JPE: UM AMBIENTE INTEGRADO DE DESENVOLVIMENTO DE SISTEMAS DE REPRESENTAÇÃO DE CONHECIMENTO E RACIOCÍNIO

João C.P. da Silva[1], Cristina T. Cerdeiral[1], Peter P. Lupo[1], Wilson P. Machado[1], Flávia Maria Santoro[2]

[1] Dept. de Ciência da Computação – IM – UFRJ ,
[2] Dept. de Informática Aplicada – UNIRIO

RESUMO

A construção de qualquer sistema de representação de conhecimento e raciocínio tem início com a definição de uma linguagem de representação não ambígua, que permita representar o conhecimento sobre certo domínio, através de um conjunto de sentenças desta linguagem. Este conjunto de sentenças é chamado base de conhecimento. Além disso, são necessárias formas bem definidas de manipular estas bases na fase de construção e utilização. Este trabalho tem como objetivo apresentar um ambiente integrado de desenvolvimento de sistemas de representação de conhecimento e raciocínio, chamado JPE (Java Prolog Environment), que pretende oferecer um conjunto pré-definido de linguagens de programação em lógica e um conjunto de ferramentas para serem utilizadas na construção e manipulação de bases de conhecimento.

1. INTRODUÇÃO

A investigação de técnicas para representação e manipulação do conhecimento é uma das áreas de pesquisa mais importantes dentro da inteligência artificial. A execução de tarefas que parecem envolver inteligência, mesmo aquelas que para nós parecem bem simples, requer uma grande quantidade de conhecimento para serem realizadas.

A construção de qualquer *sistema de representação de conhecimento e raciocínio* tem início com a definição de uma *linguagem de representação* não ambígua, que permita representar o conhecimento sobre certo domínio, através de um conjunto de sentenças desta linguagem. Este conjunto de sentenças é chamado *base de conhecimento*.

Porém, somente a definição de uma linguagem de representação e a construção de uma base a partir desta, é insuficiente para a obtenção de sistemas que se comportem de maneira inteligente. É necessária também uma forma precisa e bem definida de manipular esta base, seja na fase de sua construção ou utilização.

A área conhecida como *programação em lógica* [1] [2] [3], se desenvolveu procurando combinar a utilização da lógica como linguagem de representação e a teoria de *dedução automática* (processo pelo qual derivamos conhecimentos que se encontram implicitamente representados na base) como forma de manipulá-la. Um exemplo bem conhecido deste casamento é a linguagem Prolog [4].

A linguagem mais simples utilizada em programação em lógica, trata basicamente de problemas que envolvem conhecimento *positivo* e *definido*. Logo, se percebeu a necessidade de se estender esta linguagem, uma vez que nela não é possível representar certos tipos importantes de conhecimento como, por exemplo, conhecimento *negativo* ("*Ana não está em Paris*") e disjuntivo ("*Ana está ou em Paris ou em Londres*"). Tal aumento no poder de expressão da linguagem acarreta na necessidade de expansão dos procedimentos de prova associados às linguagens mais simples.

Além da dedução, outras formas de manipulação do conhecimento também se mostram importantes quando se pretende construir um sistema de representação e raciocínio. Por exemplo:

- *raciocínio por default* [5]: quando estamos interessados em prever o comportamento de um sistema mesmo que nosso conhecimento a cerca do mundo seja incompleto.
- *raciocínio por abdução* [6]: quando queremos usar o conhecimento expresso na base para explicar algum comportamento que está sendo observado;
- *aprendizagem (indução)* [7]: o objetivo é generalizar certos conceitos presentes na base de conhecimento, permitindo que um conjunto de fatos possa ser representado de maneira mais compacta, por exemplo, através de um conjunto de regras.
- *processo de revisão de crenças* ([8]): permitir que a base de conhecimento seja atualizada, não só através da inclusão de novas informações, mas também excluindo informações que não preservem a consistência da base.

Todos esses processos se inter-relacionam [8], [9], [10], assim como se relacionam com as diversas linguagens de programação em lógica existentes [11], [12].

Assim, este trabalho tem como objetivo apresentar uma proposta para construção de um ambiente integrado de desenvolvimento de sistemas de representação de conhecimento e raciocínio, chamado JPE (Java Prolog Environment). O JPE pretende oferecer um conjunto pré-definido de linguagens de programação em lógica e um conjunto de ferramentas para serem utilizadas na construção e manipulação de bases de conhecimento.

As ferramentas que serão disponibilizadas no JPE, além de serem utilizadas na construção de bases de conhecimento, poderão também ser incorporadas ao sistema em desenvolvimento, seja de forma individual ou pela composição, feita pelo JPE ou pelo usuário, de algumas delas. Por ser um desenvolvimento de código aberto, esperamos também incorporar ferramentas desenvolvidas por outras pessoas ou grupos.

Com o objetivo de tornar o ambiente JPE multi-plataforma, decidiu-se que a implementação será feita utilizando-se as linguagens Java [13] e SWI-Prolog [4].

Este trabalho está organizado da seguinte forma: na Seção 2, apresentaremos um protótipo chamado JPI (Java Prolog Interface), que foi o precursor do JPE; na Seção 3, descrevemos o processo inicial de modelagem do JPE; e, na Seção 4, apresentamos a conclusão e os trabalhos que serão desenvolvidos no futuro.

2. O PROTÓTIPO JPI

2.1. Arquitetura

O JPI (*Java Prolog Interface*) é um protótipo de aplicativo em Java que estabelece uma ponte entre o usuário e o mecanismo de computação do Prolog. Ele é composto por uma interface gráfica, cujas funcionalidades básicas são as mesmas de um editor de texto padrão, como edição e abertura de arquivos. Além destas, o ambiente JPI permite que, enquanto o usuário escreve seus programas, ele possa testá-los, fazendo consultas e recebendo as respostas correspondentes.

O mecanismo de inferência utilizado é o SWI-Prolog [4]. O SWI-Prolog é um software livre e de código aberto, e se trata de um interpretador que vem sendo desenvolvido na Universidade de Amsterdã. O pacote SWI-Prolog permite acesso ao núcleo do seu mecanismo de computação através de uma FLI (*Foreign Language Interface*). As funções da FLI são disponibilizadas na linguagem C ANSI. A opção por usar o SWI-Prolog ao invés de desenvolver um mecanismo de inferência próprio permitirá que, o foco do nosso trabalho seja a implementação de ferramentas que possam ser utilizadas na construção e manipulação de bases de conhecimento.

A comunicação entre a nossa interface e o SWI-Prolog é feita através do pacote JPL. Distribuído pelo grupo do SWI-Prolog, ele consiste de classes Java com métodos para comunicação com o SWI-Prolog. Através da JNI (*Java Native Interface*), o JPL porta as funções da FLI em C ANSI para o uso destas em Java. O JPL foi desenvolvido para ambientes UNIX, mas é possível utilizá-lo no ambiente MS-Windows compilando o JPL em um arquivo DLL (*Dynamic Link Library*). A seguinte figura ilustra a arquitetura do sistema:

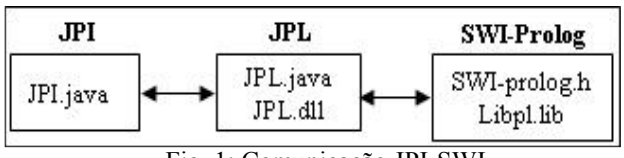


Fig. 1: Comunicação JPI-SWI

2.2. Representação de Bases de Conhecimento e Ferramenta de Abdução

Além de permitir a edição e consulta a programas Prolog a partir de seu ambiente, o JPI permite a construção e consulta de bases de conhecimento, e a utilização de uma ferramenta que implementa uma forma simples de raciocínio por abdução.

A linguagem de representação de conhecimento que o sistema reconhece tem sintaxe e semântica definidas de maneira semelhante à linguagem Prolog, sendo que os símbolos ":-" (*implicação*), "," (*conjunção*) e "*not*" (*negação por falha*) são representados, respectivamente, por "$\leftarrow$" , "*" e "$\wedge$". Uma *constante* é representada por uma seqüência de caracteres alfanuméricos começando com uma letra minúscula. Uma *variável* é uma seqüência de caracteres alfanuméricos começando com uma letra maiúscula. Um *termo* pode ser uma constante, uma variável ou $f(t_1,...,t_n)$, onde $t_1,...,t_n$ são termos ($n > 0$) e f é um *símbolo de função*. Uma *fórmula atômica* ou *átomo* é definido por $p(t_1,...,t_n)$, onde $t_1,...,t_n$ são termos ($n \geq 0$) e p é um *símbolo de predicado*. Símbolos de função e de predicado são definidos como seqüências de caracteres alfanuméricos começando com uma letra minúscula.

Uma *cláusula* é uma fórmula da forma $cabeça \leftarrow corpo_1 \wedge \ldots \wedge corpo_n$, ($n \geq 0$) onde *cabeça*, $corpo_1$, …, $corpo_n$ são átomos e representam o fato de que "se $corpo_1$, …, $corpo_n$ são verdadeiros, então *cabeça* é verdadeira". Semanticamente, uma cláusula é *falsa* sempre que seu corpo for *verdadeiro* e sua cabeça for *falsa*. Em qualquer outro caso, a cláusula é verdadeira. Quando a cláusula não possui corpo ($n = 0$), ela representa um fato que é verdadeiro.

Exemplo 2.1

Suponha que queremos representar em uma base de conhecimento as seguintes informações: sabemos que em um sistema, uma lâmpada está acessa (*lamp_on*) se nada de anormal estiver ocorrendo (*ab**). Uma anormalidade que podemos enfrentar é a falta de luz (*energia**) e sabemos que se nenhum acidente ocorreu (*acidente**), temos garantia de ter energia. Estas informações podem ser representadas por:
(i) lamp_on $\leftarrow$ ab*.
(ii) ab $\leftarrow$ energia*.
(iii) energia $\leftarrow$ acidente*. ♦

O conceito de abdução foi introduzido por C.S. Peirce [14], que distingue três formas de raciocínio: a *dedução*, a *abdução* e a *indução*. Enquanto a dedução é um processo analítico baseado na aplicação de regras gerais em casos particulares, com a inferência de um resultado, a abdução é uma forma de inferência onde o caso é inferido a partir de uma regra e de um resultado.

Dada uma base de conhecimento, inicialmente devemos definir quais são as hipóteses com as quais podemos trabalhar. No nosso caso, vamos considerar o conjunto de hipóteses *H* como sendo formado por todos os símbolos de predicado que aparecem na base, acrescidos do símbolo de negação. Os elementos de *H* são chamados *elementos abduzíveis*.

Logo, dadas uma base de conhecimento KB, o conjunto de hipótese *H* e uma sentença *g*, que corresponde a uma observação que foi feita, a tarefa de abdução pode ser caracterizada como o problema de encontrar um subconjunto $\Delta \subseteq H$, chamado de *explicação abdutiva* para *g* tal que:

(i) *g* pode ser derivado de KB $\cup$ Δ;
(ii) KB $\cup$ Δ é consistente (não podemos derivar um fato e sua negação).

Exemplo 2.1 - continuação

O conjunto *H* da nossa base de conhecimento é formado por *H* = {lamp_on*, ab*, energia*, acidente*}. Note que podemos explicar o fato de que a lâmpada está acesa supondo que nada de anormal e que nenhum

acidente está ocorrendo. Ou seja, assumindo Δ1 = {ab*, acidente*} como sendo verdadeiros.

Caso tenhamos observado que a lâmpada não está acessa (ou seja, lamp_on*), não podemos assumir mais a hipótese ab*, já que neste caso, a cláusula (i) seria falsa. Para que isso não ocorra, temos que assumir a hipótese {energia*}, que nos permitiria concluir ab. Note que ao assumir energia*, não podemos assumir também acidente*, uma vez que isso tornaria a cláusula (iii) falsa. Logo, Δ2 = {energia*}. ♦

2.3. Procedimento de Prova Abdutivo

O procedimento de prova utilizado [15] é composto de duas fases, uma *abdutiva* e outra de *consistência*. Intuitivamente, podemos considerar que na fase abdutiva, procuramos por hipóteses que expliquem algum fato, enquanto na fase de consistência, procuramos por argumentos que desacreditem tais hipóteses. Ambas utilizam como base o procedimento de resolução-SLDNF, descrito abaixo:

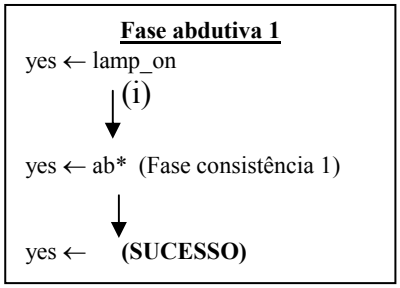
Resolver(KB, $q_1 \wedge ... \wedge q_n$)
Consulta := "yes ← $q_1 \wedge ... \wedge q_n$."
Repetir
- Selecione o elemento a_i que está mais à esquerda no corpo de Consulta;
- *Se* $a_i = b^*$
Então Se **Resolver(KB, *b*) = FALHA**
Então retire a_i do corpo(*Consulta*)
Caso contrário, retorne **FALHA**
Caso contrário,
Se existe uma cláusula *C* em KB tal que a_i = cabeça(*C*),
Então substitua a_i por corpo(*C*) em corpo(*Consulta*)
Caso contrário, retorne **FALHA.**

Até Consulta := "yes ←."
Retorne **SUCESSO.**

Na fase *abdutiva*, quando o procedimento acima encontra um elemento $a_i = b^*$, este é acrescentado ao conjunto Δ, sendo considerada uma hipótese negativa que poderia ser utilizada. Supondo que tal hipótese é assumida, entra-se na fase de *consistência*. Nesta fase, é verificada se tal hipótese mantém a consistência da base, através da chamada **Resolver(KB, *b*)**. Se o resultado obtido é **FALHA**, podemos considerar que a hipótese *b** pode ser assumida. As fases são chamadas de maneira alternada. Ou seja, caso, nesta última chamada for utilizado algum átomo da forma *b'** , é chamada uma nova fase abdutiva.

Exemplo 2.1 - continuação

A consulta lamp_on pode ser explicada por Δ = {ab*, acidente*}, cuja derivação obtida é dada por :



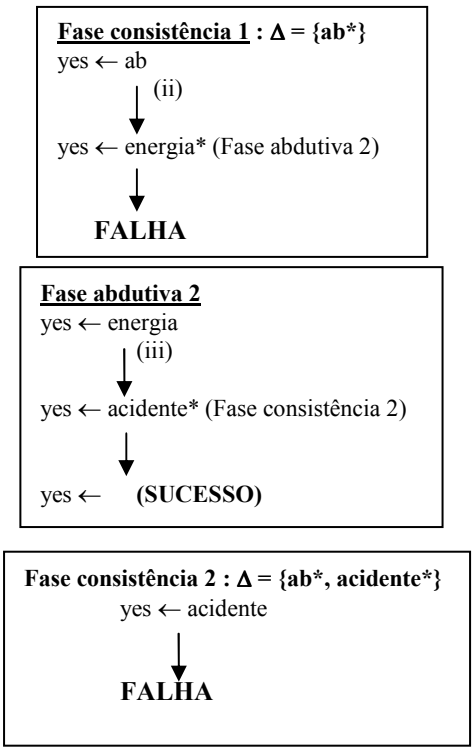


Fig. 2: Consulta abdutiva

Na fase abdutiva 1, quando *ab** é encontrado, chamamos a fase de consistência 1, para verificar se quando assumida, tal hipótese não gera inconsistência. Nesta fase, encontra-se *energia**, o que dispara uma nova fase abdutiva (2), onde verificamos se é possível obter alguma explicação para *energia*. Podemos explicar isso, supondo *acidente**, o que acarreta a chamada de uma nova fase de consistência. Nesta fase, verificamos que a hipótese *acidente**, e conseqüentemente, *ab** podem ser assumidas sem gerar inconsistência. ♦

3. JPE: REQUISITOS E ESPECIFICAÇÃO

O JPI foi desenvolvido como um protótipo para testar e avaliar as primeiras idéias de funcionamento do ambiente integrado de desenvolvimento de sistemas de representação de conhecimento e raciocínio. Uma das desvantagens de um protótipo é que o código possui baixa qualidade técnica, pois não há preocupação com uma modelagem prévia ou boas práticas de programação. No entanto, a experiência adquirida com o protótipo serviu como base para iniciar a construção do JPE. Outra grande vantagem do protótipo é confirmar a viabilidade técnica do projeto.

Para o desenvolvimento do JPE, foi definido um processo e uma série de artefatos vem sendo construída com vistas à sua especificação e implementação. Desta forma, algumas considerações são feitas a respeito da natureza deste projeto de forma justificar estas escolhas:

1- Alta rotatividade na equipe de produção, por se tratar de alunos de graduação em geral em estágio avançado no curso. Os novos alunos devem ser capazes de compreender o que já foi implementado e, gerar mais funcionalidades ou artefatos de documentação.

2- Projeto de código fonte aberto. Deve haver uma farta documentação disponível para que o código seja

facilmente entendível e outras características possam ser implementadas de acordo com a arquitetura do sistema.
3- Grande número de funcionalidades. Prevendo um crescimento quantitativo de funcionalidades, a arquitetura deve ser robusta e escalável.
4- Grande variedade de funcionalidades, sistema versátil. A arquitetura deve ser flexível para permitir que funcionalidades diferentes possam trabalhar em conjunto para finalidades muito diferentes.

Desta forma, uma larga documentação deve ser produzida, definindo exatamente o que o usuário poderá de fazer, como ele deverá fazer o que deseja como o JPE produzirá os resultados desejados. Para produzir estes artefatos organizadamente, tendo controle sobre o planejamento e histórico de modificações, fundamentos do Rational Unified Process [16] foram empregados na documentação e no processo de desenvolvimento em si. Sendo um projeto altamente modularizado, a escolha da implementação orientada a objetos foi naturalmente aceita e em seguida uma linguagem para a modelagem, também orientada a objetos foi escolhida, a UML [17].

Como o projeto se encontra no final da fase de concepção, ainda não há implementações no JPE. O processo núcleo de levantamento de requisitos deu origem a 11 requisitos. A modelagem destes requisitos ainda está em fase inicial.

3.1. Modelagem

O ato de modelar obriga um maior esforço no entendimento do problema. Um modelo permite projetar a aplicação antes de construir, permitindo assim dimensionar recursos e prazos, reduzir e simplificar a manutenção e obter aplicações de maior qualidade.

A modelagem antes do desenvolvimento evita problemas futuros, avalia a viabilidade técnica, facilita a análise de alternativas de implementação e traz soluções para problemas de arquitetura. Padrões de projeto serão utilizados para facilitar a modelagem.

3.1.1. O padrão MVC

A arquitetura MVC (Model, View, Controller - Modelo, Visualização e Controle) divide a aplicação em três camadas separadas para a manutenção dos dados, a lógica da aplicação e a interface com o usuário.

O modelo representa os dados da aplicação e as regras que governam o acesso e a modificação dos dados. O controlador define o comportamento da aplicação. É ele que interpreta as ações do usuário e as mapeia para chamadas do modelo. A visualização é responsável por apresentar as informações contidas no modelo ao usuário, sem se preocupar em como obter estas informações, que é atribuição do controlador.

As vantagens do modelo MVC são permitir que múltiplos visualisadores usem a mesma aplicação e forneçam interfaces diferentes com o usuário. Assim, será possível que uma aplicação local se torne uma web, sem que pra isso, tenha que se implementar as outras duas camadas. A manutenção se torna mais simples, pois o sistema se torna mais modularizado além de permitir que o desenvolvimento das camadas seja paralelo, já que elas têm relativa independência.

3.1.2. Padrões de Projeto (Design Patterns)

Padrões de projeto [18] são soluções genéricas e reutilizáveis para problemas bem conhecidos. Permitem obter as características de reuso e outras qualidades associadas com a orientação a objetos. Soluções que funcionaram e foram aprimoradas com o passar do tempo tornaram-se receitas para situações similares.

Os padrões de projeto são ferramentas escaláveis e de fácil implementação que tornam a arquitetura do sistema flexível e ao mesmo tempo robusta. Os padrões de projeto se dividem em criacionais (que solucionam problemas de criação de objetos), estruturais (que solucionam problemas da integração dos objetos) e comportamentais (que solucionam problemas de interação entre os objetos). Através dos padrões de projeto pretende-se resolver ainda na fase de modelagem a maior parte dos problemas e os problemas mais complexos, que interferem diretamente e vão acabar por definir a arquitetura do sistema.

3.2. Garantindo o funcionamento sem erros

Um dos grandes desafios deste projeto é garantir que durante o desenvolvimento, nenhum erro seja inserido. Este desafio se dá devido ao grande número de pessoas modificando e acrescentando código ao mesmo sistema sem necessariamente estarem em contato direto. Portanto, será adotada a abordagem de testes automatizados. Esta é uma prática que tem ganhado força graças à metodologia Extreme Programming [19].

Os testes automatizados são testes de caixa-preta, isto é, não são feitos de acordo com a implementação da característica que vai ser testada e sim com a sua interface, permitindo assim que a implementação possa ser modificada conforme o necessário. Sempre que uma modificação precisar ser feita em um trecho de código, os testes devem ser executados em todo o sistema.

Os testes se dividem basicamente em três tipos: testes de unidade (testam cada componente da aplicação, são mais constantes e detectam erros geralmente, de implementação interna); testes de integração (testam o funcionamento em conjunto de unidades e os erros detectados geralmente são relacionados com as interfaces das unidades) e testes de aceitação (testam o funcionamento do ponto de vista do usuário ou a execução adequada da funcionalidade).

Os testes irão resolver uma série de problemas inerentes ao formato do desenvolvimento desta aplicação de forma adequada e extremamente produtiva, conferindo aos desenvolvedores segurança para implementar novas funcionalidades ou melhorar o código já escrito, garantindo a sua qualidade.

3.3. Estágio atual

Atualmente, a aplicação está na fase de concepção. Isso significa que a grande maior parte do esforço até agora tem sido realizado nos requisitos e não da modelagem.

Após esta interação entre o usuário e o sistema ter sido definida em casos de uso a partir dos requisitos, cada caso de uso será modelado de acordo com o padrão MVC e padrões de projeto adequados para resolver os problemas conforme forem aparecendo.

Cada caso de uso terá, portanto, um controle de histórico de modificações (versões), uma descrição do fluxo de comunicação entre o usuário e a aplicação e ainda, diagramas UML da modelagem do sistema.

Atualmente o sistema já conta com 21 casos de uso, definindo os diálogos entre o usuário e a aplicação para todas as funcionalidades neles definidas.

3.3.1. Requisitos do sistema

Os requisitos levantados no estágio atual são:

- Utilizar provadores e predicados instalados no JPE.
- Criar provadores e predicados.
- Instalar novos provadores e predicados.
- Carregar bases de conhecimento e fazer consultas, utilizando predicados e provadores instalados.
- Armazenar programas ou partes de programas para utilizar em novos programas.
- Visualizar a árvore de derivação.
- Complemento automático de código e coloração sintática.
- Inserção de *plug-ins* que podem estar escritos em Java ou em Java e Prolog.
- Exportação de uma biblioteca jar para ser utilizada em sistemas que necessitem de uma parte de aplicação em Prolog.

Estes requisitos mostram uma grande versatilidade do sistema, que no futuro tornar-se-á uma poderosa ferramenta de desenvolvimento em Prolog.

4. CONCLUSÕES E TRABALHOS FUTUROS

O problema da construção de sistemas de representação de conhecimento e raciocínio tem basicamente duas questões relacionadas: a definição de uma linguagem de representação que permita a criação de bases de conhecimento e mecanismos de manipulação destas bases nas fases de construção de utilização.

Este artigo apresentou a proposta de um ambiente integrado para apoio ao desenvolvimento de sistemas de representação de conhecimento e raciocínio, chamado JPE (Java Prolog Environment). Um protótipo foi construído e testado, apontando os requisitos e funcionalidades deste ambiente.

Como suporte ao desenvolvedor, o JPE pretende disponibilizar através de sua interface, as seguintes facilidades: um conjunto pré-definido de linguagens de programação em lógica e um conjunto de ferramentas para serem utilizadas na construção e manipulação de bases de conhecimento.

As próximas etapas deste projeto serão a finalização da modelagem e implementação do JPE. Posteriormente serão realizados testes de validação com usuários.

5. REFERÊNCIAS

[1] P. Hayes, *Computation and Deduction*, Proc. of the 2nd Symp. On Math. Found. of Computer Science, pp.105-118, (1973).

[2] R. Kowalski, "Predicate Logic as a Programming Language", *Inf. Proc.* 74, pp. 569-574 (1974).

[3] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, NY, (1984).

[4] URL : http://www.swi-prolog.org/

[5] R. Reiter, *A logic for default reasoning*, Artif. Intell. v.13, pp. 81-132, (1980).

[6] A.C., Kakas, R.A. Kowalski and F. Toni, "The role of abduction in logic programming", *Handbook of logic in Art. Int. and Logic Prog. 5*, Oxford Univ. Press, pp. 235-324, (1998)

[7] D. Gabbay, D. Gillies, A. Hunter, S. Muggleton, Y. Ng and B. Richards, "The rule-based systems project : using confirmation theory and non-monotonic logics for incremental learning", in *Inductive Logic Prog.*, Ed. S. Muggleton, pp. 213-229. (1990)

[8] P. Gärdenfors, *Knowledge in flux: modeling the dynamics of epistemic states*, The MIT Press, Bradford Books, (1988).

[9] J. Doyle, "Reason maintenance and belief revision : foundations versus coherence theories", Belief Revision, Ed. P. Gärdenfors, Cambridge Tracts in Theor. Comp. Sci. 29, pp.29-51, (1992).

[10] D. Makinson and P. Gärdenfors, *Relations between the logic of theory change and nonmonotonic logic*, The Logic of Theory Change, LNAI 465, Springer-Verlag, pp. 185-205, (1991).

[11] C. Baral and M. Gelfond, "Logic Programming and Knowledge Representation", *Journal of Logic Prog.*, v. 19/20, pp. 73-148, (1994).

[12] W. Marek and M. Truszczynski, *Revision specifications by means of revision program*, in Proc. of JELIA'94. LNAI. Springer-Verlag, (1994).

[13] URL : http:// java.sun.com/

[14] C.S. Peirce, *Collected papers of Charles Sanders Peirce,*v.2, 1931-1958, Hartshorn et al. Eds. Harvard University Press.

[15] K. Eshghi and R.A. Kowalski, "Abduction compared woth negation by failure", *Proc. 6th Int. Conf. on Logic Prog.*, pp. 234-255, (1989)

[16] P. Kruchten, *The Rationa Unified Process An Introduction Second Edition*, Addison-Wesley, 2000

[17] G. Booch, I. Jacobson and J. Rumbaugh. *Unified Modeling Language™ (UML®)*, http://www.omg.org/cgi-bin/doc?formal/03-03-01, version: 1.5, 03-03-2001

[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley, 1994

[19] V. M. Teles, *Extreme Programming*, novatec, 2004.