeto em questão.

Para o desenvolvimento de aplicações complexas, é possível que tenhamos de manusear diferentes problemas de programação. Considerando que cada um destes problemas poderia ser mais facilmente resolvido com a utilização de uma determinada linguagem de programação, idealmente poderíamos particionar o problema em segmentos, cada qual associado à linguagem mais apropriada.

De acordo com [Zave-89], é comum nas obras de engenharia utilizar-se diferentes materiais para a construção de algum bem. Muitos diferentes materiais com diferentes operações usualmente são necessários para a implementação dos projetos de engenharia. Por exemplo, o aço pode ser cortado, perfurado, laminado, polido; e pode ser colado com plástico, parafusado com outros materiais, etc. Como projetistas de software, necessitamos de um repertório similar de operações para manusear e compor as diferentes linguagens que poderiam ser necessárias para o desenvolvimento da aplicação.

Caso as linguagens componentes da aplicação multilinguagem pertençam a diferentes paradigmas de programação, conforme [Spinellis-94], estaremos diante de uma aplicação multiparadigma. Assim uma aplicação multilinguagem poderá também ser multiparadigma, dependendo dos segmentos de linguagens componentes da aplicação.

De acordo com [Placer-91], quando os itens e relacionamentos existentes no domínio de interesse de um problema estiverem dentro do escopo descrito por um dado paradigma, a tarefa de modelar uma solução para o problema fica mais simplificada.

Por exemplo, ao utilizarmos uma linguagem lógica de programação, estaremos empregando um estilo declarativo, no qual o programador simplesmente apresenta um conjunto de cláusulas (fatos ou regras) e um conjunto de metas a serem atingidas. Este estilo de programação será mais bem aplicado à problemas no qual a forma de pensar do programador estiver atrelada à formulação do problema (estilo declarativo) do que com a especificação de todos os passos necessários à implementação da solução (estilo imperativo).

Por outro lado, caso a linguagem de programação a ser utilizada não abstraia diretamente as entidades do domínio do problema, a solução usualmente é implementada através da simulação de tais entidades com a ajuda dos construtos disponíveis na linguagem. Este procedimento acarreta uma diminuição na expressividade da solução, além de dificultar o processo de implementação da solução, uma vez que a forma de pensar do programador não estará diretamente mapeada na linguagem de implementação.

Assim, quando a implementação da solução é feita com uma única linguagem de programação, o programador usualmente modela as soluções de diversos problemas com os construtos disponíveis na linguagem de programação. Temos neste caso, uma influência direta da linguagem na definição da solução do problema, o que usualmente pode ser constatado quando programadores que trabalham há muitos anos com uma determinada linguagem de programação, quase sempre, têm dificuldades em se adaptar à novos paradigmas de programação, uma vez que sua forma de pensar está muito arraigada ao paradigma corrente.

Podemos definir a programação multilinguagem como uma técnica que tem como meta, permitir que o programador implemente o código através do uso das mais adequadas linguagens ao problema em questão, de tal forma que a implementação da solução possa considerar diversos estilos de programação.

Este artigo tem, como objetivo, especificar e apresentar uma proposta de implementação de um ambiente de programação que viabilize o emprego da programação multilinguagem, através do oferecimento de primitivas de ambiente que facilitem a interface entre os diversos segmentos de linguagens que compõem a aplicação.

2. Arquitetura do Ambiente Multilinguagem proposto

Em nossa proposta, a aplicação multilinguagem será formada por um conjunto de processos que se interagem, cada qual sendo gerado por uma diferente linguagem de programação. Estas diferentes linguagens poderão ou não pertencer a um mesmo paradigma de programação. O ambiente proposto provê algumas funções com o objetivo de facilitar a integração dos módulos componentes, a qual é obtida através de chamadas do sistema operacional, para assegurar a efetiva troca de mensagens entre os processos, de acordo com a Figura-1, extraído de [Freitas-2000].

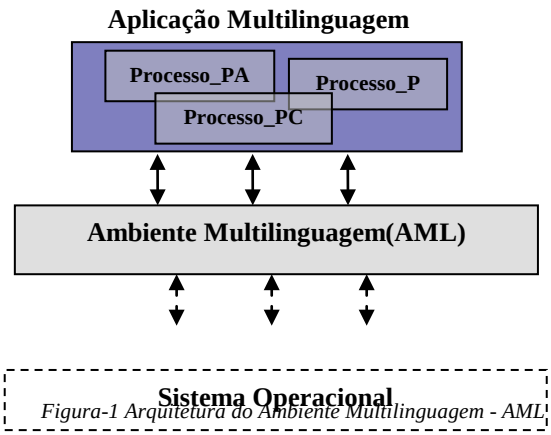


Figura-1 Arquitetura do Ambiente Multilinguagem - AML

Para facilitar a apresentação da arquitetura proposta, utilizaremos a sigla AML para denotar o nosso ambiente multilinguagem de programação. Utilizaremos o ambiente proposto sob a plataforma *Win32*, mas as idéias são aplicáveis a outras plataformas, bastando que sejam substituídas as chamadas de API's para o sistema operacional desejado.

Para implementarmos um adequado esquema, que permita a interoperação dos processos componentes, deveremos tratar os problemas de transferência de dados e de controle entre os processos que irão compor a aplicação.

Caberá ao nosso ambiente multilinguagem fornecer primitivas responsáveis pelo atendimento aos serviços de interação entre os processos componentes.

A troca de dados entre a aplicação e o nosso ambiente multilinguagem dar-se-á através de áreas compartilhadas que serão alocadas e gerenciadas pelo próprio ambiente, conforme Figura-2, adaptada de [Freitas-00].

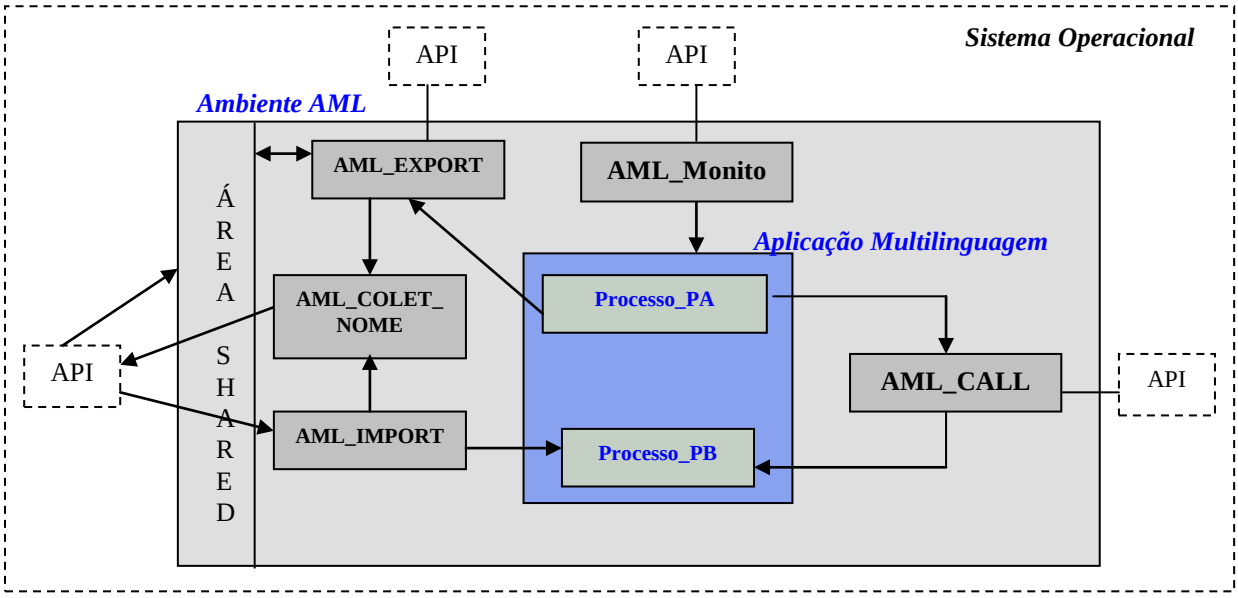


Figura-2 Interação da Aplicação com o Ambiente

3. Primitiva de ambiente para a criação de processos

Tendo em vista que o nosso ambiente multilinguagem será responsável pela interface entre os diversos processos que irão compor a aplicação, e considerando que um determinado processo (representado por um segmento de linguagem) poderá necessitar iniciar um novo processo (o qual estará associado a outro segmento de linguagem), implementaremos em nosso ambiente, uma primitiva com o objetivo de facilitar e padronizar a iniciação de processos.

Assim, caso nossa aplicação multilinguagem seja composta pelos processos P_A, P_B e P_C, sendo P_A o processo inicial, poderemos utilizar uma primitiva de ambiente que a partir do processo P_A, inicie o processo P_B ou P_C. Esta facilidade, além de padronizar todas as chamadas de iniciação de processos, encapsulará as API´s do sistema operacional responsáveis pela criação dos processos.

Considerando que nosso ambiente multilinguagem será implementado na arquitetura *Win32*, e de acordo com [Richter-97, pág. 33], um processo *Win32* é usualmente definido como uma instância de um programa em execução, no qual se associa um espaço de endereçamento onde são alocados o código executável e os respectivos dados da aplicação. Qualquer DLL requerida pelo código executável também terá seu código e dados carregados no espaço de endereçamento do processo. Adicionalmente ao espaço de endereçamento, um processo também possui certos recursos, tais como, arquivos, alocações dinâmicas de memória, e *threads*. Estes vários recursos serão destruídos pelo sistema tão logo o processo seja terminado.

Ainda de acordo com [Richter-97, pág. 34], quando um processo *Win32* é criado, seu primeiro *thread*, chamado *thread primário*, é automaticamente criado pelo sistema. Este *thread primário* pode em seguida criar *threads* adicionais, e estes, por sua vez, poderão também criar outros.

Um processo *Win32* é criado através da chamada da função de API *CreateProcess*. Quando um *thread* em nossa aplicação executa a API *CreateProcess*, o sistema cria um objeto processo que corresponde a uma pequena estrutura de dados que o sistema utiliza para administrar a execução do processo.

Para ilustrar a chamada da API *CreateProcess*, segue abaixo um exemplo da criação de um processo no qual seu *thread primário* será o interpretador *SWI-Prolog 3.2.8*, o qual representa o paradigma lógico de programação, documentado em [Wielemaker-99].

```
#include <windows.h>
#include <iostream.h>
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{  CHAR cmdStr[MAX_PATH] = "C:\\Program Files\\pl\\bin\\PLWIN.EXE";
  STARTUPINFO startUpInfo;
  PROCESS_INFORMATION procInfo;
  BOOL success;
  GetStartupInfo(&startUpInfo);
  success=CreateProcess(0, cmdStr, 0, 0, FALSE, CREATE_NEW_CONSOLE, 0, 0, &startUpInfo, &procInfo);
  if (!success) cout << "Erro na criacao do processo:" << GetLastError() << endl;  return 0; }
```

Para simplificar e uniformizar o procedimento de criação de processos, o código acima poderá ser encapsulados em uma DLL do ambiente multilinguagem, correspondendo assim à primitiva AML_CALL. Esta primitiva será, portanto, responsável pela transferência de controle entre os diversos processos componentes da aplicação.

Os parâmetros necessários para a chamada da API *CreateProcess* estão especificados e detalhados em [Richter-97] ou [Brain-96].

Ao desenharmos uma aplicação multilinguagem, poderemos estruturar os diversos processos de tal modo que cada qual seja responsável pela execução de alguma tarefa. Este procedimento se assemelha ao que empregamos durante a chamada de funções ou subrotinas durante a execução de um determinado programa. Assim, deveremos implementar um esquema de sincronização de processos, para assegurarmos que quando a partir de um processo principal efetuamos uma chamada num outro processo, o processamento do processo principal seja interrompido até que o retorno do processo chamado tenha se efetivado.

De acordo com [Richter-97, pág.70], para criarmos um novo processo e aguardarmos o processamento deste, poderemos utilizar o seguinte código, na plataforma *Win32*:

```
PROCESS_INFORMATION   pi;
DWORD                 dwExitCode;
BOOL                  fSucess=CreateProcess(..., &pi);
if (fSucess) { CloseHandle(pi.hThread); WaitForSingleObject(pi.hProcess, INFINITE);...}
```

No código acima, a API *WaitForSingleObject* assegura que o *thread* primário do processo pai é suspendo até que o processo recém-criado seja terminado.

4. Mecanismo de alocação e gerenciamento da área compartilhada

A primitiva AML_CALL vista no item anterior nos assegura a viabilidade de se implementar a transferência de controle entre os diversos processos componentes da aplicação multilinguagem. No entanto, além da transferência de controle, a interoperabilidade entre processos poderá também exigir a necessidade de transferência explícita de dados.

Para a implementação desta interoperabilidade, o ambiente multilinguagem deverá prover mecanismos responsáveis pela transferência de dados entre os processos componentes. Nosso ambiente será responsável pela alocação e gerenciamento de áreas compartilhadas de dados, disponibilizadas aos processos através de serviços do ambiente.

O módulo monitor do ambiente será responsável pela iniciação de áreas compartilhadas que deverão corresponder aos tipos de dados usualmente disponíveis nas linguagens suportadas pelo ambiente. Cabe ao ambiente cuidar para que os devidos ajustes de representação sejam implementados, de forma que se torne transparente para a aplicação o formato interno dos dados.

Por simplicidade de implementação, estaremos assumindo que a troca de dados entre processos apenas se dará através de tipos compativelmente representados em todas as linguagens componentes da aplicação.

Para a execução da aplicação multilinguagem, o ambiente AML será inicialmente carregado para a alocação de áreas compartilhadas que serão necessárias para se promover a troca de informações entre os diversos processos componentes da aplicação. Em seguida, o usuário informa o nome do processo principal de modo que o ambiente inicie a aplicação.

Conforme [Richter-97, pág. 115], a arquitetura de memória utilizada por um sistema operacional é um importante aspecto para se compreender como um sistema operacional opera. Ao iniciarmos o estudo de um sistema operacional, muitas questões imediatamente são afloradas, como por exemplo, “ Como podemos compartilhar informações entre duas aplicações ? ”. Um bom entendimento de como o sistema operacional efetua o gerenciamento de memória, pode auxiliar a encontrarmos as respostas para questões como esta formulada.

Empregaremos no nosso ambiente multilinguagem, a técnica conhecida por *memory-mapped* para a alocação das áreas compartilhadas do ambiente que serão responsáveis pelo armazenamento e recuperação de informações trocadas entre os diversos processos componentes da aplicação multilinguagem. Conforme [Richter-97, pág. 115], da mesma forma que são executados os procedimentos de gerência de memória virtual, os arquivos *memory-mapped* permitem ao programador reservar uma região do espaço de endereçamento e efetivar área física (*commit*) para esta região.

De acordo com [Kath-93, pág. 3], arquivos *memory-mapped* permitem que aplicações acessem arquivos em disco da mesma forma que as mesmas acessem memória dinâmica através de pointers. Com esta capacidade podemos mapear uma parte ou todo o arquivo para um conjunto de endereços do espaço de endereçamento do processo. Com esta característica, acessar o conteúdo de um arquivo *memory-mapped* é tão simples quanto dereferenciar um ponteiro no intervalo de endereços designados.

Para criarmos um arquivo *memory-mapped* devemos usar a API *CreateFileMapping*. A função *MapViewOfFile* mapeia uma porção do arquivo para um bloco de memória virtual.

Há ainda uma característica adicional associada aos arquivos *memory-mapped*, que corresponde a possibilidade de compartilhá-los entre aplicações. Isto significa que se duas aplicações abrirem o mesmo nome referente a um arquivo *memory-mapped*, as mesmas estarão consequentemente, criando um bloco de memória compartilhada.

As listagens abaixo demonstram o uso de objetos de memória compartilhada que foram criados para mostrar a operação do mecanismo IPC (*interprocess-communication*), conforme [Silberchatz-95, pág. 116]. Estas listagens implementam um mecanismo muito simples, onde um programa, o cliente, deposita uma simples mensagem (um valor *integer*) na memória compartilhada para um outro programa, o servidor, que a recebe e a exibe.

```
#include <iostream.h>
#include <windows.h>
void main(void) {
        HANDLE hmmf;       int * pInt;
        hmmf = CreateFileMapping((HANDLE) 0xFFFFFFFF, NULL,PAGE_READWRITE, 0, 0x1000, "AMLDATA");
        if (hmmf == NULL){cout << "Falha na alocacao da memoria compartilhada.\n";exit(1);}
        pInt = (int *) MapViewOfFile(hmmf, FILE_MAP_WRITE,0,0,0);
        if (pInt == NULL){cout << "Falha no mapeamento de memoria compartilhada!\n";exit(1);}
        Semaf_AML_NOME = OpenSemaphore(SEMAPHORE_ALL_ACCESS,1,"Semaf_AML_NOME");
            if (Semaf_AML_NOME == NULL) {cout << "Falha na abertura do Semaforo Semaf_AML_NOME!!!\n";exit(1);}
        WaitForSingleObject(Semaf_AML_NOME, INFINITE);#include <iostream.h>
            *p_Int = 1;
        ReleaseSemaphore(Semaf_AML_NOME, 1,0); }
void main(void)  {
        HANDLE hmmf;       int * pInt;
        hmmf = OpenFileMapping((HANDLE) 0xFFFFFFFF, NULL,PAGE_READWRITE, 0, 0x1000, "AMLDATA");
        if (hmmf == NULL){cout << "Falha na alocacao da memoria compartilhada\n";exit(1);}
        pInt = (int *) MapViewOfFile(hmmf, FILE_MAP_WRITE,0,0,0);
        if (pInt == NULL){cout << "Falha no mapeamento da memoria compartilhada.\n";exit(1);}
        Semaf_AML_NOME = OpenSemaphore(SEMAPHORE_ALL_ACCESS,1,"Semaf_AML_NOME");
            if (Semaf_AML_NOME == NULL) {cout << "Falha na abertura do Semaforo Semaf_AML_NOME!!!\n";exit(1);}
        WaitForSingleObject(Semaf_AML_NOME, INFINITE);
                cout << "Valor transmitido para o server = " << *pInt);
        ReleaseSemaphore(Semaf_AML_NOME, 1,0); }
```

Os programas exemplificados anteriormente ilustram o mecanismo de transferência de dados entre processos com o uso das API's *CreateFileMapping*, *OpenFileMapping* e *MapViewOfFile*, para o manuseio de um bloco de dados em memória chamado AMLDATA.

Para criarmos um bloco de memória compartilhada, as aplicações podem, durante a chamada da API *CreateFileMapping*, submeter um valor especial como parâmetro de HANDLE correspondente a 0xFFFFFFFF para obter um bloco de memória compartilhada.

Os programas também empregam a técnica de semáforos como meios de sincronização entre os processos participantes da interação. Na plataforma *Win32* um semáforo é criado através da API *CreateSemaphore*, enquanto que a abertura do mesmo é feita pela API *OpenSemaphore*.

Detalhes da codificação das API's de alocação de memória compartilhada poderão ser encontrados em [Richter-99, pág. 627]. Em [Richter-97, pág. 364] estão detalhadas as API's para o emprego de semáforos.

No nosso ambiente multilinguagem, o procedimento de alocação das áreas compartilhadas do ambiente será responsabilidade do módulo AML_MONITOR, sendo portanto, este o responsável pela chamada da API *CreateFileMapping*. Toda a gravação na área compartilhada do ambiente será feita pelo módulo AML_COLET_NOME que se encarregará de gerenciar os nomes que serão exportados e importados pelos processos da aplicação.

5. Primitivas para transferência de dados entre processos

Com a disponibilização da área de memória compartilhada pelo nosso ambiente, os processos participantes da aplicação multilinguagem poderão transferir informações, caso assim o desejem. Para facilitar e padronizar a transferência destes dados na área compartilhada do ambiente, serão implementadas as funções primitivas AML_IMPORT e AML_EXPORT que serão chamadas pelos processos que necessitem de troca de informações. A chamada dessas primitivas de ambiente deve estar acompanhada do nome interno da área no processo, o nome da área externa a ser gravada no ambiente, o tipo de dados e tamanho. Cabe ao ambiente gerenciar o espaço de nomes a ser

compartilhado entre os vários processos componentes da aplicação, além de garantir que o uso das áreas compartilhadas não acarrete anomalias de atualização devido ao seu possível uso simultâneo por processos concorrentes.

Como vimos anteriormente, o módulo do ambiente AML_MONITOR terá a responsabilidade de alocar as áreas compartilhadas do ambiente, e esta ação será feita através de três mecanismos.

O primeiro se refere a uma área de dados compartilhada chamada AML_DATA_AREA, no qual a leitura das informações é não-destrutiva, ou seja, sucessivas operações de leitura poderão ser disparadas na mesma área, sem que se perca o conteúdo da mesma. As primitivas de ambiente AML_EXPORT e AML_IMPORT farão a gravação/leitura de dados trocados entre os processos componentes da aplicação.

Uma segunda forma de implementação da interoperabilidade de processos se dá através da área de dados compartilhada AML_DATA_QUEUE, a qual aloca uma estrutura na forma de pilha ou fila que é acessada de forma compartilhada por todos os processos da aplicação. Neste caso, a gravação/leitura dos dados na fila de dados é feita pelas primitivas do ambiente AML_EXPORT_DATA_QUEUE e AML_IMPORT_DATA_QUEUE. Com este mecanismo, a leitura dos dados é destrutiva, significando que após a leitura dos dados, estes não mais residirão na área compartilhada.

Finalmente, uma terceira forma de implementarmos a comunicação de dados se dá através da implementação da área compartilhada AML_DATA_SWITCH, o qual se comporta como interruptores, e tendo assim, um comportamento análogo à variáveis booleanas de ambiente. As primitivas de ambiente AML_EXPORT_DATA_SWITCH e AML_IMPORT_DATA_SWITCH são as responsáveis pela gravação e leitura dos interruptores.

Para cada um destes mecanismos, o ambiente multilinguagem deverá alocar as áreas compartilhadas compatíveis com os tipos de dados a serem exportados/importados pelos processos, como por exemplo, integer, char, string, float, etc.

As primitivas de ambiente citadas anteriormente deverão encapsular as API's *OpenFileMapping*, uma vez que deverão manusear as áreas compartilhadas alocadas pelo módulo AML_MONITOR.

Considerando que os processos componentes da aplicação deverão chamar as primitivas de ambiente de *import* e *export* de dados, e transferir para estas os dados a serem salvos nas áreas compartilhadas do ambiente, de alguma forma deveremos implementar um mecanismo de *linkedição* destas áreas de transferência. Adotaremos, para este objetivo, no nosso ambiente multilinguagem o esquema de geração das primitivas através de *Dynamic Link Libraries* (DLL's).

De acordo com [Richter-99, pág. 675], é relativamente simples criarmos uma biblioteca de *linkedição* dinâmica. Esta simplicidade ocorre porque uma DLL consiste de um conjunto de funções autônomas que qualquer aplicação poderá utilizar. Assim, não há necessidade de provermos suporte de código para o processamento de *loops* de mensagens ou criarmos janelas quando estivermos criando DLL's . Uma biblioteca de *linkedição* dinâmica é simplesmente um conjunto de módulos de código fonte, cada qual contendo um conjunto de funções. Estas funções são escritas com o pressuposto de que uma aplicação (EXE) ou uma outra DLL irão chamá-las em tempo de execução.

O código seguinte, obtido de [Richter-97], demonstra como exportar uma função chamada *Aml_exp* e uma variável compartilhada inteira chamada *com_area* a partir de uma DLL:

```
__declspec (dllexport)  int   Aml_exp ( int nA1, int nA2) {return (nA1 + nA2);}
__declspec (dllexport)  int com_area = 0;
```

O compilador reconhece a declaração __declspec(dllexport) e a incorpora no arquivo OBJ resultante. Esta informação também é processada pelo *linker* quando todos os arquivos OBJ e DLL

forem *linkeditados*. Maiores detalhes para a construção de DLL's poderão ser encontrados em [Richter-97], [Brain-96], [Rector-97], [Petzold-99], ou ainda em [Solomon-98].

6. Um exemplo completo: Coleta de Nomes através de Autômatos Adaptativos

Para que a troca de informações entre módulos gerados por diferentes processos possa ser viabilizada, será necessário que estabeleçamos um mecanismo de gerenciamento de nomes relativos às diferentes variáveis que serão importadas ou exportadas entre os módulos componentes da aplicação multilinguagem.

Assim, deveremos estabelecer algum mecanismo de passagem de parâmetros entre um módulo qualquer da aplicação e o módulo do ambiente multilinguagem responsável pelo gerenciamento do espaço de nomes do ambiente.

Para o desenvolvimento do coletor de nomes do nosso ambiente multilinguagem, iremos utilizar a técnica de autômatos adaptativos, conforme descrito em [Neto-94]. Outra implementação do mesmo método, aplicado ao processamento de linguagens naturais, está descrito em [Menezes-00].

Podemos imaginar um autômato adaptativo como sendo uma seqüência de evoluções sucessivas de um autômato de pilha estruturado inicial, associados à execução de ações adaptativas. A cada ação adaptativa, uma nova topologia é obtida para o autômato, o qual dará continuidade ao tratamento da cadeia de entrada.

De acordo com [Pereira-99], as alterações impostas pela cadeia de entrada evoluem o autômato de pilha estruturado corrente, formando um novo autômato que continuará a tratar o restante da cadeia de entrada, e assim, sucessivamente, até que a cadeia de entrada seja totalmente processada e o autômato atinja um estado final, dizendo-se neste caso que a cadeia de entrada foi aceita pelo autômato.

Com estes conceitos, associaremos o nome a ser coletado e gerenciado pelo ambiente à um autômato, no qual cada caractere do *string* entrado irá corresponder a um estado do referido autômato. À medida que os caracteres forem sendo lidos, a rotina de coleta de nomes deverá criar os estados correspondentes e ao final da cadeia de entrada, o último estado (correspondente ao último caractere) será marcado através de um *flag*, caracterizando que o nome passará a pertencer ao ambiente.

Exemplificando, suponhamos que o nome "*abc*" deva ser exportado ao ambiente através de algum módulo escrito numa linguagem qualquer. Teremos, para o nome "*abc*", o seguinte autômato, conforme apresentado na figura 3.

Assim, para a coleta do nome "*abc*", a rotina de coleta de nomes deverá criar os estados 0,1,2 e 3, e em seguida marcar o estado 3, como um estado no qual o caminho "*abc*" passa a ser gerenciado pelo ambiente. A partir desta ocorrência, qualquer outro módulo da aplicação multilinguagem que queira também definir o nome "*abc*" receberá da rotina de coleta de nomes uma mensagem de erro, uma vez que o nome somente pode ser definido um vez.

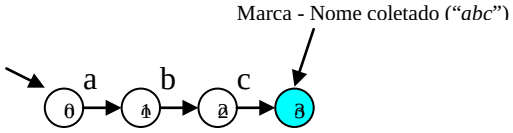


Figura 3 – Configuração do autômato p/ string "abc"

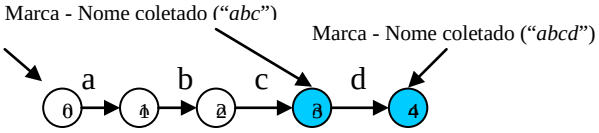
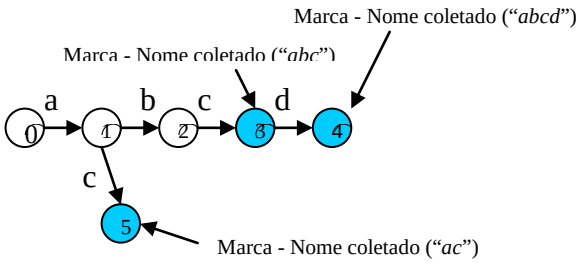


Figura 4 – Configuração do autômato p/string "abcd"

Assim, a partir da inclusão do nome "*abc*" na tabela de nomes administrada pelo ambiente, caso um outro *string*, por exemplo "*abcd*", seja entrado, a rotina de coleta de nomes apenas criará um novo estado 4, uma vez que o caminho "*abc*" já é conhecido (o caminho "*abc*" já foi aprendido pelo

autômato adaptativo). Teremos para o string ”*abcd*” uma adaptação ao autômato anteriormente descrito, conforme figura 4.

Vejamos ainda mais um exemplo. Imaginemos que algum módulo da aplicação multilinguagem necessite gravar no ambiente o nome “*ac*”. Neste caso, o autômato será novamente adaptado ao string requisitado, e por conseguinte, será criado um estado 5, no qual também será marcado o nome “*ac*” como sendo um nome armazenado no ambiente.



A figura 5, ilustra a configuração do autômato adaptativo com os três strings apresentados como exemplo. (“*abc*”, “*abcd*” e “*ac*”).

Para a implementação da rotina de coleta de nomes do ambiente multilinguagem iremos utilizar a linguagem C++, com a técnica de autômatos adaptativos, conforme descrita em [Neto, 94].

Figura 5 – Configuração do autômato adaptativo adicionando o string “ac”

Utilizaremos para a modelagem do autômato adaptativo acima descrito, as classes Estado, Transição, Célula e Autômato. A Figura 6, abaixo, mostra um esquema das estruturas de dados implementada na rotina de coleta de nomes. Esta rotina será acionada pelo módulo do ambiente AML_MONITOR, toda vez que algum processo necessitar de exportar ou importar dados para a área compartilhada do ambiente, o que se dará através das chamadas AML_IMPORT e AML_EXPORT.

Cabe também ao ambiente multilinguagem cuidar para nomes não mais utilizados sejam removidos do autômato, o que deverá ser feito pela eliminação da marca de nome coletado nas folhas da árvore, seguido de possível eliminação de estados e transições, com conseqüente eliminação da memória alocada pela estrutura de dados.

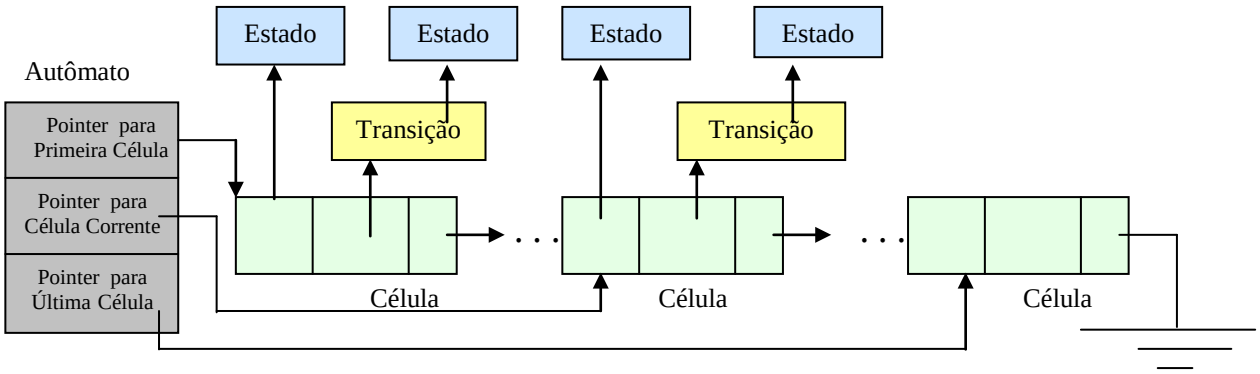


Figura 6 – Estrutura de dados para a implementação do coletor de nomes

Segue abaixo a codificação simplificada do procedimento de coleta de nomes, através da definição das classes necessárias para a construção do autômato adaptativo.

```
// Automato.h
#ifndef AUTOMATO_H
#define AUTOMATO_H
#include "Celula.h"
#include "Elemento.h"
class Automato{
public: Automato();
        int  Pesquisa_Celula(char caractere_lido);
        void Lista_Celulas();
        void Add_Celula(char caractere_lido, int tipo_ins);
        void Lista_Pilha();
        void Automato::Posiciona_celula_Est_Corrente();
```

```
// Celula.h
#ifndef Celula_H
#define Celula_H
#include "Transicao.h"
class Celula {
public: Celula(int cont_celula, Estado* pNewEstado,
          Transicao* pNewTransicao);
Celula();
int             id_Celula;
Estado*         pEstado;
Transicao*      pTransicao;
Celula*         pNext;
```

```
        void Automato::Empilha_Transicoes_Iniciais();
        void Automato::Concatena_nomes();
        char*           nome;
        void            Desempilha();
        void            Empilha(Elemento * p_El_emp);
        void            Lista_nomes();
        Celula*         p_PrimeiraCelula;
        Celula*         p_CelulaCorrente;
        Estado*         p_EstadoCorrente;
        Estado*         p_EstadoInicial;
        Elemento*       p_Topo_da_Pilha;
        Elemento*       p_Elemento_da_Pilha_Corrente;
        static int      cont_estado;
        static int      cont_transicao;
        static int      cont_celula;
};
#endif
class Estado {
public: Estado();
        Estado( int     id_Estado,
                bool    simbolo_do_ambiente);
        int     id_Estado;
        bool    simbolo_do_ambiente;
        int     Valor_Associado_ao_Nome;
};
```

```
        Celula*         pAnterior; };
#endif
#ifndef ELEMENTO_H
#define ELEMENTO_H
#include "Celula.h"
class Elemento {
public: Elemento();
Celula*         p_Celula;
Elemento*       p_Elemento_Posterior;
char*           prefixo_nome;
Elemento*       p_Elemento_Anterior;
};
#endif

#include "Estado.h"
class Transicao {

public:  Transicao(int id_Transicao,
         char atomo, Estado* pEstado);
int      id_Transicao;
char     atomo;
Estado*  p_Estado;
};
```

7. Exemplo de uma aplicação multilinguagem utilizando o ambiente AML

Apresentaremos a seguir um simples exemplo para mostrar a interação entre processos da aplicação e o ambiente multilinguagem proposto. A aplicação consiste em 3 processos P1, P2 e P3, gerados pelas linguagens MS-Visual C++6.0, SWI-Prolog 3.2.8 e Java JDK1.1, respectivamente.

O processo P1 solicita do usuário, através da console, um dado inteiro que é armazenado na área compartilhada do ambiente. Em seguida, o processo P2 importa o dado do ambiente, calcula o fatorial do valor entrado, e armazena de volta o resultado na área compartilhada do ambiente. Finalmente, o processo P3 importa o resultado do ambiente e o exibe ao usuário.

Segue abaixo, a codificação da aplicação, de forma simplificada, com as chamadas de primitivas do ambiente proposto, implementadas através de DLL's *Win32*.

```
//-------------------------------------------
//Proc. P1– Entrada de Dados–MS-Visual C++ 6.0
//-------------------------------------------
#include <stdio.h>
extern void AML_EXPORT(char *, int);
void main(void)
{
int numero = 0;
        printf("\nEntre com um numero: ");
        scanf("%d", &numero);
        fflush(stdin);
        printf("Numero lido foi: %d\n", numero);
AML_EXPORT("trab", numero);
// A variável trab é exportada p/ o ambiente…
}

//-------------------------------------------
//Proc. P2 – Cálculo Fatorial–SWI-Prolog 3.2.8
//-------------------------------------------
:-load_foreign_library(AML_IMPORT).
:-load_foreign_library(AML_EXPORT).
fatorial :-
        AML_IMPORT("trab", N),          ;
// A variavel trab é importada do ambiente…
        fact(N,F),
        AML_EXPORT("result', F).        ;
// A variavel result é exportada para o
ambiente…
```

```
fact(N,F) :-
        N>0,
        N1 is N-1,
        fact(N1,F1),
        F is N * F1.
        fact(0,1).

//-------------------------------------------
//Proc.P3-Exibição do Resultado–Java JDK 1.1
//-------------------------------------------
import java.awt.*;
import corejava.*;
class Oopsaida  extends CloseableFrame
{
   public static int n = 0;
   public void paint (Graphics g)
{g.drawString("O resultado do fatorial = " + n,
75,100);}
   public native int AML_IMPORT();
   static {System.loadLibrary("AML_IMPORT");}
   public static void main (String[] args){
   Frame  f = new Oopsaida();
   n = new Oopsaida().AML_IMPORT("result");
   // A variavel result é importada
   f.show();
   return;}
}
```

8. Conclusão

Este artigo apresentou as motivações e uma proposta de realização concreta de ambiente para programação multilinguagem. A proposta aqui apresentada foi implementada e testada em situações simples, e está sendo complementada para formar um ambiente experimental em que possam ser desenvolvidas aplicações segunda a técnica da decomposição multilinguagem. Os resultados até aqui obtidos encorajam o prosseguimento do trabalho e sua aplicação pedagógica no ensino de linguagens de programação, engenharia de software, ambientes de programação e sistemas operacionais.

Prevê-se para breve a publicação de dissertação de mestrado do primeiro autor deste trabalho, no qual os detalhes de seu projeto e realização estão descritos, bem como a implementação de uma aplicação multilinguagem que se utilize do ambiente AML proposto.

Um grande número de desdobramentos deste trabalho pode ser vislumbrado, envolvendo atividades com banco de dados e com a rede de computadores, no sentido de tornar mais prático e versátil a ferramenta aqui proposta. Trabalhos nesta direção deverão ser iniciados em breve.

9. Referências Bibliográficas

[Brain-96] Marshall Brain, *"Win32 System Services"*, Prentice Hall PTR, Second Edition, 1996.

[Freitas-00] Aparecido Valdemir de Freitas e João José Neto, *"Aspectos do Projeto e Implementação de Ambientes Multiparadigmas de Programação"*, ICIE Y2K, VI International Congress on Information Engineering, UBA- April 26-28, Argentina.

[Kath-93] Randy Kath, *"Managing Memory-Mapped Files in Win32"*, Microsoft Developer Network Technology Group, February 9, 1993.

[Menezes-00] Carlos Eduardo Dantas de Menezes e João José Neto, *"Um método para a construção de analisadores morfológicos, aplicado à língua portuguesa, baseado em autômatos adaptativos"*, Dissertação de Mestrado. Escola Politécnica da USP, 2000.

[Neto-94] João José Neto, *"Adaptive automata for context-dependent languages"*, ACM SIGPLAN Notices, Volume 29, Número 9, Setembro 1994.

[Pereira-99] João José Neto e Joel Camargo Dias Pereira – "Ambiente Integrado de Desenvolvimento de Reconhecedores Sintáticos, baseado em Autômatos Adaptativos" – Dissertação de Mestrado - EPUSP – 1999.

[Petzold-99] Charles Petzold, *"Programming Windows"*, Fifth Edition, Microsoft Press, 1999.

[Placer-91] John Placer, *"Multiparadigm Research: A New Direction in Language Design"*, ACM Sigplan Notices, Volume 26, Número 3, páginas:9-17, March 1991.

[Rector-97] Brent E. Rector e Joseph M. Newcomer, *"Win32 Programming"*, Addison Wesley Longman, Inc., Addison-Wesley Advanced Windows Series, Alan Feuer, Series Editor, 1997, ISBN: 1-57231-995-X.

[Silberschatz–95] Abraham Silberschatz , e Peter B. Galvin, *"Operating System Concepts"*, Addison-Wesley Publising Company, Inc., Fourth edition, 1995, ISBN: 0-201-50480-4.

[Richter-97] Jeffrey Richter, *"Advanced Windows"*, Third Edition, Microsoft Press, 1997, ISBN: 1-57231-548-2.

[Richter-99] Jeffrey Richter, *"Programming Applications for Windows"*, Fourth Edition, Microsoft Press, 1999.

[Solomon-98] David A. Solomon, *"Inside Windows NT"*, Second Edition, Microsoft Press, Microsoft Programming Series, 1998, ISBN: 1-57231-677-2.

[Spinellis-94] Diomidis D. Spinellis, *"Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming"*, February 1994, A thesis submitted for the degree of Doctor of Philosophy of the University of London and for the Diploma of Membership of Imperial College.

[Wielemaker-99] Jan Wielemaker, *"SWI-Prolog 3.2 Reference Manual"*, January 1999, University of Amsterdam, Dept. of Social Science Informatics (SWI) – http://www.swi.psy.uva.nl/projects/SWI-Prolog/.