Referências Bibliográficas

BENTO, J.; FEIJÓ, B. **An agent-based paradigm for building intelligent CAD Systems**. Artificial Intelligence in Engineering 11, p. 231-244 ,1997.

BRAMWELL, C. J. **Formal development methods for interactive systems: Combining interactors and design rationale**. PhD Thesis, Department of Computer Science, University of York, September,1995.

BRAMWELL, C.; FIELDS, B.; HARRISON, Michael. **Exploring Design Options Rationally**. DSVIS '95, p. 134-148, 1995.

BURGE, J.; BROWN, D. C. **Rationale Support for Maintenance of Large Scale Systems**. In: Workshop on Evolution of Large Scale Industrial Software Applications (ELISA), ICMS'03, Amsterdam, NL, 2003.

_____. **An Integrated Approach for Software Design Checking Using Rationale**. Design Computing and Cognition, Kluwer Academic Publishers, p. 557-576, 2004.

BUTLER, M. A. **CSP approach to Action Systems**. PhD Thesis, Oxford University Computing Laboratory, 1992.

CONKLIN, J. **Seven Years of Industrial Strength CSCA in an Electric Utility. Computer-Supported Collaborative Argumentation for Learning Communities**. In: CSCL'99 Workshop, Stanford University, 1999. Disponível em http://kmi.open.ac.uk/people/sbs/csca/cscl99/papers.html, Acesso em 28/12/2005.

CONKLIN, J.; BEGEMAN, M. L. **gIBIS: A Hypertext Tool for Exploratory Policy Discussion**. ACM Transactions on Office Information Systems, 6(4), p. 303-331, 1988.

CONKLIN, J.; SELVIN, A.; BUCKINGHAM, S. S.; SIERHUIS, M. **Facilitated Hypertext for Collective Sensemaking: 15 Years on from gIBIS**. In: Proceedings LAP'03: 8th International Working Conference on the Language-Action Perspective on Communication Modelling, Tilburg, The Netherlands, 2003.

DOAN, A.; MADHAVAN, J. DOMINGOS, P.; HALEVY, A. **Learning to Map between Ontologies on the Semantic Web**. In: Proceedings of the 11th World Wide Web Conference (WWW2002), Honolulu, Hawaii, USA, 2002.

FISCHER, G.; MCCALL, R. **JANUS: integrating hypertext with a knowledge-based design environment**. In: Proceedings of the second annual ACM conference on Hypertext and Hypermedia; Pittsburgh, United States, p. 105-117, 1989.

GARCIA, A. C. B. **Active Design Documents: A New Approach for Supporting Documentation in Preliminary Routine Design**. PhD Thesis, Department of Civil Engineering, Stanford University, August, 1992.

GARCIA, A. C. B.; SOUZA, C.S. **Add+: Including rhetorical structures in active documents**. Artif. Intell. Eng. Design, Analysis and Manuf., 11(2), p. 109-124, 1997.

GOEL, V.; PIROLLI, P. **Motivating the Notion of Generic Design within Information Processing Theory: The Design Problem Space**. AI Magazine 10, p. 19-36, 1989.

GRUBER, T. R. **A Translation Approach to Portable Ontologies**. Knowledge Acquisition, No. 5, p. 199-220, 1993.

GULLICHSEN, E.; D'SOUZA, D.; LINCOLN, P.; KHE-SING, T. **The PlaneText Book**. MCC Tech. Report STP-333-86(P), 1986.

HOARE, C. A. R. **Communicating Sequential Processes**. Prentice-Hall, ISBN: 0131532898, 1985.

HORDIJK, W.; KRUKKERT, D.; WIERINGA, R. **The impact of architectural decisions on quality attributes of enterprise information systems: a survey of the design space**. University of Twente, TR-CTIT-04-48, 2004.

HUBKA, V.; EDER, E. W. **Design science: Introduction to needs, scope and organization of engineering design knowledge**. 2d ed. Great Britain: Springer-Verlag London Limited, 1996.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. **The Unified Software Development Process**. Massachusetts: Addison Wesley, 1999. 463p. ISBN 0-201-57169-2.

KIFER, M.; LAUSEN, G. **F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme**. ACM SIGMOD, p. 134-146, 1989.

KIFER, M.; LAUSEN, G.; WU, J. **Logical foundations of object-oriented and frame-based languages**. Journal of the ACM, 42:741-843, July, 1995.

KUNZ, W.; RITTEL, H. W. J. **Issues as Elements of Information Systems**. Institute of Urban and Regional Development Working Paper 131, University of California, Berkeley, CA, 1970. Disponível em http://www-iurd.ced.berkeley.edu/pub/WP-131.pdf, Acesso em 29/09/2004.

LACAZE, X. **Conception rationalisée pour les systèmes interactifs – Une notation semi formelle et un environnement d'édition pour une modélisation des alternatives de conception**. These de Doctorat de l'Université Toulouse I, Juin, 2005.

LEE, J. **Design Rationale Systems: Understanding the Issues**, IEEE Expert Volume 12, No. 13, p. 78-85, 1997.

_____. **Extending the Potts and Bruns Model for Recording Design Rationale**. In: Proceedings of the 13th International Conference on Software Engineering, Austin, TX, p. 114-125,1991.

_____. **SIBYL: A Tool for Managing Group Design Rationale**. Computer Supported Cooperative Work: ACM Press: New York, p. 79-92, 1990.

LEE, J.; LAI, K. **What's in Design Rationale**. Human-Comput. Interaction, No. 6 (3-4), p. 251-280, 1991a.

_____. **A comparative analysis of design rationale representations**. CCS Tech. Report No. 121, Cambridge, MA: MIT, Center for Coordination Science, 1991b.

LIMA, F.; SCHWABE, D. **Application Modeling for The Semantic Web**. In: Proceedings of the LA Web 2003, IEEE-CS Press.

LISKOV,B.; GUTTAG, J. **Abstraction and Specification in Program Development**, MIT Press Cambridge, MA, 1986.

MACLEAN, A.; Young, R.; Bellotti, V. and Moran, T. **Questions, Options, and Criteria: Elements of Design Space Analysis**. Human-Computer. Interaction, No. 6 (3-4), p. 201-250, 1991.

MCCALL, R. J. **PHI: A conceptual foundation for design hypermedia**. Design Studies, No.12 (1), p. 30-41,1991.

_____. **PHIDIAS: A PHI-based Design Environment integrating CAD Graphics into Dynamic Hypertext**. In: Proceedings of European Conference on Hypertext, 1990.

MANN, W. C.; THOMPSON, S. A. **Rhetorical Structure Theory: A theory of text organization**. In Livia Polanyi, editor, The Strcture of Discourse. Ablex Publishing Company, Norwood, N J, p. 85-96, 1987.

MEDEIROS, A. P.; SCHWABE, D.; FEIJÓ, B. **Kuaba Ontology: Design Rationale Representation and Reuse in Model-Based Designs**. In: Proceedings of the 24th International Conference on Conceptual Modeling (ER2005), Klagenfurt, Austria, Lecture Notes in Computer Science 3716, Springer-Verlag, ISBN 3-540-29389-2, p. 241-255, 2005a.

_____. **A Design Rationale Representation for Model-Based Designs in Software Engineering**. In: Proceedings of the 17th Conference on Advanced Information Systems Forum (CAiSE'05 Forum), Porto, Portugal, FEUP, ISBN 972-752-078-2, p.163-168, 2005b.

MOURA, I.C.R. **Um ambiente para o Suporte ao Projeto e Implementação de Sistemas de Informação baseados na WWW**, Dissertação de Mestrado, Departamento de Informática PUC-Rio, 1999.

NILSSON, N. **Principles of Artificial Intelligence**. Morgan Kaufman Publishers, 1986. 476p.

NOY, N. F.; SINTEK, M.; DECKER, S.; CRUBÉZY, M.; FERGERSON, R. W. and MUSEN, M. A. **Creating Semantic Web Contents with Protégé-2000**. IEEE Intelligent Systems Vol. 16, No 2, Special Issue on Semantic Web, p. 60-71, 2001.

NUNES, D. **HyperDE - um Framework e Ambiente de Desenvolvimento dirigido por Ontologias para Aplicações Hipermídia**. Dissertação de Mestrado, Departamento de Informática PUC-Rio, 2005.

OMG: **Unified Modeling Language Specification**. Version 1.5. March, 2003.

PENA-MORA, F. **Design Rationale for computer supported conflict mitigation during the design-construction process of large-scale civil engineering systems**. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1994.

PENA-MORA, F.; VADHAVKAR, S. **Augmenting Design Patterns with Design Rationale**. Artificial Intelligence for Engineering Design, Analysis, and Manufacturing, p. 93-108, 1996.

POTTS, C.; BRUNS, G. **Recording the Reasons for Design Decisions**. In: Proceedings of 10th International Conference on Software Engineering, Singapore, p. 418-427,1988.

SCHÖN, D. **The Reflective Practitioner: How Professionals Think in Action**. New York: Basic Books, 1983.

SCHWABE, D.; ROSSI, G. **An object-oriented approach to Web-based application design**. Theory and Practice of Object Systems (TAPOS), p. 207-225,1998.

SHUM, S. J. **A Cognitive Analysis of Design Rationale Representation**. Doctoral Dissertation, Department of Psychology, University of York, UK, 1991. Disponível em http://kmi.open.ac.uk/people/sbs/phd/phd.html, Acesso em 30/08/2005.

SIMON, H. A. **The Architecture of Complexity**. Sciences of the Artificial, 2d ed., Cambridge, Mass.: MIT Press, 1981.

TOULMIN, S. **The Uses of Argument**. Cambridge University Press, Cambridge, 1958.

W3C. **OWL Web Ontology Language Overview**. W3C Recommendation, February, 2004.

VAREJAO, F. M.; GARCIA, A. C. B.; SOUZA, C. S. **ADD-GHS: A Structured Annotations Based Extension for Active Design Documents**. Rio de Janeiro: Departamento de Informática da PUC-Rio, Monografias em Ciência da Computação da PUC-Rio - MCC38,1996a.

_____. **Aquisição de Design Rationale através de Anotações Semi-Formais em Documentos Ativos de Design**. Rio de Janeiro: Departamento de Informática da PUC-Rio, Monografias em Ciência da Computação da PUC-Rio – MCC39, 1996b.

WINOGRAD, T. **Bringing Design to Software**. Addison-Wesley, 1996. 310p. ISBN: 0-201-85491-0.

YANG, G.; KIFER, M.; ZHAO, C.; CROWDHARY, V. **Flora-2: User's Manual**, Version 0.94 (Narumigata), 2005 Disponível em http://flora.sourceforge.net/documentation.php, Acesso em 06/02/2006.

Apêndice A - Resumo da Sintaxe de Flora-2 usada na Tese

Flora-2 é uma linguagem de base de conhecimento orientada a objetos sofisticada e uma plataforma de desenvolvimento de aplicação. Esta linguagem foi desenvolvida vários anos após a publicação dos trabalhos iniciais sobre F-logic e, assim, se beneficia da experiência obtida no uso e implementação da lógica. Esta experiência resultou na introdução de algumas mudanças na sintaxe (e em algum nível também na semântica) da linguagem F-logic. As diferenças relevantes para esta tese são enumeradas abaixo. Informações mais detalhadas podem ser encontradas no manual do usuário da linguagem Flora-2, disponível em (Yang et al., 2005).

1. Flora-2 usa vírgula (“,”) para separar métodos em F-moléculas. A versão da linguagem F-logic apresentada em (Kifer et al., 1995) usa ponto e vírgula (“;”). Em Flora-2, “;” representa disjunção. É também possível usar “*and*” ao invés de “,” e “*or*” ao invés de “;”.

2. A relação de subclasse representada por “::” não é reflexiva na linguagem Flora-2. Segundo seus autores, o uso não-reflexivo de “::” é um idioma mais comum em representação de conhecimento.

3. Na versão de F-logic apresentada em (Kifer et al., 1995), tipos são sempre herdáveis, mas valores não. Flora-2 usa “*=>” e “*=>>” para tipos herdáveis e “=>” e “=>>” para tipos não-herdáveis. F-logic usa apenas “=>” e “=>>”, e ambos são herdáveis.

4. A sintaxe a[b=>>{c,d}] de F-logic, que estabelece que o tipo retornado pelo atributo b é a interseção das classes c e d, não é permitida em Flora-2. Neste caso, deve-se usar a sintaxe a[b=>>(c,d)]. As sintaxes a[b=>>(c;d)], a[b=>>(c-d)] e combinações destes operadores sobre tipo também são permitidas.

5. Em Flora-2, usa-se as sintaxes “classe[método=>()]” e “classe[método=>>()]”, ao invés de “classe[método=>{}]” e “classe[método=>>{}]”.

6. Igualdade (o predicado “:=:”) é implementado apenas parcialmente na linguagem Flora-2. A principal limitação é que o axioma de congruência para igualdade (substituição por iguais) funciona apenas no nível mais alto e no primeiro nível de “aninhamento”. Para níveis mais profundos de “aninhamento”, substituição por iguais não tem sido implementada.

Programas em Flora-2

De forma semelhante à linguagem F-logic, um programa em Flora-2 pode conter diferentes definições referentes a um vocabulário, a uma base de fatos e a um conjunto de regras. Geralmente, um programa em Flora-2 é armazenado em um arquivo com extensão “.flr”. Um programa pode ser lido e compilado em Flora-2 usando-se um dos comandos abaixo.

```
?- [arquivo_programa].
                ou
? – flLoad arquivo_programa.
```

Pelo padrão, todos os programas escritos em Flora-2 são lidos em um módulo chamado “*main*”, mas é possível ler estes programas em outros módulos, usando um dos comandos abaixo. Uma vez que um programa é lido em um módulo, é possível formular consultas e invocar métodos dos objetos definidos no programa.

```
?- [arquivo>>modulo].
                ou
? – flLoad arquivo>>modulo.
```

Quando o comando *flLoad* lê uma base de regras em um módulo, ele primeiro descarta todas as regras e fatos que anteriormente formavam a base de conhecimento daquele módulo. Para adicionar os fatos e regras contidos em um certo arquivo a uma base de conhecimento já existente em um módulo, usa-se o comando *flAdd*. Este comando não apaga a base de conhecimento antiga no módulo em questão. É possível usar a sintaxe [...] prefixando o nome do arquivo com um sinal “+”, como mostra os exemplos abaixo.

```
?- [+arquivo].
?- [+arquivo>>modulo].
?- flAdd arquivo.
?- flAdd arquivo>>modulo.
```

Um módulo em Flora-2 é uma abstração de programação que permite que um grande programa seja dividido em bibliotecas separadas que podem ser reusadas de diversas maneiras no mesmo programa. Formalmente, um módulo é um par que consiste de um *nome* e um *conteúdo*. O nome deve ser um símbolo alfanumérico, e o conteúdo consiste de um código de programa que é tipicamente lido de algum arquivo (mas pode ser construído dinamicamente pela inserção de fatos dentro de um módulo).

A idéia básica por trás da modularização de Flora-2 é que bibliotecas de código reusáveis sejam gravadas em arquivos separados. Para usar uma biblioteca, ela deve ser lida dentro de um módulo. Outras partes do programa podem então invocar os métodos desta biblioteca fornecendo o nome do módulo. Não é necessário exportar nada de uma biblioteca – qualquer método público ou predicado pode ser chamado por outras partes do programa. Desta forma, a biblioteca lida dentro de um módulo torna-se o conteúdo daquele módulo. Note que não existe, a priori, associação entre módulos e nomes de arquivos. Em Flora-2, módulos são completamente desacoplados de nomes de arquivos. Um programa Flora-2 conhece apenas os nomes dos módulos que ele precisa chamar, mas não os nomes dos arquivos.

Flora-2 define três tipos de módulos. O tipo descrito acima é apenas um dos três: o módulo de usuário. Como explicado anteriormente, estes módulos são desacoplados do código real, e assim eles podem conter diferentes códigos em diferentes momentos. O próximo tipo é um módulo Prolog. Este tipo é uma abstração em Flora-2 que é usada para chamar predicados Prolog. Módulos Prolog são estáticos e considerados fortemente acoplados com seu código. O terceiro tipo de módulos descreve os módulos de sistema de Flora-2. Estes módulos são lidos juntamente com programas Flora-2 e fornecem métodos e predicados úteis, que permitem aos programas escritos pelo usuário realizar ações comuns usando nomes de predicados e métodos implementados nestes módulos.

Considerando que *literal* é uma F-molécula ou um predicado definido em um outro módulo do usuário, ele pode ser chamado usando a seguinte sintaxe:

```
literal @ modulo
```

No exemplo abaixo, podemos recuperar as informações sobre os filhos de Maria, que estão disponíveis no módulo chamado “família”.

```
?- maria[filhos->>X]@familia.
```

Alterações na Base de Conhecimento

A linguagem Flora-2 também possui primitivas para alterar a base de conhecimento em tempo de execução. Diferente de Prolog, Flora-2 não requer que o usuário defina um predicado como dinâmico para alterá-lo. Ao invés disto, todo predicado e todo objeto têm uma *parte base* e uma *parte derivada*. As alterações mudam apenas as partes base e apenas indiretamente as partes derivadas. A parte base de um predicado ou de um objeto contém ambos os fatos que foram inseridos explicitamente dentro da base de conhecimento e os fatos que o usuário especificou em um programa.

As primitivas usadas para realizar alterações podem ser classificadas em Flora-2 como primitivas para alterações que podem ser desfeitas e primitivas para alterações que não podem ser desfeitas em caso de falha durante sua execução. Neste resumo, descrevemos apenas algumas primitivas para alterações que não podem ser desfeitas em caso de falha na execução. Estas primitivas foram escolhidas para a implementação das operações apresentadas nesta tese, uma vez que as primitivas que permitem desfazer as alterações não funcionaram de forma adequada devido a algumas falhas no sistema XSB.

Neste trabalho usamos as seguintes primitivas para apoiar alterações que não podem ser desfeitas em caso de falha: *insert*, *insertall*, *delete* e *deleteall*. Estas primitivas usam uma sintaxe especial (chaves) e não são predicados. Assim, é permitido ter um predicado definido pelo usuário com o nome de “*insert*”. A sintaxe para estas primitivas é mostrada abaixo:

```
operação { literais [ | consulta] }
```

Operação especifica uma das primitivas citadas acima. A parte de *literais* representa uma lista de literais separada por vírgula e pode incluir predicados e F-moléculas. A parte opcional, “| *consulta*” é uma condição adicional que deve ser satisfeita para que os literais possam ser inseridos ou removidos (dependendo da

operação). A semântica é que a consulta seja avaliada primeiro e, se for satisfeita, os *literais* sejam inseridos ou removidos. Note que a consulta pode afetar a atribuição de valores a uma variável e assim a instância particular de literais que será inserida ou removida. Por exemplo, nos comandos abaixo a primeira operação insere uma molécula particular. No segundo caso, a consulta "S[esposa->P]" é avaliada e uma resposta (um valor para P e S) é obtida. Se o valor não existe, nada é inserido e o comando falha. Do contrário, a instância de P[esposo->S] é inserida para o valor retornado e o comando é executado com sucesso.

```
?- insert{p(a), maria[esposo->manoel, filhos->>carlos]}.
?- insert{P[esposo->S] | S[esposa->P]}.
```

O comando *insert* tem duas formas: *insert* e *insertall*. A diferença entre as duas é que *insert* insere apenas uma instância de *literais* que satisfaz a fórmula, enquanto *insertall* insere todas as instâncias que satisfaçam a fórmula. No entanto, esta diferença é um pouco sutil, uma vez que o resultado da execução destas primitivas depende do modo de execução usado na linguagem Flora-2. No modo de execução "todas as respostas", comandos usando *insert* ou *insertall* irão se comportar exatamente da mesma forma, porque Flora-2 tentará encontrar todas as respostas para a consulta informada e fará a inserção para cada resposta obtida, independente da primitiva usada. A diferença se torna aparente se Flora-2 está no modo "uma resposta a cada momento" (ou seja, o comando *flOne* foi executado em um consulta anterior) ou quando o módulo "todas as respostas" é suprimido por um comando *cut*. Neste caso, o comando usando a primitiva *insert* irá inserir apenas um fato, enquanto um comando usando *insertall* irá inserir todos. Vale notar que, ao contrário de Prolog, Flora-2 trata a base de conhecimento como um conjunto de fatos sem duplicidades. Assim, a inserção de um fato que já existe na base de conhecimento não terá efeito.

As primitivas *delete* e *deleteall* possuem uma sintaxe semelhante às primitivas para inserção. Em um comando usando a primitiva *delete*, se existe mais de um valor ou combinação para os *literais* a serem removidos, a primitiva *delete* irá escolher apenas um deles de forma não determinística e remove-lo. No entanto, como na inserção, no modo de execução "todas as repostas" a remoção será realizada para cada valor possível que satisfaz a condição descrita na *consulta*.

Para evitar isto, deve-se usar o modo “uma resposta a cada momento” ou o comando *cut*.

Ao contrário de *delete*, a primitiva *deleteall* tenta remover todos os valores e combinações possíveis. Ou seja, para cada valor de variável retornado pela *consulta* esta primitiva remove a instância correspondente de *literal*. Se a combinação “*consulta* ^ *literal*” é falsa, *deleteall* falha.

Apêndice B - Especificação da Ontologia Kuaba em Flora-2

Vocabulário

```
/*****************************************************************
** File:    kuaba_ontology.flr
** Author:  Adriana P. de Medeiros
** Contact: adri@inf.puc.rio.br
**
** Copyright (C) Adriana P. de Medeiros, 2005
**
*****************************************************************/

//===============================================================
//  SCHEMA DATA: CLASSES & SIGNATURES
//===============================================================

atomic_artifact::artifact.
composite_artifact::artifact.
question::reasoning_element.
idea::reasoning_element.
argument::reasoning_element.

method[hasName=>string,
       hasDescription=>string,
       aggregates=>>activity,
       prescribes=>>formal_model].

formal_model[hasName=>string,
             hasLocalization=>string,
             describes=>>artifact,
             isPrescribedBy=>>method].

activity[hasName=>string,
         hasDescription=>string,
         hasStartDate=>string,
         hasFinishDate=>string,
         hasExpectedDuration=>expected_duration,
         involves=>>reasoning_element,
         requires=>>role,
         isExecutedBy=>>person].

expected_duration[hasAmount=>integer,
                  hasUnitTime=>string].

artifact[hasName*=>string,
         hasDescription*=>string,
         hasCreationDate*=>string,
         hasUrl*=>string,
         isResulted*=>>idea,
         isCreatedBy*=>>person,
         isDescribedBy*=>formal_model].

atomic_artifact[isVersionOf=>atomic_artifact].

composite_artifact[compositionFunction=>string,
                   compositionOf=>>artifact,
                   isVersionOf=>composite_artifact].
```

```
relation_type[hasOriginArtifact=>artifact,
              hasDestinationArtifact=>>artifact,
              hasType=>string].

reasoning_element[hasText*=>string,
                  hasCreationDate*=>string,
                  isInvolved*=>activity,
                  suggests*=>>question,
                  isPresentedBy*=>person,
                  isDefinedBy*=>formal_model].

question[hasType=>string,
         isAddressedBy=>>idea,
         isSuggestedBy=>>reasoning_element,
         isVersionOf=>question,
         hasDecision=>>decision].

idea[address=>>question,
     hasArgument=>>argument,
     results=>artifact,
     isVersionOf=>idea,
     isConcludedBy=>>decision].

argument[inFavorOf=>>idea,
         objectsTo=>>idea,
         considers=>question,
         isVersionOf=>argument].

decision[isAccepted=>boolean,
         hasDate=>string,
         isMadeBy=>>person,
         hasJustification=>justification,
         concludes=>idea].

justification[hasText=>string,
              isDerivedOf=>>argument].

person[hasName=>string,
       presents=>>reasoning_element,
       creates=>>artifact,
       makes=>>decision,
       hasE_mail=>>string,
       hasAddress=>string,
       hasTelephone=>>string,
       performs=>>role,
       executes=>>activity].

role[hasName=>string,
      isPerformedBy=>>person,
      isRequiredBy=>>activity].
```

Regras

```
/*****************************************************************
** File:    kuaba_rules.flr
** Author:  Adriana P. de Medeiros
** Contact: adri@inf.puc.rio.br
**
** Copyright (C) Adriana P. de Medeiros, 2005
**
*****************************************************************/

//===============================================================
//  RULES (INVERSE RELATIONS)
//===============================================================

X[involves->>{Y}] :- Y[isInvolved->X].
X[isInvolved->Y] :- Y[involves->>{X}].

X[address->>{Y}] :- Y[isAddressedBy->>{X}].
X[isAddressedBy->>{Y}] :- Y[address->>{X}].

X[suggests->>{Y}] :- Y[isSuggestedBy->>{X}].
X[isSuggestedBy->>{Y}] :- Y[suggests->>{X}].

X[results->Y] :- Y[isResulted->>{X}].
X[isResulted->>{Y}] :- Y[results->X].

X[presents->>{Y}] :- Y[isPresentedBy->X].
X[isPresentedBy->Y] :- Y[presents->>{X}].

X[creates->>{Y}] :- Y[isCreatedBy->>{X}].
X[isCreatedBy->>{Y}] :- Y[creates->>{X}].

X[makes->>{Y}] :- Y[isMadeBy->>{X}].
X[isMadeBy->>{Y}] :- Y[makes->>{X}].

X[describes->>{Y}] :- Y[isDescribedBy->X].
X[isDescribedBy->Y] :- Y[describes->>{X}].

X[prescribes->>{Y}] :- Y[isPrescribedBy->>{X}].
X[isPrescribedBy->>{Y}] :- Y[prescribes->>{X}].

X[concludes->Y] :- Y[isConcludedBy->>{X}].
X[isConcludedBy->>{Y}] :- Y[concludes->X].

X[performs->>{Y}] :- Y[isPerformedBy->>{X}].
X[isPerformedBy->>{Y}] :- Y[performs->>{X}].

X[executes->>{Y}] :- Y[isExecutedBy->>{X}].
X[isExecutedBy->>{Y}] :- Y[executes->>{X}].

X[requires->>{Y}] :- Y[isRequiredBy->>{X}].
X[isRequiredBy->>{Y}] :- Y[requires->>{X}].
```

Axiomas

```
/*****************************************************************
** File:    kuaba_axioms.flr
** Author:  Adriana P. de Medeiros
** Contact: adri@inf.puc.rio.br
**
** Copyright (C) Adriana P. de Medeiros, 2005
**
*****************************************************************/

//================================================================
// AXIOMS
//================================================================

Person[performs->>{Role}]
    :- Person:person[executes->>{Activity}],
       Activity:activity[requires->>{Role:role}].

Idea[hasArgument->>{Argument}]
    :- (Argument:argument[inFavorOf->>{Idea:idea}],
        not Argument[objectsTo->>{Idea}]).

Decision2[isAccepted->'false']
    :- _Question:question[hasType->'XOR',
                         isAddressedBy->>{Idea1,Idea2},
                         hasDecision->>{Decision1,Decision2}],
       Decision1:decision[isAccepted->'true',
                         concludes->Idea1:idea],
       Decision2:decision[concludes->Idea2:idea],
       not (Idea1=Idea2).

Decision2[isAccepted->'false']
    :- _Question:question[hasType->'AND',
                         isAddressedBy->>{Idea1,Idea2},
                         hasDecision->>{Decision1,Decision2}],
       Decision2:decision[concludes->Idea2:idea],
       Decision1:decision[isAccepted->"false",
                         concludes->Idea1:idea],
       not (Idea1=Idea2).
```

Apêndice C - Implementação da Operação de União em Flora-2

```
/*****************************************************************
** File:      union.flr
** Author:    Adriana P. de Medeiros
** Contact:   adri@inf.puc.rio.br
**
** Version: 2.0
**
** Copyright (C) Adriana P. de Medeiros, 2006
*****************************************************************/

/*
** Loads the kuaba ontology and the instances in their respective
** modules. The base representation is always loaded in the mod2
*/

?- [kuaba_ontology>>mod1,+kuaba_rules>>mod1,+kuaba_axioms>>mod1,
    +kuaba_facts1>>mod1].
?- [kuaba_ontology>>mod2,+kuaba_rules>>mod2,+kuaba_axioms>>mod2,
    +kuaba_facts2>>mod2].

/*
** Loads the rules that define the operations
*/

?- [+operations].

/*
** Defines the equivalence between the domain information items
*/

category:=:genre.
categoryName:=:genreName.
hwModelCategory:=:hwModelGenre.
hwModelNamecategory:=:hwModelGenreName.

/*
** Verify the equivalence between the ideas of the representations
** being integrated and copy the arguments of the equivalent ideas
** for the base representation
*/

?- L=collectset{I|I:idea@mod1},
   L2=collectset{I|I:idea@mod2},
   verify_equivalence(L,L2,List).

/*
** Copy the questions that are not equivalent for the base
** representation
*/

?- L=collectset{Q|Q[isSuggestedBy->>I]@mod1},
   L2=collectset{Q|Q[isSuggestedBy->>I]@mod2},
   get_non_equivalent_question(L,L2,List), copy_question(List).
/*
** Copy the ideas that are not equivalent for the base
```

```
** representation
*/

?- L=collectset{I|I:idea@mod1},
   L2=collectset{I|I:idea@mod2},
   get_non_equivalent_idea(L,L2,List), copy_idea(List).

/*
** Treat the equivalent questions ---------------
*/

?- L=collectset{Q|Q:question@mod1},
   L2=collectset{Q|Q:question@mod2},
   verify_equivalence_question(L,L2,List).

/*
** Delete all references to decisions of the elements of the type
** "question"
*/

?- delete{Q[hasDecision->>X]@mod2}.

/*
** Generate the new representation file ---------------
*/

?- reasoning_element[#pp_class(mod2,
                    'new_representation.flr')]@flora(pp).

?- [kuaba_ontology>>mod3,+new_representation>>mod3].

?- delete{I[isConcludedBy->>D]@mod3}.

?- reasoning_element[#pp_class(mod3,
                    'new_representation.flr')]@flora(pp).
```

Regras

```
/*****************************************************************
** File:      operations.flr
** Author:    Adriana P. de Medeiros
** Contact:   adri@inf.puc.rio.br
**
** Version: 2.0
** Copyright (C) Adriana P. de Medeiros, 2005
**
*****************************************************************/

/*
** Defines rules to find the equivalent elements between two
** representations
*/

// Equivalence Rules: IDEAS

/*
** Domain ideas (that are not defined by a formal model) are
** considered equivalent if they have equal texts, or they were
** specified as equivalent ideas by the designer, and they address
** questions with equal texts or questions that were specified as
** equivalent questions by the designer
*/

equivalent_domainIdea(Idea1,Idea2) :- Idea1[not (isDefinedBy->_X),
                                      address->>Q1[hasText->_TQ1],
                                      hasText->_TI1]@mod1,
                                      Idea2[not (isDefinedBy->_Y),
```

```
                                        address->>Q2[hasText->_TQ2],
                                        hasText->_TI2]@mod2,
                                        (_TI1=_TI2;Idea1:=:Idea2),
                                        (_TQ1=_TQ2;_Q1:=:_Q2),
                                         not(_Q1=_Q2).

/*
** Design ideas (that are defined by a formal model) are
** considered equivalent if they have equal texts and they
** address at least one same equivalent question
*/

equivalent_designIdea(Idea1,Idea2) :-
                Idea1[hasText->_TI1]@mod1,
                Idea2[hasText->_TI2]@mod2,
                TI1=_TI2,
                LQ1=collectset{Q1|Q1[isAddressedBy->>Idea1]@mod1},
                LQ2=collectset{Q2|Q2[isAddressedBy->>Idea2]@mod2},
                get_equivalent_question(LQ1,LQ2,List),
                not(List = []).

/*
** Create a list of the equivalent questions between the
** representations
*/

get_equivalent_question([], _, []).

get_equivalent_question([Head|Tail], List2, [Head|T]) :-
                has_equivalent_question(List2, Head),
                get_equivalent_question(Tail, List2, T).

get_equivalent_question([Head|Tail], List2, T) :-
                not(has_equivalent_question(List2, Head)),
                get_equivalent_question(Tail, List2, T).

// Equivalence Rules: QUESTIONS

/*
** Questions are considered equivalent if they have equal texts
** and they are suggested by equivalent ideas
*/

equivalent_question(Question1, Question2) :-
               Question1[hasText->_TQ1, isSuggestedBy->>_I1]@mod1,
               Question2[hasText->_TQ2, isSuggestedBy->>_I2]@mod2,
               (_TQ1=_TQ2;Question1:=:Question2),
               equivalent_domainIdea(_I1,_I2);
               equivalent_designIdea(_I1,_I2);
               _I1:=:_I2).

/*
** Transfer the arguments of the ideas that are equivalent to the
** base representation
*/

verify_equivalence_idea([], _, []).

verify_equivalence_idea([Head|Tail], List2, [Head|T]) :-
                        treat_equivalent_idea(List2,Head),
                        verify_equivalence_idea(Tail, List2, T).

verify_equivalence_idea([Head|Tail], List2, T) :-
                        not(treat_equivalent_idea(List2, Head)),
                        verify_equivalence_idea(Tail, List2, T).

treat_equivalent_idea([Element1|_],Element) :-
```

```
                    equivalent_domainIdea(Element,Element1);
                    equivalent_designIdea(Element,Element1);
                    Element1:=:Element),
                    LA=collectset{A1|A1[inFavorOf->>Element]@mod1;
                                    A1[objectsTo->>Element]@mod1},
                    transfer_argument(LA, Element1, Element)

treat_equivalent_idea([_|Tail],Element):-
                            treat_equivalent_idea(Tail, Element).

/*
** Copy the arguments related to the ideas that are equivalent for
** the representation base
*/

transfer_argument([Argument1|Tail], Base, Idea) :-
    insert{(Argument1:argument,Argument1[hasText->X,
            hasCreationDate->Y,isInvolved->Z])@mod2 |
            Argument1[hasText->X, hasCreationDate->Y,
            isInvolved->Z]@mod1},
    insert{Base[hasArgument->>{Argument1}]@mod2},
    /*
     ** This delete before the insert is necessary. Otherwise,
     ** Flora copies all arguments of the Idea@mod1 to the
     ** Base@mod2.
    */
    if Argument1[inFavorOf->>Idea]@mod1
    then
      (delete{Argument1[inFavorOf->>Idea]@mod1},
       insert{Argument1[inFavorOf->>Base]@mod2}),

    if Argument1[objectsTo->>Idea]@mod1
    then
      (delete{Argument1[objectsTo->>Idea]@mod1},
       insert{Argument1[objectsTo->>Base]@mod2}),

    transfer_argument(Tail, Base, Idea).

/*
** Create a list of the non-equivalent questions between the
** representations
*/

get_non_equivalent_question([], _, []).

get_non_equivalent_question([Head|Tail], List2, [Head|T]) :-
                      not(has_equivalent_question(List2, Head)),
                      get_non_equivalent_question(Tail, List2, T).

get_non_equivalent_question([Head|Tail], List2, T) :-
                      has_equivalent_question(List2, Head),
                      get_non_equivalent_question(Tail, List2, T).

has_equivalent_question([Element1|_], Element) :-
                      equivalent_question(Element,Element1).

has_equivalent_question([_|Tail], Element) :-
                      has_equivalent_question(Tail, Element).

/*
** Create a list of the non-equivalent ideas between the
** representations
*/

get_non_equivalent_idea([], _, []).
```

```
get_non_equivalent_idea([Head|Tail], List2, [Head|T]) :-
                      not(has_equivalent_idea(List2, Head)),
                      get_non_equivalent_idea(Tail, List2, T).

get_non_equivalent_idea([Head|Tail], List2, T) :-
                      has_equivalent_idea(List2, Head),
                      get_non_equivalent_idea(Tail, List2, T).

has_equivalent_idea([Element1|_], Element) :-
                      (equivalent_domainIdea(Element,Element1);
                       equivalent_designIdea(Element,Element1);
                       Element1:=:Element).

has_equivalent_idea([_|Tail], Element) :-
                       has_equivalent_idea(Tail, Element).

/*
** Transfer the relations of the equivalent questions to the base
** representation
*/

verify_equivalence_question([], _, []).

verify_equivalence_question([Head|Tail], List2, [Head|T]) :-
                      treat_equivalent_question(List2,Head),
                      verify_equivalence_question(Tail, List2, T).

verify_equivalence_question([Head|Tail], List2, T) :-
                      not(treat_equivalent_question(List2, Head)),
                      verify_equivalence_question(Tail, List2, T).

treat_equivalent_question([Element1|_],Element) :-
                    equivalent_question(Element,Element1);
                    Element1:=:Element),
                    LI1=collectset{I1|I1[address->>Element]@mod1},
                    LI2=collectset{I2|I2:idea@mod2},
                    identify_equivalent_idea(LI2, LI1, Element1).

treat_equivalent_question([_|Tail],Element) :-
                    treat_equivalent_question(Tail, Element).

/*
** Transfer questions to the base representation
*/

identify_equivalent_question([], _, Idea).

identify_equivalent_question([Head|Tail], List1, Idea) :-
                   treat_question(List1,Head, Idea),
                   identify_equivalent_question(Tail, List1, Idea).

identify_equivalent_question([Head|Tail], List2, Idea) :-
                   not(treat_question(List1, Head, Idea)),
                   identify_equivalent_question(Tail, List1, Idea).

treat_question([Question1|_],Question2, Idea) :-
                if equivalent_question(Question1,Question2)
                then
                  (delete{Idea[address->>{Question1}]@mod2 |
                          Idea[address->>Question1]@mod1},
                   insert{Idea[address->>{Question2}]@mod2 |
                          Idea[address->>Question1]@mod1}).

treat_question([_|Tail],Element, Idea) :-
                treat_question(Tail, Element, Idea).

/*
** Transfer ideas to the base representation
*/
```

```
identify_equivalent_idea([], _, Question).

identify_equivalent_idea([Head|Tail], List1, Question) :-
                  treat_idea(List1,Head,Question),
                  identify_equivalent_idea(Tail, List1,Question).

identify_equivalent_idea([Head|Tail], List1, Question) :-
                  not(treat_idea(List1, Head, Question)),
                  identify_equivalent_idea(Tail, List1, Question).

treat_idea([Idea1|_],Idea2, Question) :-
                 if (equivalent_domainIdea(Idea1, Idea2);
                     equivalent_designIdea(Idea1,Idea2);
                     Idea1:=:Idea2)
                 then
                   insert{Idea2[address->>{Question}]@mod2 |
                          Idea1[address->>Question]@mod1}.

treat_idea([_|Tail],Element, Question) :-
                    treat_idea(Tail, Element, Question).

/*
** Copy the questions suggested by an idea ----------------------
*/

copy_question([]).

copy_question([Question1|Tail]) :-
    insert{(Question1:question, Question1[hasText->X, hasType->T,
           hasCreationDate->Y,isInvolved->Z,isDefinedBy->W])@mod2|
            Question1[hasText->X, hasCreationDate->Y, hasType->T,
            isInvolved->Z, isDefinedBy->W]@mod1},
    LI1=collectset{I|I[address->>Question1]@mod1},
    LI2=collectset{I|I:idea@mod2},
    identify_equivalent_idea(LI2,LI1,Question1),
    copy_question(Tail).

/*
** Copy ideas --------------------------------
*/
copy_idea([]).

copy_idea([Idea1|Tail]) :-

    if not(Idea1:idea@mod2)
    then
      (insert{(Idea1:idea,Idea1[hasText->X, hasCreationDate->Y,
         isInvolved->Z])@mod2|Idea1[hasText->X,hasCreationDate->Y,
         isInvolved->Z]@mod1},
       insert{Idea1[address->>{Q}]@mod2|Idea1[address->>Q]@mod1},
       insert{Idea1[hasArgument->>{A}]@mod2 |
              Idea1[hasArgument->>A]@mod1},
       LA=collectset{A1|A1[inFavorOf->>Idea1]@mod1;
                     A1[objectsTo->>Idea1]@mod1},
       copy_argument(LA,Idea1),
       LQ1=collectset{Q1|Q1[isAddressedBy->>Idea1]@mod1},
       LQ2=collectset{Q2|Q2:question@mod2},
       identify_equivalent_question(LQ2, LQ1, Idea1),

       if Idea1[suggests->>_X]@mod1
       then
         (insert{Idea1[suggests->>{Q1}]@mod2 |
                 Idea1[suggests->>Q1]@mod1}),
       if Idea1[isDefinedBy->M]@mod1
       then
         (insert{Idea1[isDefinedBy->M]@mod2 |
```

```
                Idea1[isDefinedBy->M]@mod1}))
       else
         (insert{Idea1[address->>{Q}]@mod2 |
                 Idea1[address->>Q]@mod1}),

       copy_idea(Tail).

/*
** Copy arguments --------------------------------
*/

copy_argument([], _).

copy_argument([Argument1|Tail], Idea) :-
    insert{(Argument1:argument,Argument1[hasText->X,
            hasCreationDate->Y, isInvolved->Z])@mod2 |
            Argument1[hasText->X, hasCreationDate->Y,
            isInvolved->Z]@mod1},

    if Argument1[inFavorOf->>Idea]@mod1
    then
      insert{Argument1[inFavorOf->>Idea]@mod2},

    if Argument1[objectsTo->>Idea]@mod1
    then
      insert{Argument1[objectsTo->>Idea]@mod2},

    copy_argument(Tail, Idea).
```