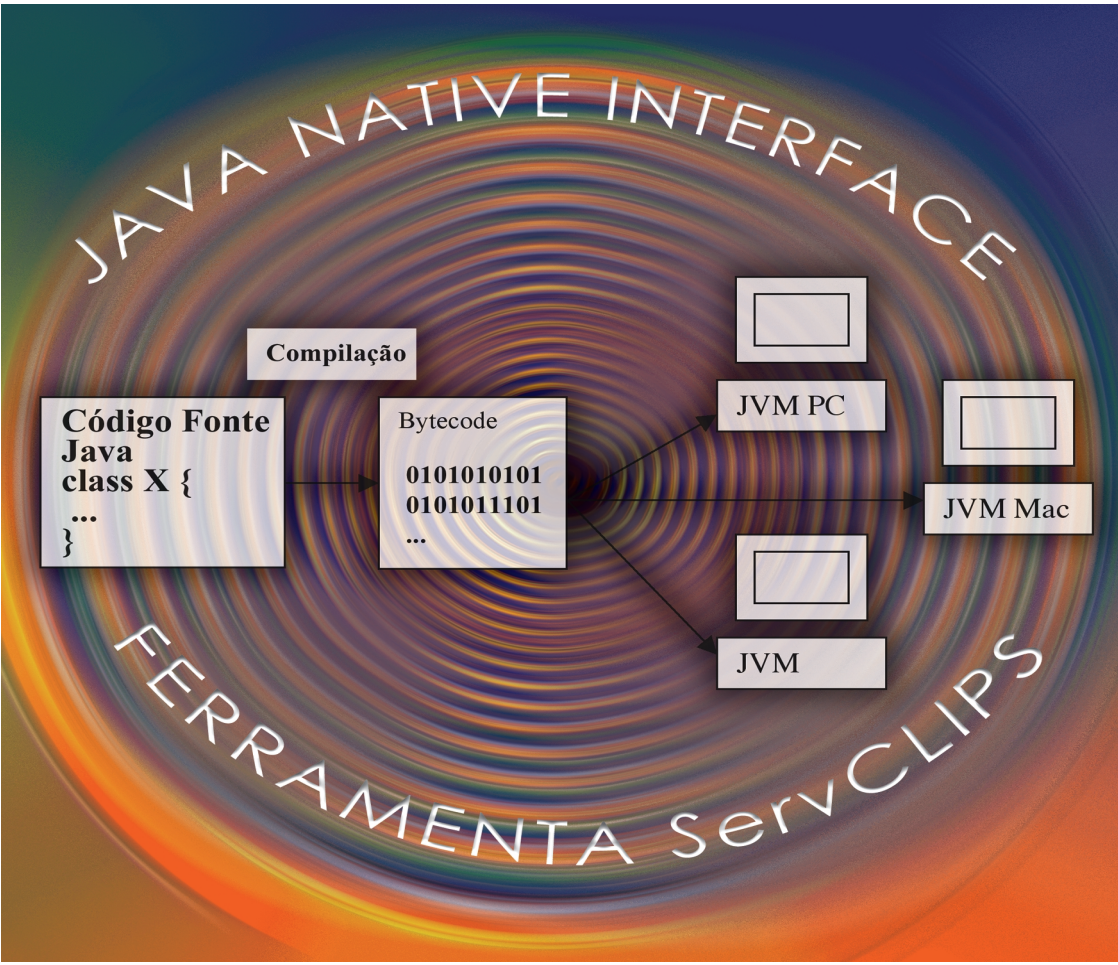
Ministério da Agricultura, Pecuária e Abastecimento

Documentos 9

Dezembro, 2001

ISSN 1677-9274

Aplicação da JNI – Java Native Interface – na Construção da Ferramenta ServCLIPS



Embrapa

Embrapa

Novembro, 2001

Empresa Brasileira de Pesquisa Agropecuária
Embrapa Informática Agropecuária
Ministério da Agricultura, Pecuária e Abastecimento

Documentos 9

ISSN 1677-9274

Aplicação da JNI – Java Native Interface – na Construção da Ferramenta ServCLIPS

Sérgio Aparecido Braga da Cruz

Campinas, SP
2001

Embrapa Informática Agropecuária
Área de Comunicação e Negócios (ACN)
Av. Dr. André Tosello s/nº
Cidade Universitária "Zeferino Vaz" – Barão Geraldo
Caixa Postal 6041
13083-970 – Campinas, SP
Telefone/Fax: (19) 3789-5743
URL: http://www.cnptia.embrapa.br
Email: sac@cnptia.embrapa.br

Supervisor editorial: *Ivanilde Dispato*
Normalização bibliográfica: *Marcia Izabel Fugisawa Souza*
Capa: *Intermídia Publicações Científicas*
Editoração eletrônica: *Intermídia Publicações Científicas*

1ª edição

Cruz, Sergio Aparecido Braga da.
Aplicação da JNI – Java Native Interface – na construção da ferramenta ServCLIPS / Sergio Aparecido Braga da Cruz. — Campinas : Embrapa Informática Agropecuária, 2001.

57 p. : il. — (Documentos / Embrapa Informática Agropecuária ; 9)

ISSN 1677-9274

1. Linguagem Java. I. Título. II. Série.

CDD – 005.133 (21. ed.)

Autor

Sergio Aparecido Braga da Cruz

Mestre em Engenharia Elétrica, Pesquisador da Embrapa Informática Agropecuária, Caixa Postal 6041, Barão Geraldo - 13083-970 - Campinas, SP. e-mail sergio@cnptia.embrapa.br

Apresentação

Este documento reflete a experiência adquirida no desenvolvimento de ferramentas criadas no âmbito do projeto SVTTA (Serviços Virtuais para Transferência de Tecnologia Agropecuária) que teve como objetivo agilizar o processo de transferência de tecnologia agropecuária através da adoção de recursos e serviços de multimídia da Internet. Um dos tópicos abordados neste projeto foi a criação de um ambiente para diagnóstico de doenças consultado remotamente através da WWW (World Wide Web). Este ambiente, atualmente, é formado por um *site* contendo um sistema automatizado para identificação de doenças do milho e uma série de recursos permitindo a comunicação entre técnicos e produtores agropecuários com especialistas em fitopatologia. A tecnologia JNI (*Java Native Interface*) descrita neste documento foi utilizada na implementação da ferramenta ServCLIPS, base para automação do processo de identificação de doenças neste ambiente de diagnóstico.

José Gilberto Jardine
Chefe-Geral

Sumário

Aplicação da JNI - Java Native Interface - na Construção da Ferramenta Servclips

Sergio Aparecido Braga da Cruz

Introdução

Algumas vezes a construção de uma nova solução de *software* envolve a reutilização de soluções prontas, evitando duplicação de esforços e diminuindo o seu tempo de desenvolvimento. Esta estratégia foi adotada no desenvolvimento da ferramenta ServCLIPS, implementada para suprir demandas do subprojeto Diagnóstico Remoto, vinculado ao projeto Serviços Virtuais para Transferência de Tecnologia Agropecuária (SVTTA) da Embrapa Informática Agropecuária.

A ferramenta ServCLIPS permite a execução de sistemas especialistas que podem ser consultados via WWW (Krol, 1993) e foi desenvolvida para permitir a construção de sistemas especialistas para diagnóstico remoto de doenças.

Para o desenvolvimento desta ferramenta foi necessária a utilização de uma solução já existente responsável pela execução de regras de produção de sistemas especialistas conhecida como CLIPS (Riley, 2001; Giarratano, 1998). A extensão desta ferramenta para permitir consultas a sistemas especialistas via WWW envolveu a utilização do mecanismo conhecido como Java Servlets (Hunter & Crawford, 1998).

CLIPS é uma ferramenta para construção de sistemas especialistas desenvolvida pela *Software Technology Branch* (STB), NASA/*Lyndon B. Johnson Space Center*. Desde a sua primeira versão liberada em 1986 ela vem sendo constantemente refinada e aperfeiçoada. Seu projeto visa facilitar o desenvolvimento de software que modele o conhecimento e raciocínio humano. Estes são representados pela CLIPS utilizando regras, funções e programação orientada a objeto. CLIPS foi projetado para uma integração completa com outras linguagens. Pode tanto ser usada como uma aplicação *stand-alone*, ser invocada a partir de outra aplicação ou invocar funções externas. Esta ferramenta está disponível na forma de código fonte em linguagem C, o qual pode ser livremente usado, modificado e redistribuído sem restrições. (Giarratano, 1998).

Na solução implementada no subprojeto Diagnóstico Remoto, a ferramenta CLIPS foi utilizada em modo embutido, ou seja, o programa ServCLIPS é o programa principal que invoca funções da API (*Application Program Interface*) da CLIPS (Cubert et al., 1998), para realização de tarefas relativas a execução de regras de produção de sistemas especialistas.

A linguagem Java (Niemeyer & Peck, 1997), também utilizada no desenvolvimento da ferramenta ServCLIPS, surgiu em meados dos anos 90, tendo como principal características a independência da plataforma de execução por meio da utilização de uma máquina de execução virtual conhecida como JVM (*Java Virtual Machine*).

A linguagem Java é uma linguagem compilada e interpretada. Um arquivo fonte em Java deve ser compilado gerando como resultado um arquivo binário contendo um conjunto de instruções em um formato universal para execução pela JVM. O código Java compilado é conhecido como *bytecode* e para ser executado deve ser interpretado pelo interpretador de *bytecode* ou JVM. Este interpretador simula todas as atividades de um processador real, porém num ambiente seguro e controlado. Este interpretador executa um conjunto de instruções baseados em pilha, gerencia a memória *heap*, cria e manipula tipos de dados primitivos, carrega e invoca blocos de código recém-referenciados, além de exercer outras atividades de controle. Todas as propriedades do interpretador Java são definidas por meio de uma especificação aberta produzida pela Sun MicroSystems (Gosling et al., 1996), a qual permite que qualquer

um possa implementar uma JVM. Nesta especificação nenhuma característica da linguagem é deixada dependente da implementação, por exemplo, todos os tipos primitivos da linguagem Java tem dimensões especificadas.

O interpretador é geralmente implementado nas linguagens C ou C++, podendo ser executado isoladamente ou embutido dentro de outra aplicação, como por exemplo nos navegadores Web.

Com o uso da JVM todo código Java torna-se implicitamente portável (Fig. 1). Uma mesma aplicação Java pode ser executada em qualquer plataforma que possua uma JVM.

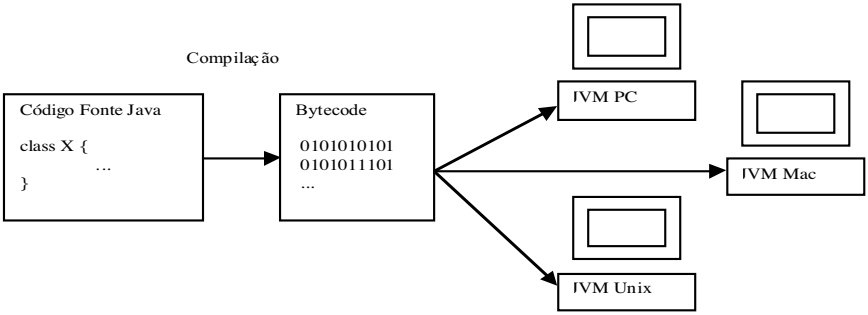


Fig.1. Ambiente JVM.

Na seção Linguagem de Programação Java são apresentadas em mais detalhes algumas características da linguagem de programação Java necessárias ao entendimento do funcionamento da JNI. Na seção *Java Native Interface* - JNI o funcionamento e a utilização da interface JNI são descritos. A seção Ferramenta ServCLIPS: um exemplo de Uso da JNI aborda o uso da JNI no desenvolvimento da ferramenta ServCLIPS. E finalmente, Conclusão e Comentários Gerais sobre o uso de JNI.

Linguagem de Programação Java

A unidade fundamental de construção na linguagem de programação Java é a classe. Seguindo a mesma idéia de outras linguagens orientadas à objeto, em Java classes são componentes que definem uma estrutura de

dados e operações válidas, conhecidas como métodos, que podem ser aplicadas sobre estes dados. As classes relacionam-se de modo a implementar os conceitos do paradigma de orientação à objeto. Ao serem compiladas as classes Java geram *bytecodes* que determinam como os métodos definidos na classe devem ser executados pela JVM, além de armazenarem outras informações sobre a classe. O *bytecode* resultado da compilação das classes pode ser carregado dinamicamente em tempo de execução pela JVM para que sejam executados a partir de arquivos locais ou de servidores de rede.

O ambiente de desenvolvimento Java fornece ao programador um conjunto de classes iniciais definindo uma biblioteca padrão sobre a qual novas classes podem ser implementadas. Esta biblioteca é formada por classes completamente implementadas em Java e por classes básicas cuja implementação é dependente da plataforma (Fig. 2). As classes definidas na biblioteca padrão da linguagem Java podem fazer uso de classes básicas da linguagem, as quais fazem o mapeamento entre o recurso virtual definido pela JVM e o recurso real disponível na plataforma específica para a qual a JVM foi implementada. Estas classes definem, por exemplo, acesso ao sistema de arquivos, rede, e sistema de gerenciamento de janelas. A implementação dos métodos destas classes é dependente da plataforma onde a JVM deve ser executada e para suportá-los é necessária a utilização de código nativo. As demais classes e ferramentas do ambiente Java são independentes de plataforma, pois são implementadas em Java, como por exemplo o compilador Java.

Alguns aspectos importantes relativos ao ambiente de execução da linguagem Java:

- Ambiente seguro – a JVM realiza uma série de verificações de segurança e confiabilidade do *bytecode* antes que ele seja executado. Esta verificação é realizada em vários níveis, desde a verificação do *bytecode* bem formado até o gerenciamento dos acessos a recursos;
- *Garbage colection* (coleta de lixo) – processo de gerenciamento de memória implementado na JVM que elimina a necessidade de alocação e desalocação explícita de memória. A JVM detecta quando objetos em memória não são mais referenciados por outros objetos, eliminando-os no momento mais adequado;

- *Multithreading* – Java suporta *threads* utilizando contruções próprias da linguagem. As *threads* permitem que partes do código Java possam ser executadas paralelamente. A execução paralela pode ser sincronizada ou não, sendo que esta característica também é suportada pela linguagem.

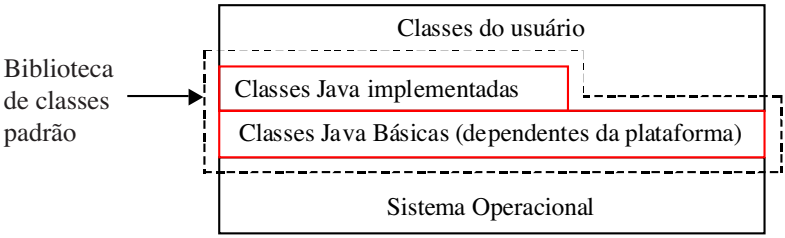


Fig. 2. Biblioteca de classes padrão.

Java Native Interface - JNI

Apesar da variedade de recursos disponíveis no ambiente Java por meio de sua biblioteca de classes padrão, em algumas situações pode ser necessário a utilização de código nativo para implementação de funcionalidades específicas. Algumas destas situações podem ser:

- a biblioteca de classes padrão Java não suporta alguma característica dependente de plataforma necessária a uma aplicação;
- já existe uma biblioteca em código nativo implementada em outra linguagem, a qual se deseja tornar acessível ao código Java;
- a aplicação possui trechos com restrições de execução de tempo que devem ser implementadas em linguagem *assembly* ou outra linguagem.

A JNI permite que aplicações construídas em outras linguagens possam:

- criar, inspecionar e alterar objetos Java (incluindo *arrays* e *strings*);
- chamar métodos Java;
- capturar e disparar exceções Java;
- carregar e obter informações de classes;
- realizar validação de tipo em tempo de execução.

Uma aplicação implementada em outra linguagem pode também utilizar a JNI em conjunto com a API de invocação da JVM para tornar a JVM embutida na aplicação, como é o caso dos navegadores Web. Isto pode ser realizado caso a JVM esteja disponível também na forma de uma biblioteca compartilhada (*shared library* ou *dynamic link library* no ambiente Windows). Neste caso a aplicação em código nativo deve ser compilada em conjunto com a biblioteca compartilhada da JVM. A JNI corresponde a uma API definindo funções sobre a JVM que podem ser invocadas em tempo de execução por intermédio de uma estrutura de ponteiros.

A JNI que será descrita neste documento corresponde a especificação definida pela Javasoft (Sun Microsystems, 2001a) e implementada no ambiente Java JDK 1.1.8 (Sun Microsystems, 2001b), a qual tenta assegurar que a implementação da JNI atinja os seguintes objetivos:

- compatibilidade binária: uma mesma biblioteca criada em código nativo deve ser suportada pelas diferentes implementações da JVM para uma mesma plataforma;
- eficiência: a invocação de código nativo deve ser realizada com o menor *overhead* possível de maneira a diminuir o impacto no tempo de execução de aplicações em tempo real;
- funcionalidade: a interface deve expor estruturas e funcionalidades da JVM na medida certa, de modo a não comprometer a conformidade da JVM com a sua especificação.

Existem outras interfaces definidas para JVM específicas; no entanto, a utilização destas interfaces implica em não portabilidade do código nativo entre as diferentes JVM. Exemplos de outras especificações de interface são a JRI (*Java Runtime Interface*) da Netscape (Harris, 1996) e as soluções Microsoft RNI (*Raw Native Interface*) e J/Direct (Microsoft Corporation, 2002). A especificação da interface JNI definida pela Javasoft atende aos objetivos por ela definidos e foi elaborada com base nas experiências da JRI e RNI.

A principal alternativa ao JNI atualmente são as soluções Microsoft RNI e J/Direct. O uso da RNI permite o acesso a estruturas e funções mais internas da JVM e desta forma propicia uma melhor eficiência na execu-

ção do código nativo pela JVM. Isto, por outro lado, exige uma maior atenção do desenvolvedor, uma vez que a especificação da interface se torna mais complexa e o uso inadequado destas estruturas e funções pode corromper a JVM. A J/Direct simplifica a utilização de código nativo pela JVM definindo uma interface mais simples em relação a RNI e realizando algumas operações automaticamente, como por exemplo, a tradução de tipos de dados comuns de Java para C/C++ e vice-versa sem intervenção do desenvolvedor. O uso de código nativo pela JVM via o mecanismo J/Direct reduz a necessidade de codificação porém não é tão eficiente quanto a RNI.

As soluções Microsoft para uso de código nativo pela JVM estão mais fortemente integradas ao ambiente Windows e desta forma fazem melhor uso de seus recursos permitindo uma solução mais eficiente. Esta solução pode ser adotada quando a restrição de plataforma de execução do programa Java não for um fator importante, uma vez que o uso da RNI e J/Direct impede que o código nativo seja usado por outras JVM para Windows que não sejam da Microsoft. A migração de plataforma de hardware ou sistema operacional se torna mais complexa uma vez que a RNI e a J/Direct não são especificações padrão para interfaceamento com código nativo e podem não estar disponíveis em JVMs implementadas para outras plataformas. A migração do código relativo ao interfaceamento se torna uma atividade a mais, além do processo de migração do código nativo em si.

Várias JVMs de diferentes fabricantes e para diferentes plataformas suportam a JNI. Um pequeno conjunto de JVMs que suportam JNI está listado na Tabela 1.

Tabela 1. JVM e fabricantes correspondentes.

JVM	*Site do fabricante*
kaffe	http://www.kaffe.org
Excelsior JET	http://www.excelsior-usa.com
Appeal JRockit	Http://www.jrockit.com
JDK 1.3 IBM/Windows	Http://www7b.boulder.ibm.com/wsdd/wspvtdevkit-info.html
JDK 1.3 IBM/Linux	Http://www-106.ibm.com/developerworks/java/jdk/linux130/devkit-info.html
JDK HP	Http://www.hp.com/products1/unix/java/
JDK SGI	Http://www.sgi.com/developers/devtools/languages/java.html
JDK Apple/Mac	Http://devworld.apple.com/java/

Com o uso da JNI, a troca da JVM para uma mesma plataforma de hardware e sistema operacional não implicará em nenhuma mudança tanto do código nativo quanto do código Java que o utiliza. A troca de plataforma de hardware ou sistema operacional exigirá migração somente do código nativo não relacionado ao interfaceamento com o Java. O uso da JNI permite uma maior flexibilidade na escolha da plataforma e da JVM que podem ser utilizadas para execução de uma solução Java com acesso a código nativo, além de reduzir o esforço no processo de migração de código quando este é necessário. O ServCLIPS surgiu da necessidade de utilização de ferramentas que pudessem ser executadas em máquinas mais robustas para atendimento de serviços WWW, no caso uma máquina Sun Ultra 2. O uso da JNI possibilitará que o esforço de migração do ServCLIPS para outra plataforma, quando necessário, seja menor.

Arquitetura da JNI

A JNI organiza as informações necessárias para o interfaceamento com código nativo por meio de uma estrutura de dados contendo um *array* de ponteiros para funções da JNI (Fig. 3). A JVM coloca disponível para o código nativo um ponteiro para esta estrutura. Tal abordagem impede o conflito de nomes de funções com o código nativo. A interface JNI organizada desta maneira também propicia que uma mesma JVM possa suportar diferentes versões de JNI, por exemplo:

- uma versão JNI com checagem de tipo e suporte à depuração;
- uma versão JNI mais eficiente com menor esforço na checagem de tipo e suporte a depuração.

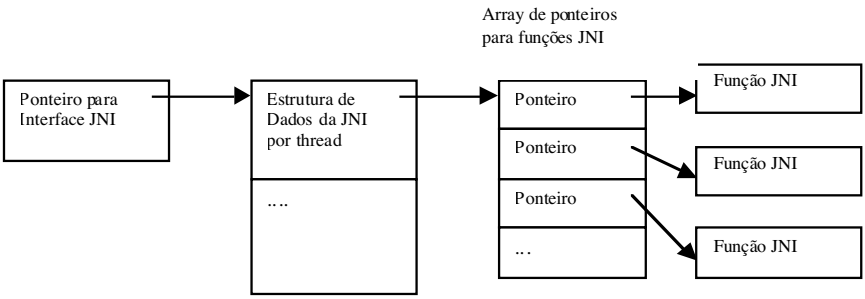


Fig. 3. Estrutura da JNI.

O ponteiro para a interface JNI organiza as funções e dados da JVM por *threads* Java, ou seja, a cada *thread* Java em execução corresponde um ponteiro JNI. Com isto códigos nativos chamados de uma mesma *thread* poderão armazenar dados locais ao contexto de execução daquela *thread*. Um mesmo codigo nativo chamado de *threads* diferentes receberá ponteiros JNI diferentes.

Visibilidade dos Elementos Java pelo Código Nativo

O código nativo tem acesso a elementos Java por meio da estrutura de dados e funções da JNI. Estas informações são passadas para o código nativo utilizando o ponteiro da interface JNI. Este ponteiro organiza as funções e dados no contexto de uma *thread*.

O conjunto de funções da JNI pode ser dividido nas seguintes categorias:

- Informações sobre versão
- Operações sobre classes
- Exceções
- Referências globais e locais
- Operações sobre objetos
- Acesso a campos de objetos
- Chamada de métodos de instâncias
- Acesso a campos estáticos
- Chamada a métodos estáticos
- Operações sobre *strings*
- Operações sobre *array*
- Registro de métodos nativos
- Operações de monitoramento (controle de sincronização de *threads*)
- Interface com a JVM

A definição do ponteiro para interface JNI pode ser encontrada no arquivo jni.h como sendo:

*typedef const struct JNINativeInterface *JNIEnv;*

O arquivo jni.h acompanha o ambiente de desenvolvimento Java e permite a integração da JVM com código nativo desenvolvido nas linguagens C ou C++. A integração com outras linguagens além destas pode ser realizada de duas maneiras. Na primeira maneira o arquivo jni.h pode ser traduzido para a linguagem desejada. Este é o caso, por exemplo, do arquivo jni.pas correspondendo a tradução do arquivo jni.h para permitir a integração da JVM com a código nativo desenvolvido na linguagem Delphi (Mead, 2002). Uma segunda maneira é utilizar uma camada intermediária de código nativo desenvolvido em C ou C++ permitindo a comunicação com o código nativo desenvolvido na linguagem desejada, desde que esta última possua uma interface com C ou C++.

A tabela de ponteiros de funções disponível no arquivo jni.h está representada de maneira simplificada a seguir.

```
const struct JNINativeInterface ... = {
        NULL,
        NULL,
        NULL,
        NULL,
        GetVersion,
        DefineClass,
        FindClass,
        NULL,
        NULL,
        NULL,
        GetSuperclass,
        IsAssignableFrom,
    ...
        MonitorEnter,
        MonitorExit,
        GetJavaVM,
}
```

Alguns ponteiros são inicializados com NULL, de maneira a reservar espaço na tabela para futuras atualizações. Todas as funções estão descritas no manual de referência da JNI (Sun Microsystems, 2001a), como exemplificado a seguir.

GetVersion
jint GetVersion(JNIEnv *env);
Returns the version of the native method interface.

PARAMETERS:
env: the JNI interface pointer.

RETURNS:
Returns the major version number in the higher 16 bits and the minor version number in the lower 16 bits.

In JDK1.1, GetVersion() returns 0x00010001.

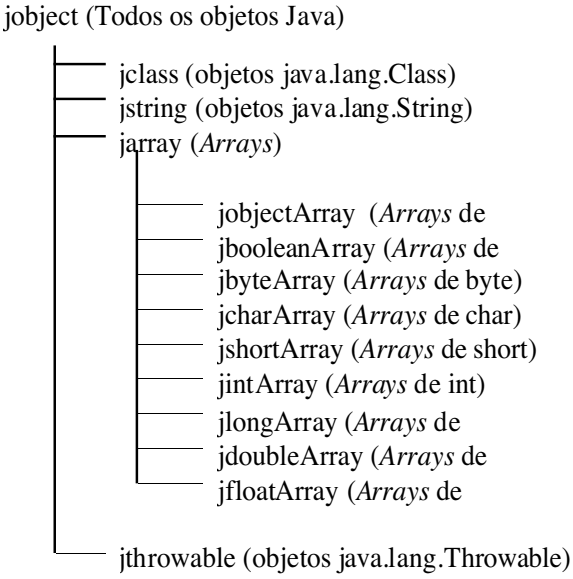
Todas as funções da JNI têm como argumento o ponteiro JNIEnv, a partir do qual dados da JVM podem ser manipulados e funções JNI podem ser chamadas. O código nativo faz uso destas funções da JNI para acessar e manipular os elementos da JVM.

Os tipos de dados disponíveis na linguagem Java são mapeados para tipos de dados na linguagem de desenvolvimento do código nativo para que possam ser passados como argumentos nos métodos nativos ou serem obtidos como valores de retorno de sua execução. O mapeamento de tipos primitivos do Java para tipos nativos definidos pela JNI está descrito na Tabela 2.

Tabela 2. Tipos Java e correspondentes tipo nativo definidos pela JNI.

Tipo Java	*Tipo nativo*	*Descrição*
Boolean	jboolean	*unsigned* 8 bits
Byte	jbyte	*signed* 8 bits
Char	jchar	*unsigned* 16 bits
Short	jshort	*signed* 16 bits
Int	jint	*signed* 32 bits
Long	jlong	*signed* 64 bits
Float	jfloat	32 bits
Double	jdouble	64 bits
Void	void	vazio

Tipos de objetos Java são mapeados seguindo a hierarquia de tipos nativo definida pela JNI:



A JNI fornece um conjunto de funções para manipulação de cada um destes tipos de objeto.

Quando a JVM invoca a execução de código nativo é utilizada a convenção de invocação padrão de código binário executável definido para a plataforma. Ou seja, o modo como os argumentos são armazenados na pilha de passagem de parâmetros durante a execução do código binário seguirá convenções especificadas para a plataforma. O modificador JNICALL, definido no arquivo jni.h e utilizado na declaração das funções C/C++ implementando o código nativo, indica qual é esta convenção quando necessário. Outro modificador importante definido no arquivo jni.h é o JNIEXPORT. Este modificador indica ao compilador que a função C/C++ correspondente ao código nativo deve ser preparada para uso externo, ou seja, seu nome deve ser acrescentado numa tabela de nomes que será utilizada no processo de *link* durante a carga do código nativo pela JVM.

A seguir são apresentados vários trechos de código fonte da ferramenta ServCLIPS ilustrando a implementação de um método Java em C.

Primeiramente apresenta-se um trecho de arquivo Java contendo declaração do método *int Load(String filename)* da classe JavaCLIPS que deverá ser implementado em C. A indicação de que este método está implementado em código nativo é feita pelo modificador *native* existente na declaração do método.

```
public class JavaCLIPS {
                    ...
/**
 * Equivale a funcao Load da API do CLIPS
 * Carrega arquivo clp no ambiente de execucao do CLIPS.
 *
 * @param filename nome do arquivo CLP
 * @return          0 se arquivo não pode ser aberto, -1 se o arquivo
 *          foi aberto mas ocorreu erro durante a carga e          1 se o arquivo
 *          foi carregado com sucesso.
 */
          public native int Load(String filename);

          ...
}
```

O código seguinte corresponde a um trecho do arquivo javaclips.h contendo o protótipo da função C que implementa o método *int Load(String filename)*. O nome da função C que implementa o método Java deve ser construído seguindo regras de nomenclatura que permitirão que ela seja adequadamente carregada e executada pela JVM:

```
#include <jni.h>
/* Header for class JavaCLIPS */

...

/*
 * Class:     JavaCLIPS
 * Method:    Load
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_JavaCLIPS_Load(JNIEnv *, jobject, jstring);

...
```

Por fim, a seguir esta listado um trecho de arquivo javaclips.c contendo a implementação do método *Load* em C. Esta implementação realiza a chamada da função *Load* da API da CLIPS. Nesta função C a maior parte do

código fonte está relacionada com a tradução de tipos de dados definidos pela JNI para tipos de dados C.

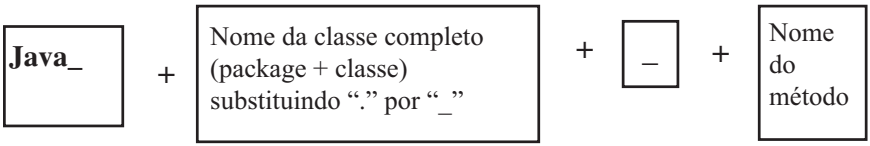
```
...

/*
 * Class:     JavaCLIPS
 * Method:    Load
 * Signature: ()I
 */

JNIEXPORT jint JNICALL Java_JavaCLIPS_Load
  (JNIEnv *env, jobject thisObj, jstring s)
{
      jint  r;
      const char *filename = (*env)->GetStringUTFChars(env, s, 0);
      r=Load(filename);
      (*env)->ReleaseStringUTFChars(env,s,filename);
      return r;
}
```

Visibilidade do Código Nativo pelo Java

A JVM é informada a respeito da implementação de métodos nativos por meio do modificador ***native*** antes do nome do método.

```
public class JavaCLIPS {
...
public native void InitializeEnvironment();
public native void Clear();
public native int Load(String filename);
…
}
```

Para que o método nativo seja executado, a JVM realiza a chamada do código nativo correspondente ao método. A correspondência do nome do método com o da sua função C/C++ equivalente segue a seguinte regra:



Algumas vezes métodos de uma classe Java são sobrecarregados ou seja, dois ou mais métodos de uma classe apresentam o mesmo nome. A diferenciação dos métodos é feita então pelos tipos de seus argumentos. A assinatura de um método corresponde a uma representação da ordem e tipo de seus argumentos. Se o método nativo estiver sobrecarregado (*overloaded*) a assinatura do método deve ser acrescentada ao nome da função C/C++ gerado através da regra anterior separado deste por "__". Cada elemento da assinatura deve estar separado por um "_" . Na montagem do nome da função C/C++ correspondente ao método nativo os tipos de argumentos devem ser representados de acordo com a Tabela 3.

Tabela 3. Símbolos correspondentes aos tipos de argumento em métodos nativos.

Tipo de dado Java	*Símbolo*
Boolean	Z
Byte	B
Char	C
Short	S
Int	I
Long	J
Float	F
Double	D
classe (completamente qualificada)	L*classe*_2
tipo[]	_3tipo

Na representação de assinatura, "/" em nome de classes completamente qualificadas devem ser substituídas por "_". A Tabela 4 apresenta exemplos de formação de nomes de funções C/C++.

Tabela 4. Exemplos de formação de nomes de métodos nativos em C.

Classe e método	*Nome da função C*
Método de classe JavaCLIPS sem *package*	

public class JavaCLIPS {
...
public native void Load();
...
}
	Java_JavaCLIPS_Load

Método de classe JavaCLIPS em *package*	

package cnptia.diagnose;
 	
public class JavaCLIPS {
...
public native void Load();
...
}
	Java_cnptia_diagnose_JavaCLIPS_Load

Método sobrecarregado de classe JavaCLIPS sem *package*	

public class JavaCLIPS {
...
public native void Load();
public native void Load(int i);
public native void Load(int i,float f);
public native boolean Load(String s);
public native boolean Load(boolean b, String [] s);
...
}
	Java_JavaCLIPS_Load__
	Java_JavaCLIPS_Load__I
	Java_JavaCLIPS_Load__IF
	Java_JavaCLIPS_Load__Ljava_lang_String_2
	Java_JavaCLIPS_Load__Z_3Ljava_lang_String_2

Método sobrecarregado de classe JavaCLIPS em *package*	

package cnptia.diagnose;
public class JavaCLIPS {
...
public native void Load();
public native void Load(int i);
public native void Load(int i,float f);
public native boolean Load(String s);
public native boolean Load(boolean b, String [] s);
...
}
	Java_cnptia_diagnose_JavaCLIPS_Load__
	Java_cnptia_diagnose_JavaCLIPS_Load__I
	Java_cnptia_diagnose_JavaCLIPS_Load__IF
	Java_cnptia_diagnose_JavaCLIPS_Load__Ljava_lang_String_2
	Java_cnptia_diagnose_JavaCLIPS_Load__Z_3Ljava_lang_String_2

Apesar do programador poder gerar manualmente estes nomes, na prática utiliza-se a ferramenta **javah,** distribuída no pacote de desenvolvimento Java, a qual gera arquivos .h com os protótipos para as funções C a partir de uma classe Java compilada contendo métodos nativos. A chamada da ferramenta **javah** para geração de arquivos.h compatíveis com a JNI deve ser:

javah –jni <Nome da Classe>

Não existe ferramenta com funcionalidade similar à **javah** para outras linguagens fornecida no ambiente de desenvolvimento Java padrão.

O código nativo sempre recebe como argumento um ponteiro para a interface JNI, por meio do qual poderão ser chamadas funções da JNI que permitem acesso aos recursos da JVM. Para que possa ser carregado pela JVM, o código nativo deve formar uma biblioteca. A carga dinâmica da biblioteca deve ser realizada explicitamente pela aplicação Java utilizando, por exemplo, o código que segue:

```
static {
     System.loadLibrary("clips");
}
```

O método System.loadLibrary(String nomedebiblioteca) faz com que a JVM carregue dinamicamente bibliotecas, estendendo desta forma as suas funcionalidades. No sistema operacional (SO) Solaris arquivos que podem ser carregados dinamicamente tem extensão **.so**, no ambiente Windows tem extensão **.dll**. O mapeamento do nome da biblioteca utilizado como argumento em System.loadLibrary com o nome do arquivo real da biblioteca é dependente do sistema. No Solaris a biblioteca armazenada no arquivo de nome "libclips.so" é referenciada com o nome "clips". Para que o código nativo possa ser utilizado na forma de biblioteca estática ele deverá ser previamente *link* editado com a JVM.

Passos para Criação de Aplicações Utilizando JNI

Para implementação de uma aplicação que necessita utilizar a interface JNI para interação com código nativo, os seguintes passos podem ser seguidos (SUN ...,2001c):

1. Editar arquivo fonte Java contendo classe que declara métodos nativos.
2. Compilar a classe Java.
3. Gerar o arquivo .h descrevendo os protótipos das funções para o código nativo. Para isto utilizar a ferramenta javah com a opção –jni. Conhecendo o protótipo da função é possível criar a sua implementação.
4. Implementar as funções nativas utilizando qualquer linguagem como, por exemplo, C ou C++. Quando for necessário, as funções nativas deverão utilizar as funções da JNI para interação com a JVM.
5. Gerar uma biblioteca compartilhada (*shared library*) a partir da compilação do código fonte das funções nativas.
6. Executar a aplicação.

Métodos adequados, tais como o System.loadLibrary(String nomedabiblioteca) ou Runtime.getRuntime().loadLibrary(nomedabiblioteca), deverão ser chamados pela aplicação Java para carga dinâmica da biblioteca compartilhada. O ambiente de execução da JVM deverá estar adequadamente configurado para permitir a carga da biblioteca. No SO Solaris a variável de ambiente LD_LIBRARY_PATH deverá conter o diretório onde esta o arquivo da biblioteca.

Para reusar bibliotecas já existentes que não seguem o padrão JNI de interface é necessário gerar uma camada de código intermediário para mapeamento de chamadas de métodos nativos pela JVM, seguindo a interface JNI para código nativo. Este foi o caso do ServCLIPS, onde a API da CLIPS já está definida. O mapeamento das chamadas da JVM para a API da CLIPS é realizado pelo conjunto de funções em C implementadas no arquivo javaclips.c (Anexo C).

Ferramenta ServCLIPS: um Exemplo de Uso da JNI

A interface JNI foi utilizada na implementação da ferramenta ServCLIPS para desenvolvimento de sistemas especialistas com acesso via Web. O uso da JNI possibilitou a integração do código Java com a biblioteca da ferramenta CLIPS implementada na linguagem de programação C. A biblioteca CLIPS possui uma API ampla, abrangendo vários aspectos da ferramenta. Na implementação da integração entre Java e CLIPS foram escolhidas as funções mais adequadas da API da CLIPS que permitem a construção da aplicação. Foi necessária a criação de uma camada intermediária de funções C, mapeando as funções da API da CLIPS com funções seguindo a nomenclatura de acordo com as regras da JNI. Esta camada de funções intermediárias serve também para a conversão adequada de tipos de dados e gerenciamento da transferência de dados realizadas entre a API CLIPS e o restante da aplicação implementada na linguagem de programação Java.

O Anexo A fornece o código fonte Java utilizado na implementação da classe JavaCLIPS, que encapsula os métodos nativos e constantes necessárias para integração da CLIPS. Esta classe faz a interface entre a parte Java e a parte C da ServCLIPS. Para definição das variáveis mem-

bros e métodos desta classe foi necessário um levantamento das funções, constantes e variáveis definidas na API da CLIPS que seriam necessárias para implementação da ServCLIPS. Cada uma das funções necessárias foi então mapeada para um método da classe JavaCLIPS.

O Anexo B fornece o código fonte do arquivo de protótipos das funções C, definindo a camada de funções intermediárias para comunicação com a CLIPS. Este arquivo foi gerado utilizando a ferramenta javah a partir da classe JavaCLIPS compilada.

O Anexo C fornece o código fonte em C da implementação da camada intermediária de funções nativas, que realizam chamadas diretas à API da CLIPS. Este camada é responsável pela conversão de tipos e armazenamento de informações trocadas entre a parte Java e a CLIPS.

Um aspecto importante na implementação da classe JavaCLIPS e das funções C correspondentes refere-se ao modo como a CLIPS interage com o usuário. CLIPS faz uso de um mecanismo denominado *ROUTER,* que define um dispositivo virtual de entrada ou saída. Na CLIPS são definidos sete *ROUTERs* básicos, mas outros podem ser definidos pelo usuário (programador). O comportamento do ROUTER quanto ao tratamento de saídas ou obtenção de entrada de dados pode ser redefinido. No arquivo javaclips.c (Anexo C) foram criadas funções que redefinem o comportamento das saídas da CLIPS de modo que estas são direcionadas para um *buffer*. Foram então definidos métodos nativos na classe JavaCLIPS que recuperam o valor armazenado neste *buffer* para utilização no restante do código Java da ServCLIPS.

Na Tabela 5 são apresentados os métodos na classe JavaCLIPS e a sua função C correspondente, cuja implementação completa está disponível no arquivo javaclips.c.

Antes que o código nativo seja executado é necessário que ele seja carregado pela JVM. Isto é realizado na classe JavaCLIPS (Anexo A) através da chamada *System.loadLibrary("clips")* ilustrada no trecho de código a seguir. A chamada deste método da classe *System* é realizada dentro de um bloco de código estático de modo que seja executada na primeira vez que a classe JavaCLIPS for carregada pela JVM.

Tabela 5. Métodos nativos da classe Java CLIPS e funções C correspondentes.

Método na classe JavaClips	*Função C correspondente*
Mapeamento da API do CLIPS	
public native void InitializeEnvironment();	JNIEXPORT void JNICALL Java_JavaCLIPS_InitializeEnvironment (JNIEnv *env, jobject thisObj)
public native void Clear();	JNIEXPORT void JNICALL Java_JavaCLIPS_Clear(JNIEnv *env, jobject thisObj)
public native int Load(String filename);	JNIEXPORT jint JNICALL Java_JavaCLIPS_Load(JNIEnv *env, jobject thisObj, jstring s)
public native void Reset();	JNIEXPORT void JNICALL Java_JavaCLIPS_Reset(JNIEnv *env, jobject thisObj)
public native boolean Save(String filename);	JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Save(JNIEnv *env, jobject thisObj, jstring s1)
public native long Run(int runlimit);	JNIEXPORT jlong JNICALL Java_JavaCLIPS_Run(JNIEnv *env, jobject thisObj, jint runLimit)
public native int Focus(String moduleName);	JNIEXPORT jint JNICALL Java_JavaCLIPS_Focus(JNIEnv *env, jobject thisObj, jstring modulename)
public native void AssertString(String strfact);	JNIEXPORT void JNICALL Java_JavaCLIPS_AssertString(JNIEnv *env, jobject thisObj, jstring s2)
public native boolean LoadFacts(String filename);	JNIEXPORT jboolean JNICALL Java_JavaCLIPS_LoadFacts (JNIEnv *env, jobject thisObj, jstring s3)
public native boolean LoadFactsFromString (String inputstring);	JNIEXPORT jboolean JNICALL Java_JavaCLIPS_LoadFactsFromString(JNIEnv *env, jobject thisObj, jstring s4)
public native boolean SaveFacts(String filename);	JNIEXPORT jboolean JNICALL Java_JavaCLIPS_SaveFacts(JNIEnv *env, jobject thisObj, jstring s5)
public native int GetWatchItem(String item);	JNIEXPORT jint JNICALL Java_JavaCLIPS_GetWatchItem(JNIEnv *env, jobject thisObj, jstring jitem)
public native boolean Watch(String item);	JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Watch(JNIEnv *env, jobject thisObj, jstring jitem)
public native boolean Unwatch(String item);	JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Unwatch(JNIEnv *env, jobject thisObj, jstring jitem)
public native void CommandLoop();	JNIEXPORT void JNICALL Java_JavaCLIPS_CommandLoop (JNIEnv *env, jobject thisObj)
Tratamento de *ROUTERs* (obtenção de saída do CLIPS)	
public native void InitRouters();	JNIEXPORT void JNICALL Java_JavaCLIPS_InitRouters(JNIEnv *env, jobject thisObj)
public native String GetOut(int codigorouter);	JNIEXPORT jstring JNICALL Java_JavaCLIPS_GetOut(JNIEnv *env, jobject thisObj,jint codigorouter)
public native byte [] GetOutByteArray (int codigorouter);	JNIEXPORT jbyteArray JNICALL Java_JavaCLIPS_GetOutByteArray(JNIEnv *env, jobject thisObj, jint codigorouter)
public native void CloseOut();	JNIEXPORT void JNICALL Java_JavaCLIPS_CloseOut(JNIEnv *env, jobject thisObj)
Função definida para verificação do estado da CLIPS	
public native boolean ehFim();	JNIEXPORT jboolean JNICALL Java_JavaCLIPS_ehFim(JNIEnv *env, jobject thisObj)

```
public class JavaCLIPS {
            ...
/**
 * Equivale a funcao Load da API do CLIPS
 * Carrega arquivo clp no ambiente de execucao do CLIPS.
 *
 * @param filename nome do arquivo CLP
 * @return  0 se arquivo não pode ser aberto, -1 se o arquivo
 *    foi aberto mas ocorreu erro durante a carga e 1 se o arquivo
 *    foi carregado com sucesso.
 */
     public native int Load(String filename);

     ...

      static {
           System.loadLibrary("clips");
     }
}
```

A carga do código nativo pela JVM deve resolver todas as pendências criadas pelos métodos nativos da classe JavaCLIPS que até então não apresentavam um código executável. Isto é realizado através da identificação do código nativo correspondente a cada um dos métodos nativos de JavaCLIPS no processo conhecido como *link* edição dinâmica (realizada em tempo de execução).

Feito isto qualquer método nativo da classe JavaCLIPS poderá ser utilizado por outras classes Java. Se, por exemplo, o método *Load(String filename)* for chamado, a JVM identificará que este é um método nativo. O argumento *filename* na chamada do método é adequadamente convertido para o tipo *jstring*, estrutura de dados que armazena uma cadeia de caracteres no formato UTF-8. Uma estrutura de dados do tipo *jobject* é criada para armazenar informações da instância do objeto JavaCLIPS utilizada para realizar a chamada do método nativo. Através desta estrutura e utilizando funções da JNI o código nativo poderá acessar e alterar valores de variáveis membros e chamar outros métodos da classe (mesmo métodos Java). Uma estrutura do tipo JNIEnv contendo informações gerais sobre a JVM e permitindo o acesso às funções JNI é sempre passada para o código nativo.

A JVM invoca o código nativo correspondente ao método nativo *Load*, neste exemplo, *Java_JavaCLIPS_Load* (Anexo C) passando estas três estruturas de dados como valor dos argumentos. No trecho do código fonte em C a seguir está a implementação correspondente ao método nativo *Load(String filename)* da classe JavaCLIPS. Esta função C realiza a carga de um arquivo de regras de um sistema especialista no ambiente de execução da CLIPS. Para isto é utilizada a função da API do CLIPS *Load(char* filename)*, porém esta chamada não poderá ser realizada utilizando diretamente o argumento recebido pelo método nativo. O nome do arquivo armazenado em *jstring s* recebido pela função *Java_JavaCLIPS_Load* está codificado no formato UTF-8. A função JNI *GetStringUTFChars* é chamada para criar uma *string* C (*array* de *char)*. A função *Load(char* filename)* da API da CLIPS pode então ser chamada usando este valor convertido e retornando um valor na variável *r* do tipo *jint*. Como uma variável do tipo *int* do C corresponde ao tipo *jint* da JNI, não haverá necessidade de conversões. Para liberação do recurso de memória utilizado no armazenamento da *string* C é utilizado a função JNI *ReleaseStringUTFChars*. O valor de *r* é retornado pela função *Java_JavaCLIPS_Load* e será recebido pelo método da classe Java que invocou a chamada do método nativo *Load(String filename)* da classe JavaCLIPS.

```
...

/*
 * Class:     JavaCLIPS
 * Method:    Load
 * Signature: ()I
 */

JNIEXPORT jint JNICALL Java_JavaCLIPS_Load
  (JNIEnv *env, jobject thisObj, jstring s)
{
      jint  r;
      const char *filename = (*env)->GetStringUTFChars(env, s, 0);
      r=Load(filename);
      (*env)->ReleaseStringUTFChars(env,s,filename);
      return r;
}
```

No ServCLIPS a classe ConsultaSE responsável pelo encapsulamento de uma sessão de consulta a um sistema especialista utiliza a classe JavaCLIPS para acessar os recursos da máquina de inferência CLIPS.

Conclusão e Comentários Gerais

Apesar do grande número de recursos e funcionalidades disponíveis na plataforma Java, existe um limite de viabilidade para o desenvolvimento de aplicações 100% Java para certas situações. A busca por integração com códigos legados, restrições de tempo em aplicações de tempo real, acesso a recursos de *hardware* específicos, dentre outros motivos, exigem que haja algum mecanismo de interação da JVM com o mundo externo. A interface JNI permite esta interação de maneira padronizada e segura. A sua especificação tenta equilibrar um alto desempenho na comunicação com códigos nativos sem expor demasiadamente as estruturas e funcionalidades internas da JVM.

A especificação atual da interface JNI não tem o objetivo de ser a única resposta para a necessidade de integração entre a JVM e códigos nativos. Implementadores de JVM estão livres para criarem suas próprias soluções dando, por exemplo, maior ênfase em eficiência permitindo o acesso às estruturas internas da JVM num nível mais baixo, ou simplificando a interface permitindo a interação num nível mais alto. Porém, utilizar a JNI garante que um mesmo código binário seja compatível com as diversas implementações de JVM para uma mesma plataforma uma vez que a interface JNI deverá estar disponível em todas elas.

A utilização da interface JNI no desenvolvimento da ferramenta ServCLIPS possibilitou o uso do código CLIPS, desenvolvido em linguagem diferente do Java (linguagem C). Isto possibilitou uma redução de tempo e de esforço no desenvolvimento da ferramenta. A ferramenta CLIPS possui uma interface textual simples em sua versão *stand-alone.* Por meio da JNI foi possível aperfeiçoar os recursos de interface da ferramenta, tornando-a acessível via WWW. A interface JNI pode ser então usada para aumentar a vida útil de um código legado, possibilitando a sua integração com novas tecnologias por meio da linguagem Java. O ambiente Java pode ser usado como uma plataforma para integração de sistemas,

encapsulando código legado e tornando-o disponível, de maneira transparente, para utilização por outros sistemas.

Existe uma ferramenta alternativa integrando a CLIPS com a WWW e disponível sob a licença GNU (Free Software Foundation, 2002) denominada WebCLIPS (Giordano, 2002). Esta ferramenta adota a abordagem CGI (*Common Gateway Interface*) para a integração com a WWW. O WebCLIPS não foi utilizado devido aos seguintes fatores:

- a solução CGI é reconhecidamente de grande *overhead;*
- o uso desta solução dificultaria a construção de versões *stand-alone*;
- menor flexibilidade para integração com outros sistemas.

Além dos fatores citados, a implementação de uma nova solução possibilita a capacitação em tecnologia de integração de Java com código legado. Foram comparados o desempenho do WebCLIPS (tecnologia CGI) e do ServCLIPS (tecnologia Servlet + JNI) na execução de uma mesma regra de sistema especialista. Para esta comparação foram utilizadas as ferramentas JMeter 1.5.1 (Apache Software Foundation, 2002) e perfmeter do Solaris. A ferramenta JMeter foi utilizada na geração do teste de carga e monitoramento de tempo de resposta. A ferramenta perfmeter foi utilizada no monitoramento da porcentagem de utilização e média de número de processos (*Load*) em execução no processador.

Os dados obtidos são mostrados na Tablela 6.

Tabela 6. Comparação entre ServCLIPS e WebCLIPS.

Ferramenta	*CPU (%)*	*Load (processos)*	*Tempo resposta (ms)*
ServCLIPS	50	1.07	2890
WebCLIPS	51	1.48	2000

Configuração do JMeter: 5 *threads* , taxa de geração de requisições 300 ms.

O ServCLIPS apresentou um tempo médio de resposta maior que o WebCLIPS. Isto provavelmente devido a reconhecida demora na invocação de código nativo pela JVM. Por outro lado a média de processos executáveis no processador (*Load*) foi menor durante a execução do

ServCLIPS. Para cada requisição atendida pelo WebCLIPS é invocada a execução de um novo processo. Este é um comportamento normal para programas CGI.

Apesar do maior tempo de resposta a ferramenta ServCLIPS possui como vantagens a flexibilidade na geração de interfaces e de acesso a outros recursos e sistemas via ambiente Java, tais como:

- acesso a banco de dados utilizando o recurso JDBC do Java;
- acesso a internet através de protoloco TCP-IP ou HTTP utilizando bibliotecas de comunicação do Java;
- flexibilidade na construção de interface em aplicação *stand-alone* utilizando bibliotecas AWT ou Swing do Java.

No ServCLIPS é utilizado o acesso a banco de dados MySQL via JDBC permitindo que sejam armazenados todos os resultados de diagnóstico gerados pela ferramenta. Com base nestes dados podem ser gerados vários relatórios sobre ocorrência de doenças.

Referências Bibliográficas

APACHE SOFTWARE FOUNDATION. *JMeter - Apache Jmeter.* Disponível em: <http://jakarta.apache.org/jmeter/index.html>. Acesso em: 23 jan. 2002.

CULBERT, C.; RILEY, G.; DONNELL, B. *CLIPS:* reference manual, version 6.10. 1998. [Boston: International Thompson Publishing] ,1998. v. 2: Advanced programming guide.

FREE SOFTWARE FOUNDATION. *GNU general public license - GNU project - Free Software Foundation (FSF)* Disponível em: <http://www.gnu.org/copyleft/gpl.html>. Acesso em: 23 jan. 2002.

GIARRATANO, J. C. *CLIPS*: user´s guide, version 6.10. Boston: International Thompson Publishing ,1998. 154 p.

GIORDANO, M. *WebCLIPS home page.* Disponível em: <http://www.monmouth.com/~km2580/wchome.htm>. Acesso em: 23 jan. 2002

GOSLING, J.; JOY, B.; STEELE, G. *The Java language specification*. Reading: Addison-Wesley, 1996. 825 p. (The Java Series).

HARRIS, W. *The Java runtime interface, version 0.6.* Disponível em: <http://home.netscape.com/eng/jri/>. Acesso em: 15 jan. 2002.

HUNTER, J; CRAWFORD, W. *Java servlet programming*. Beijing: O' Reilly, 1998. 510 p. (The Java Series).

KROL, E. *The whole internet*: user's guide & catalog. Sebastopol: O'Reilly, 1993. 376 p.

MEAD, M. *Using the Java native interface with Delphi* Disponível em: <http://home.pacifier.com/~mmead/borland/borcon2001/index.html>. Acesso em: 15 jan. 2002.

MICROSOFT CORPORATION. *Comparing J/Direct to raw native interface.* Disponível em: <http://msdn.microsoft.com librarydefault .asp?url =/ library/en-us/vjcore98/html/vjcon comparingjdirecttorawnative interface.asp>. Acesso em: 15 jan. 2002.

NIEMEYER, P.; PECK, J. *Exploring Java*. 2. ed. Cambridge: O'Reilly & Associates, 1997. 594 p. (The Java Series).

RILEY, G. *CLIPS:* a tool for building expert systems. Disponível em: <http://www.ghg.net/clips/CLIPS.html>. Acesso em: 19 abr. 2001.

SUN MICROSYSTEMS, *Java native interface specification.* Disponível em: <http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html>. Acesso em: 14 set. 2001.

SUN MICROSYSTEMS. *JDK 1.1.x documentation*. Disponível em: <http://java.sun.com/products/jdk/1.1/docs/index.html>. Acesso em: 14 set. 2001.

SUN MICROSYSTEMS. *The Java tutorial.* Disponível em: <http://java.sun.com/docs/books/tutorial/index.html>. Acesso em: 18 set. 2001.

Anexos

Anexo A: Código Fonte Java da Classe JavaCLIPS

```
/**
 *    Classe que implementa a interface Java com o CLIPS (em C).
 *    Esta classe usa o modelo Singleton, pois nao existem garanti-
as
 *    de que o CLIPS esta preparado para executar em multi-thread.
 *    Neste caso somente sera permitida uma instancia de objeto
 *    da classe JavaCLIPS, a qual pode ser obtida atraves do metodo
 *    getInstance() conforme ilustrado no exemplo abaixo:
 *    <PRE>
 *          JavaCLIPS jc=JavaCLIPS.getInstance();
 *    </PRE>
 *    Os metodos definidos nesta classe correspondem a um
subconjunto
 *    de funcoes da API do CLIPS, possuindo o mesmo nome, as quais
 *    estao explicadas no manual.
 *    <PRE>
 *    CLIPS Reference Manual
 *    Volume II
 *    Advanced Programming Guide
 *    Version 6.10
 *    August 5th 1998
 *    </PRE>
 *    onde podem ser encontrados mais detalhes sobre as funcoes.
 *
 *    @version    0     10 nov 2000
 *    @author           Sergio Cruz (<A
HREF="mailto:sergio@cnptia.embrapa.br">sergio@cnptia.embrapa.br</
A>)
 *    @author           Carlos Azevedo (<A
HREF="mailto:cazevedo@cnptia.embrapa.br">sergio@cnptia.embrapa.br</
A>)
 */

public class JavaCLIPS {
      /**
       * Refer&ecirc;ncia a instancia &uacute;nica de JavaCLIPS
       */
      private static JavaCLIPS jc_unico=null;

      /**
       * codigo de Routers stdout usado no CLIPS
       * ver arquivo: javaclips.c
       */
      final static int ROUTERSTDOUT=0;
      /**
       * codigo de Routers wprompt usado no CLIPS
       * ver arquivo: javaclips.c
       */
      final static int ROUTERWPROMPT=1;
```

```
/**
 * codigo de Routers wdisplay usado no CLIPS
 * ver arquivo: javaclips.c
 */
final static int ROUTERWDISPLAY=2;
/**
 * codigo de Routers wdialog usado no CLIPS
 * ver arquivo: javaclips.c
 */
final static int ROUTERWDIALOG=3;
/**
 * codigo de Routers werror usado no CLIPS
 * ver arquivo: javaclips.c
 */
final static int ROUTERWERROR=4;
/**
 * codigo de Routers wwarning usado no CLIPS
 * ver arquivo: javaclips.c
 */
final static int ROUTERWWARNING=5;
/**
 * codigo de Routers wtrace usado no CLIPS
 * ver arquivo: javaclips.c
 */
final static int ROUTERWTRACE=6;

/**
 * Constante identificando item que pode ser depurado
 * atraves do comando Watch do CLIPS
 */
final static String FACTS="facts";
/**
 * Constante identificando item que pode ser depurado
 * atraves do comando Watch do CLIPS
 */
final static String RULES="rules";
/**
 * Constante identificando item que pode ser depurado
 * atraves do comando Watch do CLIPS
 */
final static String ACTIVATIONS="activations";
/**
 * Constante identificando item que pode ser depurado
 * atraves do comando Watch do CLIPS
 */
final static String FOCUS="focus";
/**
 * Constante identificando item que pode ser depurado
 * atraves do comando Watch do CLIPS
 */
final static String COMPILATIONS="compilations";
/**
 * Constante identificando item que pode ser depurado
 * atraves do comando Watch do CLIPS
 */
```

```
        final static String STATISTICS="statistics";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String GLOBALS="globals";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String INSTANCES="instances";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String SLOTS="slots";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String MESSAGES="messages";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String MESSAGE_HANDLERS="message-handlers";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String GENERIC_FUNCTIONS="generic-functions";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String METHOD="method";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String DEFFUNCTIONS="deffunctions";
        /**
         * Constante identificando item que pode ser depurado
         * atraves do comando Watch do CLIPS
         */
        final static String ALL="all";

    /**
     * Metodos para tratamento de ambiente de execucao do CLIPS
     */
    /**
     * Equivale a funcao InitializeEnvironment da API do CLIPS
     * Inicializacao do CLIPS
     */
        public native void InitializeEnvironment();

    /**
```

```
 * Equivale a funcao Clear() da API do CLIPS
 * Limpa memória do CLIPS. levando ao seu estado
 * inicial.
 */
      public native void Clear();

/**
 * Equivale a funcao Load da API do CLIPS
 * Carrega arquivo clp no ambiente de execucao do CLIPS.
 *
 * @param filename nome do arquivo CLP
 * @return  0 se arquivo não pode ser aberto, -1 se o arquivo
 *    foi aberto mas ocorreu erro durante a carga e 1 se o arquivo
 *    foi carregado com sucesso.
 */
      public native int Load(String filename);

/**
 * Equivalente a funcao Reset da API do CLIPS. Limpa
 * fatos, inserindo fato inicial.
 */
      public native void Reset();

/**
 * Equivalente da funcao Save da API do CLIPS.
 *
 * @param filename de arquivo CLP
 * @return true se nao ocorreu erro durante a gravacao
 *    dos dados. false caso contrario.
 */
      public native boolean Save(String filename);

/**
 * Metodos de agenda
 */
/**
 * Equivalente ao Run da API CLIPS.
 * @param runlimit numero maximo de regras que poderam ser executa-
das
 * durante a execucao. Se igual a -1 o CLIPS executara todas as
regras
 * possíveis.
 * @return numero de regras que foram executadas
 */
      public native long Run(int runlimit);

/**
 * Equivalente ao Focus da API CLIPS.
 * Muda foco do CLIPS para módulo especificado.
 * @param moduleName nome do módulo
 */
      public native int Focus(String moduleName);

/**
 * Metodos para tratamento de fatos
 */
```

```
/**
 * Equivale a funcao AssertString da API do CLIPS
 * Asserta um fato CLIPS
 * @param strfact string correspondente ao fato
 */
      public native void AssertString(String strfact);

/**
 * Equivalente a funcao LoadFacts da API do CLIPS
 * carrega um arquivo de fatos no ambiente de execucao
 * do CLIPS.
 * @param filename nome do arquivo contendo fatos CLIPS
 * @return true se carga ocorreu com sucesso, false caso
 *    contrario.
 */
      public native boolean LoadFacts(String filename);

/**
 * Equivalente a funcao LoadFactsFromString da APi do CLIPS
 * Carrega fatos na base de dados do CLIPS a partir
 * de uma string
 * @param inputstring string contendo as definições dos fatos
 *    a serem carregados
 * @return false se ocorreu erro, true caso contrario
 */
      public native boolean LoadFactsFromString(String
inputstring);

/**
 * Equivalente a funcao SaveFacts da API do CLIPS
 * Salva fatos do ambiente de execucao do CLIPS
 * em arquivo.
 * @param filename nome do arquivo onde os fatos CLIPS
 *    seão salvos.
 * @return true se salvou com sucesso, false caso
 *    contrario.
 */
      public native boolean SaveFacts(String filename);


/**
 * Metodos para tratamento de roteamentos
 */
/**
 * Inicializa Routers do CLIPS que permitem a comunicacao entre
 * a saída do CLIPS e a API JavaCLIPS.
 * Todos as saidas enviadas aos Routers CLIPS sao armazenadas
 * em buffer que são posteriormente lidos através do
 * método GetOUt.
 * ver arquivo javaclips.c
 * @see     JavaCLIPS#GetOut(java.lang.String)
 * @version 0     10 nov 2000
 * @author Sergio Cruz
 */
      public native void InitRouters();

/**
```

```
 * Recupera saídas do CLIPS armanezadas nos buffers de
 * comunicacao entre o CLIPS e API JavaCLIPS. A cada router
 * corresponde um buffer que pode ser acessado especificando
 * o seu codigo.
 * OBS: Quando a string voltada pelo CLIPS apresentar caracteres
 *    especiais (acentuados, por exemplo) a conversao de
 *    caracteres na funcao C suportando este metodo nativo
 *    nao esta sendo realizada corretamente e todos os
 *    acentos são perdidos. (ver NewStringUTF manual JNI)
 *    nestes casos eh aconselhavel utilizar o metodo
 *    byte [] GetOutByteArray(int codigorouter)
 * @see javaclips.c JavaCLIPS#GetOutByteArray(int)
 * @param codigorouter uma das constantes desta classe
 *    especificando o router de interesse.
 * @return string contendo dados enviados ao router.
 * @version 0     10 nov 2000
 * @author Sergio Cruz
 */
     public native String GetOut(int codigorouter);

/**
 * Recupera saídas do CLIPS armanezadas nos buffers de
 * comunicacao entre o CLIPS e API JavaCLIPS. A cada router
 * corresponde um buffer que pode ser acessado especificando
 * o seu codigo.
 * @see javaclips.c JavaCLIPS#GetOut(int)
 * @param codigorouter uma das constantes desta classe
 *    especificando o router de interesse.
 * @return array de bytes (caracteres C) contendo dados enviados ao
router.
 * @version 0     2 fev 2001
 * @author Sergio Cruz
 */
     public native byte [] GetOutByteArray(int codigorouter);

/**
 * Desaloca toda memoria utilizada nos buffers de comunicacao
 * entre o CLIPS e a API JavaCLIPS. Deve obrigatoriamente
 * ser executada antes do fim de execucao de aplicacoes
 * que utilizam a API JavaCLIPS
 */
     public native void CloseOut();

/**
 * Metodos para depuracao
 */
/*
 * Verifica se um determinado item do CLIPS esta
 * selecionado para watch. Os itens que podem
 * ser verificados sao as constantes definidas para
 * isto.
 *
 * @version 0     14 nov 2000
 * @author Sergio Cruz
 * @param item uma das constantes definindo tipo
 *    de item que pode ser depurados com o comando
```

```
 *    watch.
 * @return 1 se item esta selecionado para watch, 0
 *    se nao esta e -1 se item desconhecido.
 */
     public native int GetWatchItem(String item);

/**
 * Ativa item do CLIPS para
 * para watch. Os itens que podem
 * ser ativados estao definidos como constantes
 * da classe definindo os itens possiveis.
 *
 * @version 0     14 nov 2000
 * @author Sergio Cruz
 * @param item    uma das constantes definindo tipo
 *    de item que pode ser depurados com o comando
 *    watch.
 * @return true se item foi ativado com sucesso,
 *    false caso contrario.
 *    se nao esta e -1 se item desconhecido.
 */
     public native boolean Watch(String item);


/**
 * Desativa determinado item do CLIPS para
 * para watch. Os itens que podem
 * ser selecionados estao definidos como constantes
 * da classe definindo os itens possiveis.
 *
 * @version 0     14 nov 2000
 * @author Sergio Cruz
 * @param item    uma das constantes definindo tipo
 *    de item que pode ser depurados com o comando
 *    watch.
 * @return true se item foi ativado com sucesso,
 *    false caso contrario.
 */
     public native boolean Unwatch(String item);

/**
 * Metodos gerais
 */

/**
 * Deve ser chamado apos o Run para verificar o
 * estado da maquina de inferencia.
 * ver funcao de usuario CLIPS (fim) definida no
 * arquivo javaclips.c
 *
 * @version 0     14 nov 2000
 * @author Sergio Cruz
 *
 * @return true se a funcao CLIPS (fim) foi chamada durante
 *    a execucao do CLIPS, e false caso contrario.
 */
     public native boolean ehFim();
```

```
/**
 *    Comando para teste
 */
      public native void CommandLoop();

/**
 * Construtor protegido para garantir que nao podem ser
 * criadas varias instancias da classe JavaCLIPS
 *
 * @version 0     14 nov 2000
 * @author Sergio Cruz
 */
      private JavaCLIPS() {;}

/**
 * Cria uma instancia unica de objeto da classe JavaCLIPS e
 * chama metodos para inicializar o CLIPS.
 * Unico modo de acessar uma referencia de JavaCLIPS
 *
 * @version 0     14 nov 2000
 * @author Sergio Cruz
 *
 * @return  instancia unica de objeto da classe JavaCLIPS
 */
      public static JavaCLIPS getInstance() {
            if(jc_unico==null)
            {
                  jc_unico=new JavaCLIPS();
                  if(jc_unico!=null)
                  {
                        jc_unico.InitializeEnvironment();
                        jc_unico.InitRouters();
                  }
                  else
                  {
                        System.err.println("Não consegui criar
JavaCLIPS");
                  }
            }
            return jc_unico;
      }

      static {
            System.loadLibrary("clips");
      }
}
```

Anexo B: Arquivo javaclips.h contendo declaração de funções nativas implementadas na linguagem de programação "C"

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JavaCLIPS */

#ifndef _Included_JavaCLIPS
#define _Included_JavaCLIPS
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JavaCLIPS
 * Method:    InitializeEnvironment
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_InitializeEnvironment
  (JNIEnv *, jobject);

/*
 * Class:     JavaCLIPS
 * Method:    Clear
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_Clear
  (JNIEnv *, jobject);

/*
 * Class:     JavaCLIPS
 * Method:    Load
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_JavaCLIPS_Load
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    Reset
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_Reset
  (JNIEnv *, jobject);

/*
 * Class:     JavaCLIPS
 * Method:    Save
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Save
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    Run
 * Signature: (I)J
```

```
 */
JNIEXPORT jlong JNICALL Java_JavaCLIPS_Run
  (JNIEnv *, jobject, jint);

/*
 * Class:     JavaCLIPS
 * Method:    Focus
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_JavaCLIPS_Focus
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    AssertString
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_AssertString
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    LoadFacts
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_LoadFacts
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    LoadFactsFromString
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_LoadFactsFromString
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    SaveFacts
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_SaveFacts
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    InitRouters
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_InitRouters
  (JNIEnv *, jobject);

/*
 * Class:     JavaCLIPS
 * Method:    GetOut
 * Signature: (I)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_JavaCLIPS_GetOut
  (JNIEnv *, jobject, jint);
```

```
/*
 * Class:     JavaCLIPS
 * Method:    GetOutByteArray
 * Signature: (I)[B
 */
JNIEXPORT jbyteArray JNICALL Java_JavaCLIPS_GetOutByteArray
  (JNIEnv *, jobject, jint);

/*
 * Class:     JavaCLIPS
 * Method:    CloseOut
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_CloseOut
  (JNIEnv *, jobject);

/*
 * Class:     JavaCLIPS
 * Method:    GetWatchItem
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_JavaCLIPS_GetWatchItem
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    Watch
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Watch
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    Unwatch
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Unwatch
  (JNIEnv *, jobject, jstring);

/*
 * Class:     JavaCLIPS
 * Method:    ehFim
 * Signature: ()Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_ehFim
  (JNIEnv *, jobject);

/*
 * Class:     JavaCLIPS
 * Method:    CommandLoop
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_CommandLoop
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Anexo C: Implementação das funções nativas na linguagem de programação "C"

```
/*****************************************************/
/*      "C" Language Integrated Production System      */
/*                                                     */
/*            LIBCLIPS Versao 0.00  22 set 2000        */
/*                                                     */
/*              MODULO DE INTERFACE JAVA CLIPS         */
/*****************************************************/

/*************************************************************/
/* Descricao:                                                */
/*    Neste arquivo sao implementadas funcoes que            */
/*    permitem o mapeamento de metodos da linguagem          */
/*    java em funcoes CLIPS disponiveis na biblioteca        */
/*    As funcoes de mapeamento estao declaradas no arquivo   */
/*    javaclips.h o qual eh gerado automaticamente           */
/*    a partir do arquivo JavaCLIPS.java                     */
/*    Ao se incluir novos metodos "native" na classe JavaCLIPS*/
/*    deve gerar javaclips.h (geracao automatica) e corrigir  */
/*    adequadamente este arquivo de modo a refletir as       */
/*    chamadas de funcoes "C" a partir da JVM                */
/*                                                 */
/* Autor(es):                                                */
/*    Sergio Cruz (sergio@cnptia.embrapa.br)                 */
/*                                                 */
/* Revisoes:                                                 */
/*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <jni.h>
#include "javaclips.h"
#include "setup.h"
#include "sysdep.h"
#include "extnfunc.h"
#include "commline.h"
#define     TAMBLOCOMEM       512

#define NOROUTER        -1
#define ROUTERSTDOUT          0
#define ROUTERWPROMPT         1
#define ROUTERWDISPLAY        2
#define      ROUTERWDIALOG          3
#define ROUTERWERROR          4
#define ROUTERWWARNING        5
#define ROUTERWTRACE          6

//Numero de routers
#define     NUMROUTERS        7
```

```
/*
Indica estado do CLIPS apos a execucao de Run, esta flag deve ser
usada
pela funcao
CLIPS (fim) para indicar que a maquina chegou no fim de execucao
de um conjunto de regras
*/
int   ehfim=0;

//7 buffers, cada um correspondente a um ROUTER do CLIPS
unsigned char     *bufferout[NUMROUTERS];
int   bufsize[NUMROUTERS];

#if ANSI_COMPILER
int main(int,char *[]);
VOID UserFunctions(void);
#else
int main();
VOID UserFunctions();
#endif

int FindOut(char *);
void PrintOut(char *,char *);
void ExitOut(void);
int CodigoRouter(char *);

void fim();


/*************************************************************/
/* Funcao: CodigoRouter                                      */
/* Descricao:                                                */
/*    Esta funcao faz o mapeamento entre o nome de uma saida     */
/*    Logica (ROUTER) padrao do CLIPS com o indice          */
/*    correspondente    ao buffer armazenando a saida enviadas   */
/*    pelo router                                        */
/*                                                       */
/* Autor:                                                */
/*    Sergio Cruz (sergio@cnptia.embrapa.br)                 */
/*                                                       */
/* Revisoes:                                             */
/*************************************************************/
int CodigoRouter(char *LogicalName)
{
      if(strcmp(LogicalName,"stdout") == 0)
      {
            return ROUTERSTDOUT;
      }
      else if(strcmp(LogicalName,"wprompt") == 0)
      {
            return ROUTERWPROMPT;
      }
      else if(strcmp(LogicalName,"wdisplay") == 0)
```

```
      {
            return ROUTERWDISPLAY;
      }
      else if(strcmp(LogicalName,"wdialog") == 0)
      {
            return ROUTERWDIALOG;
      }
      else if(strcmp(LogicalName,"werror") == 0)
      {
            return ROUTERWERROR;
      }
      else if(strcmp(LogicalName,"wwarning") == 0)
      {
            return ROUTERWWARNING;
      }
      else if(strcmp(LogicalName,"wtrace") == 0)
      {
            return ROUTERWTRACE;
      }
      return NOROUTER;
}

/*************************************************************/
/* Funcao: FindOut                                           */
/* Descricao:                                                */
/*    Verifica se a saida do CLIPS atraves do router de nome     */
/*    LogicalName deve ser tratado pelo router "jout"        */
/*    Ver funcao AddRouter                                   */
/*                                                 */
/* Autor:                                          */
/*    Sergio Cruz (sergio@cnptia.embrapa.br)                 */
/*                                                 */
/* Revisoes:                                                 */
/*************************************************************/
int FindOut(char *LogicalName)
{
      if(strcmp(LogicalName,"stdout") == 0 ||
            strcmp(LogicalName,"wprompt") == 0 ||
            strcmp(LogicalName,"wdisplay") == 0 ||
            strcmp(LogicalName,"wdialog") == 0 ||
            strcmp(LogicalName,"werror") == 0 ||
            strcmp(LogicalName,"wwarning") == 0 ||
            strcmp(LogicalName,"wtrace") == 0)
            return 1;
      return 0;
}

/*************************************************************/
/* Funcao: PrintOut                                          */
/* Descricao:                                                */
/*    Guarda saida do CLIPS atraves do router de nome*/
/*    LogicalName no buffer correspondente                   */
/*                                                 */
/* Autor:                                          */
```

```
/*    Sergio Cruz (sergio@cnptia.embrapa.br)                    */
/*                                                    */
/* Revisoes:                                                 */
/***********************************************************/
void PrintOut(char *LogicalName,char *str)
{
      int   i;

      //Identifica codigo do router
      i=CodigoRouter(LogicalName);

      if(i==NOROUTER)
            return;

      //Aloca memoria no buffer caso esta nao seja suficiente para
      //armazenar os dados enviados pelo CLIPS

      if(bufferout[i]==NULL)
      {
            bufsize[i]=TAMBLOCOMEM;
            bufferout[i]=(char *) malloc(bufsize[i]*sizeof(char));
            bufferout[i][0]='\0';
      }

      while((strlen(bufferout[i])+strlen(str)+1)>bufsize[i])
      {
            bufsize[i]=bufsize[i]+TAMBLOCOMEM;
            bufferout[i]=(char *)
realloc(bufferout[i],bufsize[i]*sizeof(char));
      }

      //concatena saida do CLIPS ao buffer adequado
      strcat(bufferout[i],str);
}

/***********************************************************/
/* Funcao: ExitOut                                         */
/* Descricao:                                              */
/*    Libera memoria aloca nos buffers para armazenamento de     */
/*    saida do CLIPS atraves do routers                    */
/*                                                    */
/* Autor:                                                  */
/*    Sergio Cruz (sergio@cnptia.embrapa.br)                    */
/*                                                    */
/* Revisoes:                                               */
/***********************************************************/
void ExitOut()
{
      int i;

      for(i=0;i<NUMROUTERS;i++)
      {
```

```
            if(bufferout[i]!=NULL)
            {
                  free((char*)bufferout[i]);
                  bufferout[i]=NULL;
            }
      }
}


/*
Funcao Clips definida pelo usuario para indicar conclusao de re-
gras.
Em Java_JavaCLIPS ehfim eh setado em 0(FALSE) antes da execucao.
Se a funcao (fim) for chamada o valor de ehfim sera alterado para
1(TRUE);
*/
void fim() {
      ehfim=1;//TRUE
}

/*
 * Class:     JavaClips
 * Method:    InitializeEnvironment
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_InitializeEnvironment
  (JNIEnv *env, jobject thisObj)
{
      InitializeEnvironment();
}

/*
 * Class:     JavaCLIPS
 * Method:    Clear
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_Clear
  (JNIEnv *env, jobject thisObj)
{
      Clear();
}

/*
 * Class:     JavaCLIPS
 * Method:    Load
 * Signature: ()I
 */

JNIEXPORT jint JNICALL Java_JavaCLIPS_Load
  (JNIEnv *env, jobject thisObj, jstring s)
{
      jint  r;
```

```
      const char *filename = (*env)->GetStringUTFChars(env, s, 0);
      r=Load(filename);
      (*env)->ReleaseStringUTFChars(env,s,filename);
      return r;
}

/*
 * Class:     JavaCLIPS
 * Method:    Reset
 * Signature: ()V
 */

JNIEXPORT void JNICALL Java_JavaCLIPS_Reset
  (JNIEnv *env, jobject thisObj)
{
      Reset();
}

/*
 * Class:     JavaCLIPS
 * Method:    Save
 * Signature: ()I
 */

JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Save
  (JNIEnv *env, jobject thisObj, jstring s1)
{
      jboolean r;
      const char *savename = (*env)->GetStringUTFChars(env, s1, 0);
      r=Save(savename);
      (*env)->ReleaseStringUTFChars(env,s1,savename);
      return r;
}

/*
 * Class:     JavaCLIPS
 * Method:    Run
 * Signature: (J)J
 */
JNIEXPORT jlong JNICALL Java_JavaCLIPS_Run
  (JNIEnv *env, jobject thisObj, jint runLimit)
{
      ehfim=0;//FALSE
      return Run((long)runLimit);
}

/*
 * Class:     JavaCLIPS
 * Method:    Focus
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_JavaCLIPS_Focus
```

```
  (JNIEnv *env, jobject thisObj, jstring modulename)
{
      return 1;
}


/*
 * Class:     JavaCLIPS
 * Method:    AssertString
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_JavaCLIPS_AssertString
  (JNIEnv *env, jobject thisObj, jstring s2)
{
      const char *factstring = (*env)->GetStringUTFChars(env, s2,
0);
      AssertString(factstring);
      (*env)->ReleaseStringUTFChars(env,s2,factstring);
}

/*
 * Class:     JavaCLIPS
 * Method:    LoadFacts
 * Signature: (Ljava/lang/String;)Z
 */

JNIEXPORT jboolean JNICALL Java_JavaCLIPS_LoadFacts
  (JNIEnv *env, jobject thisObj, jstring s3)
{
      jboolean r;
      const char *factname = (*env)->GetStringUTFChars(env, s3, 0);
      r=LoadFacts(factname);
      (*env)->ReleaseStringUTFChars(env,s3,factname);
      return r;
}


/*
 * Class:     JavaCLIPS
 * Method:    LoadFactsFromString
 * Signature: (Ljava/lang/String;I)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_LoadFactsFromString
  (JNIEnv *env, jobject thisObj, jstring s4)
{
      jboolean r;
      const char *loadname = (*env)->GetStringUTFChars(env, s4, 0);
      r=LoadFactsFromString(loadname,-1);
      (*env)->ReleaseStringUTFChars(env,s4,loadname);
      return r;
}
/*
```

```
 * Class:     JavaCLIPS
 * Method:    SaveFacts
 * Signature: (Ljava/lang/String;I)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_SaveFacts
  (JNIEnv *env, jobject thisObj, jstring s5)
{
      jboolean r;
      const char *savename = (*env)->GetStringUTFChars(env, s5, 0);
      r=SaveFacts(savename,-1,NULL);
      (*env)->ReleaseStringUTFChars(env,s5,savename);
      return r;
}

/*
 * Class:     JavaCLIPS
 * Method:    ehFim
 * Signature: ()Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_ehFim
  (JNIEnv *env, jobject thisObj)
{
      return ehfim;
}

/*
 * Class:     JavaClips
 * Method:    CommandLoop
 * Signature: ()V
 */


JNIEXPORT void JNICALL Java_JavaCLIPS_InitRouters
  (JNIEnv *env, jobject thisObj)
{

   AddRouter("jout",10,FindOut,PrintOut,NULL,NULL,ExitOut);

}

JNIEXPORT jstring JNICALL Java_JavaCLIPS_GetOut
  (JNIEnv *env, jobject thisObj,jint codigorouter)
{

      if(codigorouter>=0 && codigorouter<=6)
      {
            if(bufferout[codigorouter]!=NULL)
            {
//Esta ocorrendo uma conversao incorreta dos caracteres:
// àáâãèéêìíîòóôõuùúû
// quando se usa diretamente (*env)->NewStringUTF(env,
bufferout[codigorouter])
```

```
//
// Abaixo esta uma tentativa de se usar caracteres extendidos do
Unix
// para tentar contornar este problema, porem sem sucesso
//
//
//                wchar_t* wcstr;
//                int   l;
//                printf("String do
CLIPS=%s",bufferout[codigorouter]);
//                l=strlen(bufferout[codigorouter]);
//                wcstr = (wchar_t*)malloc(2*l*sizeof(wchar_t));
//                mbstowcs(wcstr,bufferout[codigorouter],l+1);
//                printf("String do CLIPS W =%s",wcstr);
//                return (*env)->NewStringUTF(env, wcstr);
//
//Talvez converter estes caracteres para notacao HTML no proprio
CLIPS
//A funcao C JNIEXPORT jbyteArray JNICALL
Java_JavaCLIPS_GetOutByteArray
//deve ser usada quando os caracteres geradas pelo CLIPS possuirem
acento
//ou outros caracteres especiais
//SABC 2/2/2001

                return (*env)->NewStringUTF(env,
bufferout[codigorouter]);
            }
      }
      return (jstring)NULL;
}

/*
 * Class:     JavaCLIPS
 * Method:    GetOutByteArray
 * Signature: (I)[B
 */
JNIEXPORT jbyteArray JNICALL Java_JavaCLIPS_GetOutByteArray
  (JNIEnv *env, jobject thisObj, jint codigorouter)
{
      if(codigorouter>=0 && codigorouter<=6)
      {
            if(bufferout[codigorouter]!=NULL)
            {
                  int len = strlen(bufferout[codigorouter]);
                  jbyteArray strbyte = (*env)-
>NewByteArray(env,(jsize)len);
                  (*env)-
>SetByteArrayRegion(env,strbyte,(jsize)0,(jsize)len,(jbyte*)bufferout[codigorouter]);
                  return strbyte;
            }
      }
```

```
      return (jbyteArray)NULL;
}


JNIEXPORT void JNICALL Java_JavaCLIPS_CloseOut
  (JNIEnv *env, jobject thisObj)
{
      ExitOut();
}

/*
 * Class:     JavaCLIPS
 * Method:    GetWatchItem
 * Signature: (Ljava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_JavaCLIPS_GetWatchItem
  (JNIEnv *env, jobject thisObj, jstring jitem)
{
      jint r;
      const char *item = (*env)->GetStringUTFChars(env, jitem, 0);
      r=GetWatchItem(item);
      (*env)->ReleaseStringUTFChars(env,jitem,item);
      return r;
}

/*
 * Class:     JavaCLIPS
 * Method:    Watch
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Watch
  (JNIEnv *env, jobject thisObj, jstring jitem)
{
      jboolean r;
      const char *item = (*env)->GetStringUTFChars(env, jitem, 0);
      r=Watch(item);
      (*env)->ReleaseStringUTFChars(env,jitem,item);
      return r;
}

/*
 * Class:     JavaCLIPS
 * Method:    Unwatch
 * Signature: (Ljava/lang/String;)Z
 */
JNIEXPORT jboolean JNICALL Java_JavaCLIPS_Unwatch
  (JNIEnv *env, jobject thisObj, jstring jitem)
{
      jboolean r;
      const char *item = (*env)->GetStringUTFChars(env, jitem, 0);
      r=Unwatch(item);
      (*env)->ReleaseStringUTFChars(env,jitem,item);
```

```
      return r;
}


JNIEXPORT void JNICALL Java_JavaCLIPS_CommandLoop
  (JNIEnv *env, jobject thisObj)
{
      CommandLoop();
}



/*******************************************************/
/* UserFunctions: Informs the expert system environment  */
/*   of any user defined functions. In the default case, */
/*   there are no user defined functions. To define      */
/*   functions, either this function must be replaced by */
/*   a function with the same name within this file, or  */
/*   this function can be deleted from this file and     */
/*   included in another file.                           */
/*******************************************************/
VOID UserFunctions()
{
      DefineFunction("fim",'v',PTIF fim,"fim");
}
```

Informática Agropecuária

MINISTÉRIO DA AGRICULTURA,
PECUÁRIA E ABASTECIMENTO